z/OS

IBM

XL C/C++
Runtime Library Reference

Version 2 Release 2

SC14-7314-07

Note

Before using this information and the product it supports, read the information in "Notices" on page 2161.

This edition applies to Version 2 Release 2 of z/OS (5650-ZOS) and to all subsequent releases and modifications until otherwise indicated in new editions.

Contents

Figures

Tables

About this document

This document contains reference information that is intended to help you use the header files, macros, and functions provided by the IBM® z/OS® XL C/C++ runtime to write C or C++ applications.

This document contains terminology, maintenance, and editorial changes. Technical changes or additions to the text and illustrations are indicated by a vertical line (|) to the left of the change.

You may notice changes in the style and structure of some of the contents in this document; for example, headings that use uppercase for the first letter of initial words only, and procedures that have a different look and format. The changes are ongoing improvements to the consistency and retrievability of information in our documents.

Who should read this document

This document is intended for application programmers interested in writing C and C++ applications using the z/OS XL C/C++ runtime.

How to read syntax diagrams

This section describes how to read syntax diagrams. It defines syntax diagram symbols, items that may be contained within the diagrams (keywords, variables, delimiters, operators, fragment references, operands) and provides syntax examples that contain these items.

Syntax diagrams pictorially display the order and parts (options and arguments) that comprise a command statement. They are read from left to right and from top to bottom, following the main path of the horizontal line.

For users accessing the Information Center using a screen reader, syntax diagrams are provided in dotted decimal format.

The following symbols may be displayed in syntax diagrams:

Symbol	**Definition**
▸▸——	Indicates the beginning of the syntax diagram.
——▸	Indicates that the syntax diagram is continued to the next line.
▸——	Indicates that the syntax is continued from the previous line.
——▸◂	Indicates the end of the syntax diagram.

Syntax diagrams contain many different items. Syntax items include:
- Keywords - a command name or any other literal information.
- Variables - variables are italicized, appear in lowercase, and represent the name of values you can supply.
- Delimiters - delimiters indicate the start or end of keywords, variables, or operators. For example, a left parenthesis is a delimiter.

- Operators - operators include add (+), subtract (-), multiply (*), divide (/), equal (=), and other mathematical operations that may need to be performed.
- Fragment references - a part of a syntax diagram, separated from the diagram to show greater detail.
- Separators - a separator separates keywords, variables or operators. For example, a comma (,) is a separator.

Note: If a syntax diagram shows a character that is not alphanumeric (for example, parentheses, periods, commas, equal signs, a blank space), enter the character as part of the syntax.

Keywords, variables, and operators may be displayed as required, optional, or default. Fragments, separators, and delimiters may be displayed as required or optional.

Item type
Definition

Required
Required items are displayed on the main path of the horizontal line.

Optional
Optional items are displayed below the main path of the horizontal line.

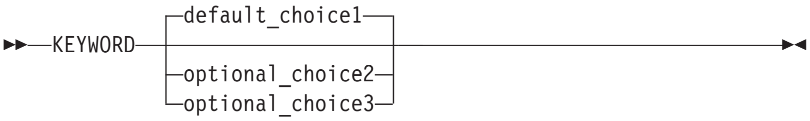
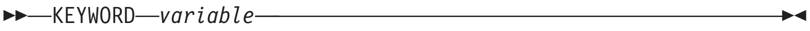
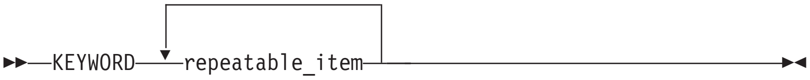
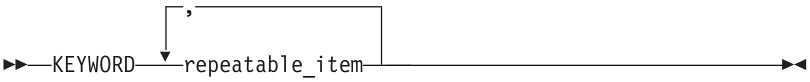
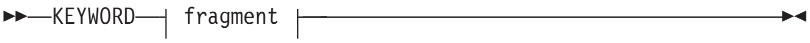
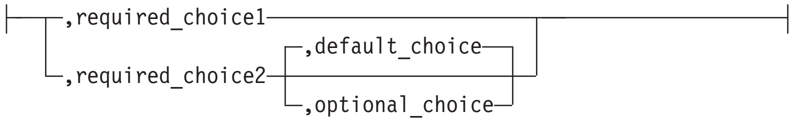
Default
Default items are displayed above the main path of the horizontal line.

The following table provides syntax examples.

Table 1. Syntax examples

Item	Syntax example
Required item. Required items appear on the main path of the horizontal line. You must specify these items.	►►─KEYWORD─required_item─────────────────────►◄
Required choice. A required choice (two or more items) appears in a vertical stack on the main path of the horizontal line. You must choose one of the items in the stack.	►►─KEYWORD─┬─required_choice1─┬──────────────►◄ └─required_choice2─┘
Optional item. Optional items appear below the main path of the horizontal line.	►►─KEYWORD─┬───────────────┬─────────────────►◄ └─optional_item─┘
Optional choice. An optional choice (two or more items) appears in a vertical stack below the main path of the horizontal line. You may choose one of the items in the stack.	►►─KEYWORD─┬──────────────────┬──────────────►◄ ├─optional_choice1─┤ └─optional_choice2─┘

Table 1. Syntax examples (continued)

Item	Syntax example
Default. Default items appear above the main path of the horizontal line. The remaining items (required or optional) appear on (required) or below (optional) the main path of the horizontal line. The following example displays a default with optional items.	 KEYWORD default_choice1 optional_choice2 optional_choice3
Variable. Variables appear in lowercase italics. They represent names or values.	
Repeatable item. An arrow returning to the left above the main path of the horizontal line indicates an item that can be repeated. A character within the arrow means you must separate repeated items with that character. An arrow returning to the left above a group of repeatable items indicates that one of the items can be selected,or a single item can be repeated.	 
Fragment. The fragment symbol indicates that a labelled group is described below the main syntax diagram. Syntax is occasionally broken into fragments if the inclusion of the fragment would overly complicate the main syntax diagram.	 **fragment:**  ,required_choice1 ,required_choice2 ,default_choice ,optional_choice

Where to find more information

For an overview of the information associated with z/OS, see *z/OS V2R2 Information Roadmap*.

Information updates on the web

For the latest information updates that have been provided in PTF cover letters and documentation APARs for z/OS, see the online document z/OS APAR book (http://publibz.boulder.ibm.com/cgi-bin/bookmgr_OS390/Shelves/ ZDOCAPAR).

This document is updated weekly and lists documentation changes before they are incorporated into z/OS publications.

The z/OS Basic Skills Information Center

The z/OS Basic Skills Information Center is a Web-based information resource intended to help users learn the basic concepts of z/OS, the operating system that runs most of the IBM mainframe computers in use today. The Information Center is designed to introduce a new generation of Information Technology professionals to basic concepts and help them prepare for a career as a z/OS professional, such as a z/OS system programmer.

Specifically, the z/OS Basic Skills Information Center is intended to achieve the following objectives:

- Provide basic education and information about z/OS without charge
- Shorten the time it takes for people to become productive on the mainframe
- Make it easier for new people to learn z/OS.

To access the z/OS Basic Skills Information Center, open your Web browser to the following Web site, which is available to all users (no login required): z/OS Basic Skills in IBM Knowledge Center (http://www.ibm.com/support/knowledgecenter/zosbasics/com.ibm.zos.zbasics/homepage.html)

How to send your comments to IBM

We appreciate your input on this documentation. Please provide us with any feedback that you have, including comments on the clarity, accuracy, or completeness of the information.

Use one of the following methods to send your comments:

Important: If your comment regards a technical problem, see instead "If you have a technical problem."

- Send an email to mhvrcfs@us.ibm.com.
- Send an email from the "Contact us" web page for z/OS (http://www.ibm.com/systems/z/os/zos/webqs.html).

Include the following information:

- Your name and address
- Your email address
- Your phone or fax number
- The publication title and order number:
  z/OS XL C/C++ Runtime Library Reference
  SC14-7314-07
- The topic and page number or URL of the specific information to which your comment relates
- The text of your comment.

When you send comments to IBM, you grant IBM a nonexclusive right to use or distribute the comments in any way appropriate without incurring any obligation to you.

IBM or any other organizations use the personal information that you supply to contact you only about the issues that you submit.

If you have a technical problem

Do not use the feedback methods that are listed for sending comments. Instead, take one or more of the following actions:

- Visit the IBM Support Portal (support.ibm.com).
- Contact your IBM service representative.
- Call IBM technical support.

New and updated information in z/OS Version 2 Release 2

This document contains terminology, maintenance, and editorial changes, including updates to header files, macros, and functions.

New information

- New functions:
  - printf() extended support, see "fprintf(), printf(), sprintf() — Format and write data" on page 588
  - scanf() extended support, see "fscanf(), scanf(), sscanf() — Read and format data" on page 625

Changed information

- The following header files were updated:
  - "stdio.h" on page 68 has been updated to support the __BLOCKED macro.
- The following functions were updated:
  - "exec functions" on page 436
  - "getaddrinfo() — Get address information" on page 684
  - "__librel() — Query release level" on page 968
  - "__osname() — Get true operating system name" on page 1166
  - "pthread_security_np(), pthread_security_applid_np() — Create or delete thread-level security" on page 1324
  - "spawn(), spawnp() — Spawn a new process" on page 1693
  - "system() — Execute a command" on page 1805
  - "uname() — Display current operating system name" on page 1951

Chapter 1. About IBM z/OS XL C/C++

For an introduction to z/OS XL C/C++ and for information on changes to z/OS XL C/C++, see *z/OS XL C/C++ User's Guide*.

AMODE 64 considerations

AMODE 64 restrictions are indicated as a dependency or as a restriction note. For example:

Restriction: This header file is not supported in AMODE 64.

Following are restrictions for AMODE 64:

- Hiperspace™ is not supported.
- The SPC facility is not supported.
- The following header files are not supported:
  - csp.h
  - ims.h
  - leawi.h
  - mtf.h
  - re_comp.h
  - regexp.h
  - spc.h
- The following feature test macros are obsolete:
  - _LARGE_FILES
  - _LARGE_MEM

  See the feature test macro description for more details.
- The following feature test macros are not supported:
  - __LIBASCII
  - _OE_SOCKETS
- The following functions are not supported:
  - advance()
  - brk()
  - compile()
  - __console()
  - __csplist
  - ctdli()
  - fortrc()
  - __openMvsRel()
  - __pcblist
  - re_comp()
  - re_exec()
  - regcmp()
  - regex()

  - sbrk()
  - sock_debug_bulk_perf0()
  - sock_do_bulkmode()
  - step()
  - tsched()
  - tsetsubt()
  - tsyncro()
  - tterm()
  - valloc()
- The following external variables are not supported:
  - __loc1
  - loc1
  - loc2
  - locs

Chapter 2. Header files

This part describes each header file, explains its contents, and lists the functions that use the file. The function descriptions are described in Chapter 3, “Library functions,” on page 89.

The header files provided with the z/OS XL C/C++ Library contain macro and constant definitions, type definitions, and function declarations. Some functions require definitions and declarations from header files to work correctly. The inclusion of header files is optional, as long as the necessary statements from the header files are coded directly into the source.

The C/C++ header files are shipped in the CEE.SCEEH* data sets and in the /usr/include directory in the Hierarchical File System (HFS).

The following header files are not supported in AMODE 64:

- csp.h
- ims.h
- leawi.h
- mtf.h
- re_comp.h
- regexp.h
- spc.h

Use the `#include` directive to select header files to include with your application, for example, **`#include <stdio.h>`**.

For information about the `#include` directive, see *z/OS XL C/C++ Language Reference* and *z/OS XL C/C++ User's Guide*.

Feature test macros

Many of the symbols that are defined in headers are “protected” by a feature test macro. These “protected” symbols are invisible to the application unless the user defines the feature test macro with `#define`, using either of the following methods:

- In the source code before including any header files.
- On the compilation command.

Note that the LANGLVL compiler option does not define or undefine these macros.

The following feature test macros are obsolete in AMODE 64:

- _LARGE_FILES
- _LARGE_MEM

See the feature test macro description for more details.

The following feature test macros are not supported in AMODE 64:

- __LIBASCII
- _OE_SOCKETS

Table 2 summarizes the relationships between the feature test macros and the standards. ‘Yes’ indicates that a feature test macro makes visible the symbols related to a standard.

Feature test macros that do not apply to POSIX standards are not listed in this table.

Table 2. Feature Test Macros and Standards

Feature Test Macro	POSIX.1	POSIX.1a	POSIX.2	POSIX.4a	XPG4.2	XPG4.2 Ext	SUSv3
_ALL_SOURCE	Yes	Yes	Yes	Yes	Yes	Yes	
_ALL_SOURCE_NO_THREADS	Yes	Yes	Yes		Yes	Yes	
_OE_SOCKETS	Yes	Yes					
_OPEN_DEFAULT	Yes	Yes	Yes				
_OPEN_SOURCE 1	Yes	Yes	Yes	Yes			
_OPEN_SOURCE 2	Yes	Yes	Yes	Yes	Yes	Yes	
_OPEN_SOURCE 3	Yes	Yes	Yes	Yes	Yes	Yes	
_OPEN_SYS	Yes	Yes	Yes	Yes			
_OPEN_SYS_IPC_EXTENSIONS	Yes	Yes			Yes		
_OPEN_SYS_PTY_EXTENSIONS	Yes	Yes			Yes	Yes	
_OPEN_SYS_SOCK_EXT	Yes	Yes		Yes	Yes		
_OPEN_THREADS	Yes	Yes		Yes			
_POSIX1_SOURCE 1	Yes						
_POSIX1_SOURCE 2	Yes	Yes					
_POSIX_C_SOURCE 1	Yes						
_POSIX_C_SOURCE 2	Yes		Yes				
_POSIX_C_SOURCE 200112L	Yes	Yes	Yes				Yes
_POSIX_SOURCE	Yes						
_XOPEN_SOURCE	Yes		Yes		Yes		
_XOPEN_SOURCE_EXTENDED 1	Yes		Yes		Yes	Yes	
_XOPEN_SOURCE 500	Yes		Yes		Yes	Yes	
_XOPEN_SOURCE 600	Yes	Yes	Yes		Yes	Yes	Yes

The following feature test macros are supported:

_ALL_SOURCE

This feature test macro exposes the following namespaces: POSIX.1, POSIX.1a, POSIX.2, POSIX.4a draft 6, XPG4, and XPG4.2, as well as additions to z/OS UNIX drawn from Single UNIX Specification, Version 2.

In addition, defining _ALL_SOURCE makes visible a number of symbols which are not permitted under ANSI, POSIX or XPG4, but which are provided as an aid to porting C-language applications to z/OS UNIX System Services. Extensions made visible with the following feature test macros are implicit in _ALL_SOURCE:

- _OPEN_SYS_DIR_EXT
- _OPEN_SYS_EXT
- _OPEN_SYS_IPC_EXTENSIONS
- _OPEN_SYS_MAP_EXTENTION
- _OPEN_SYS_PTY_EXTENSIONS
- _OPEN_SYS_SOCK_EXT
- _OPEN_SYS_SOCK_EXT2
- _OPEN_SYS_SOCK_IPV6

If _OPEN_THREADS is not explicitly defined in the application, _ALL_SOURCE will define _OPEN_THREADS 1 except when any of the following are present:

- _ALL_SOURCE_NO_THREADS
- _UNIX03_THREADS
- _XOPEN_SOURCE 600

_ALL_SOURCE does not expose functionality first introduced in Single UNIX Specification, Version 3 under macro definitions _XOPEN_SOURCE 600 or _POSIX_C_SOURCE 200112L, although it does tolerate interfaces made visible by defining _OPEN_THREADS to 2 or 3.

In order to stabilize the namespace, no future extensions, whether POSIX, XOPEN, or MVS™, will expand the definition of _ALL_SOURCE. Any future enhancement will require new, explicit feature test macros to add symbols to this namespace.

_ALL_SOURCE_NO_THREADS

This feature test macro provides the same function as _ALL_SOURCE, except it does not expose threading services (_OPEN_THREADS).

_IEEEV1_COMPATIBILITY

In 1999, the C/C++ Runtime Library provided IEEE754 floating-point arithmetic support in support of the IBM Java group. The Java™ language had a bit-wise requirement for its math library, meaning that all platforms needed to produce the same results as Sun Microsystems' fdlibm (Freely Distributed LIBM) library. Therefore, Sun Microsystems' fdlibm code was ported to the C/C++ Runtime Library to provide IEEE754 floating-point arithmetic support. Subsequent to the C/C++ Runtime Library's 1999 release of IEEE754 floating-point math support, IBM's Java group provided their own support of IEEE754 floating point arithmetic and no longer use the C/C++ Runtime Library for this support.

Beginning in z/OS V1R9, a subset of the original fdlibm functions are being replaced by new versions that are designed to provide improved performance and accuracy. The new versions of these functions are replaced at the existing entry points. However, as a migration aid, IBM has provided new entry points for the original fdlibm versions. Applications that take no action will automatically use the updated functions. There are two methods for accessing the original functions.

This feature test macro provides an environment for the following C/C++ functions:

- do not include <math.h>
- include <math.h> and define the _FP_MODE_VARIABLE feature test macro

Either of the above will cause the application to be running in what is called "variable" mode with respect to floating-point math functions called within the compile unit. See z/OS XL C/C++ Programming Guide for more details on the environment variable.

The second method is through a feature test macro, described here, that can be used by applications that do include <math.h> and do not define the _FP_MODE_VARIABLE feature test macro.

If the application conforms to the rules of the second method, then the feature test macro can be used to access the original fdlibm versions of the following functions:

```
acos(), acosh(), asin(), asinh(), atan(), atanh(), atan2(),
cbrt(), cos(), cosh(), erf(), erfc(), exp(), expm1(), gamma(),
hypot(), lgamma(), log(), log1p(), log10(), pow(), rint(), sin(),
sinh(), tan(), tanh()
```

A recompile and relink of the application is required to access the original fdlibm versions.

`_ISOC99_SOURCE`

This feature test macro makes available all interfaces associated with ISO/IEC 9899:1999 except for interfaces requiring a compiler that is designed to support C99. This feature test macro also exposes the namespace normally exposed by the _MSE_PROTOS feature test macro, unless _NOISOC99_SOURCE is defined. The _ISOC99_SOURCE feature test macro is not required when a compiler that is designed to support C99 is used.

Note: If both _NOISOC99_SOURCE and _ISOC99_SOURCE are defined before inclusion of the first header, new C99 interfaces will not be exposed.

`_LARGE_FILES`

The _LARGE_FILES feature test macro enables certain functions to operate on MVS data sets and z/OS UNIX files that are larger than 2 GB in size and VSAM extended addressability data sets larger than 4 GB in size. When this feature test macro is selected it must be used in conjunction with the compiler option LANGLVL(LONGLONG) to activate the `long long` data type.

The following functions are enabled to operate on z/OS UNIX files of all sizes by expanding appropriate offset and file size values to a 64-bit value:

```
creat(), fcntl(), fgetpos(), fopen(), freopen(), fseek(), fseeko(),
fsetpos(), fstat(), ftell(), ftello(), ftruncate(), getrlimit(),
lockf(), lseek(), lstat(), mmap(), open(), read(), setrlimit(),
stat(), truncate(), write()
```

The _LARGE_FILES feature test macro also enables the fseeko() and ftello() functions to operate on MVS data sets larger than 2 GB and VSAM data sets larger than 4 GB by expanding the parameter and return type width to 64 bits. The fgetpos(), fopen(), freopen(), and fsetpos() functions implicitly support operations on these sized data sets and therefore do not require the _LARGE_FILES feature test macro to be defined.

Note: Using AMODE 64 obsoletes this feature test macro. Large files are automatically supported in the LP64 programming model, therefore automatically included for AMODE 64 C/C++ applications. The c99 compiler defines the `long long` data type as a standard type by default.

Restriction: This feature test macro is incompatible with the __LIBASCII feature test macro.

`_LARGE_MEM`

This feature test macro is provided for AMODE 31 applications that need access to AMODE 64 values. Use of large memory support requires LANGLVL(LONGLONG).

Note: This feature test macro is obsolete in AMODE 64. Large memory support is automatic in the LP64 programming model, therefore all behaviors with respect to large memory are automatically included for AMODE 64 C/C++ applications.

`_LARGE_TIME_API`

This feature test macro exposes new typedefs, structures, and functions so that an application can work with constructed calendar times up to and including the artificial limit of 23:59:59 on December 31, 9999 UTC.

`__LIBASCII`

This feature test macro provides an ASCII-like environment for the following C/C++ functions:

```
access(), asctime(), atof(), atoi(), atol(), chdir(), chmod(),
chown(), creat(), ctime(), dllload(), dllqueryfn(), dynalloc()
ecvt(), execv(), execve(), execvp(), fcvt(), fdopen(), fopen(),
freopen(), ftok(), gcvt(), getcwd(), getenv(), getgrnam(),
gethostbyaddr(), gethostbyname(), gethostname(), getlogin(),
getopt(), getpass(), getpwnam(), getpwuid(), getservbyname(),
getwd(), inetaddr(), inet_ntoa(),isalnum(), isalpha(), iscntrl(),
isdigit(), isgraph(), islower(), isprint(), ispunct(), isspace(),
isupper(), isxdigit(), link(), localeconv(), mbstowcs(), mbtowc(),
mkdir(), mknod(), mktemp(), nl_langinfo(), open(), opendir(),
perror(), popen(), ptsname(), putenv(), readdir(), regcomp(),
remove(), rename(), rexec(), rmdir(), scanf(), setenv(),  setkey(),
setlocale(), setvbuf(), sprintf(), sscanf(),  stat(), statvfs(),
strcasecmp(), strerror(), strncasecmp(),  strtod(), strtol(),
strtoul(), system(), tempnam(), tmpnam(),  toascii(), tolower(),
toupper(), uname(), unlink(), utime(),  utimes()
```

For each application program using one or more of these functions, where the input/output is ASCII, add the following feature test macro:

- #define __LIBASCII
- Recompile using the CONV(ISO8859-1) option to cause the compiler to generate all strings defined in the source program in ASCII rather than EBCDIC format.

Note:

- This feature test macro is not supported in AMODE 64.
- Enhanced ASCII and __LIBASCII are independent, and should not be used together. Using Enhanced ASCII and __LIBASCII together is not supported.
- The libascii functions are as thread-safe as the runtime library with the exception of the getopt() function. The libascii getopt() function is not thread-safe. The second argument is changed for a short period of time from EBCDIC to ASCII and then back to EBCDIC. This feature test macro is incompatible with the _LARGE_FILES feature test macro.

`_LONGMAP`

Programs compiled with the LONGNAME compiler option and which use POSIX functions must define _LONGMAP when using the Prelinker outside of a z/OS UNIX shell environment.

`_MSE_PROTOS`

The `_MSE_PROTOS` feature test macro does the following:

1. Selects behavior for a multibyte extension support (MSE) function declared in wchar.h as specified by ISO/IEC 9899:1990/Amendment

1:1994 instead of behavior for the function as defined by CAE Specification, System Interfaces and Headers, Issue 4, July 1992 (XPG4), and

2. Exposes declaration of an MSE function declared in wchar.h which is specified by ISO/IEC 9899:1990/Amendment 1:1994 but not by XPG4.

Note: Defining _ISOC99_SOURCE or using a compiler that is designed to support C99 also exposes this namespace provided that _NOISOC99_SOURCE is not also defined.

_NOISOC99_SOURCE

This feature test macro prevents exposure of new interfaces that are part of the C99 standard. This feature test macro must be defined before inclusion of the first header in order to prevent new C99 interfaces from being exposed.

Note: If both _NOISOC99_SOURCE and _ISOC99_SOURCE are defined before inclusion of the first header, new C99 interfaces will not be exposed.

_OE_SOCKETS

This feature test macro defines a BSD-like socket interface for the function prototypes and structures involved. This can be used with `_XOPEN_SOURCE_EXTENDED` 1 and the XPG4.2 socket interfaces will be replaced with the BSD-like interfaces.

Restriction: This feature test macro is not supported in AMODE 64.

_OPEN_DEFAULT

When defined to 0, and if no other feature test macro is defined, then all symbols will be visible. If in addition to _OPEN_DEFAULT only POSIX and/or XPG4 feature test macros are defined, then only the symbols so requested will be visible. Otherwise, additional symbols (for example, those visible when the LANGLVL(EXTENDED) compiler option is specified), may be exposed.

When defined to 1, this provides the base level of z/OS UNIX functionality, which is POSIX.1, POSIX.1a and POSIX.2.

_OPEN_MSGQ_EXT

This feature test macro defines an interface which enables use of select(), selectex() and poll() to monitor message and file descriptors.

_OPEN_SOURCE

When defined to 1, this defines all of the functionality that was available on MVS 5.1. This macro is equivalent to specifying `_OPEN_SYS`.

When defined to 2, this defines all of the functionality that is available on MVS 5.2.2, including XPG4, XPG4.2, and all of the z/OS UNIX extensions.

When defined to 3, this macro is equivalent to specifying _ALL_SOURCE.

If _OPEN_THREADS is not explicitly defined in the application, _OPEN_SOURCE will define _OPEN_THREADS 1 except when any of the following are present:

- _ALL_SOURCE_NO_THREADS
- _UNIX03_THREADS
- _XOPEN_SOURCE 600

_OPEN_SYS

When defined to 1, this indicates that symbols required by POSIX.1, POSIX.1a, POSIX.2 are made visible. Any symbols defined by the _OPEN_THREADS macro are allowed.

If _OPEN_THREADS is not explicitly defined in the application, _OPEN_SYS will define _OPEN_THREADS 1 except when any of the following are present:

- _ALL_SOURCE_NO_THREADS
- _UNIX03_THREADS
- _XOPEN_SOURCE 600

Additional symbols can be made visible if any of the exposed standards explicitly allows the symbol to appear in the header in question or if the symbol is defined as a z/OS UNIX System Services extension.

_OPEN_SYS_DIR_EXT

This feature test macro defines the interface and function prototypes for __opendir2() and __readdir2().

_OPEN_SYS_FILE_EXT

When defined to any value with #define, _OPEN_SYS_FILE_EXT indicates that symbols required for file conversion, file tagging, and file attributes manipulation functionality are made visible.

_OPEN_SYS_IF_EXT

When defined to 1, this feature test macro exposes BSD-like socket definitions found in <net/if.h> and <sys/ioctl.h> that are needed to manipulate network interfaces. This feature test macro is made available beginning with z/OS V1R9.

_OPEN_SYS_IPC_EXTENSIONS

This feature test macro defines z/OS UNIX extensions to the X/Open InterProcess Communications functions. When _OPEN_SYS_IPC_EXTENSIONS is defined, the POSIX.1, POSIX.1a, and the XPG4 symbols are visible. This macro should be used in conjunction with _XOPEN_SOURCE.

_OPEN_SYS_MUTEX_EXT

This feature test macro allows pthread condition variables and mutexes in shared memory. When this feature is defined, pthread_mutex_t and pthread_cond_t will grow significantly in size.

When either _XOPEN_SOURCE 600 or _UNIX03_THREADS are defined, the namespace includes all elements made visible by the _OPEN_SYS_MUTEX_EXT macro. In this case, _OPEN_SYS_MUTEX_EXT is redundant and does not need to be defined by the application.

_OPEN_SYS_PTY_EXTENSIONS

This feature test macro defines z/OS UNIX extensions to the X/Open Pseudo TTY functions. When _OPEN_SYS_PTY_EXTENSIONS is defined, the POSIX.1, POSIX.1a, XPG4, and XPG4.2 symbols are visible. This macro should be used in conjunction with _XOPEN_SOURCE_EXTENDED 1.

_OPEN_SYS_SOCK_EXT

This feature test macro defines the interface for function prototypes and structures for the extended sockets and bulk mode support.

_OPEN_SYS_SOCK_EXT2

This feature test macro defines the function prototype and interfaces for accept_and_recv().

_OPEN_SYS_SOCK_EXT3

This feature test macro defines the function prototypes and interfaces for multicast source filtering.

_OPEN_SYS_SOCK_EXT4

This feature test macro defines interfaces in netinet/in.h used for sending UDP reply packets on the same inbound interface as the request arrived.

_OPEN_SYS_SOCK_IPV6

When defined, indicates that symbols related to Internet Protocol Version 6 (IPv6) are made visible.

Defining _XOPEN_SOURCE to 600 will expose the IPv6 symbols required in Single Unix Specification, Version 3. However, these symbols only comprise a subset of the complete namespace associated with _OPEN_SYS_SOCK_IPV6. Although an application is allowed to define both macros, such an application may not be strictly conforming to Single UNIX Specification, Version 3.

_OPEN_THREADS

When defined to 1, this indicates that symbols required by `POSIX.1`, `POSIX.1a`, and `POSIX.4a(draft 6)` are made visible.

When defined to 2, additional pthread functions introduced in z/OS V1R07 from Single UNIX Specification, Version 3 are made visible, along with those made visible when this is defined to 1. The following symbols are added to the namespace when _OPEN_THREADS is defined to 2:

Interfaces	Constants
pthread_getconcurrency()	PTHREAD_CANCEL_ENABLE
pthread_setconcurrency()	PTHREAD_CANCEL_DISABLE
pthread_setcancelstate()	PTHREAD_CANCEL_DEFERRED
pthread_setcanceltype()	PTHREAD_CANCEL_ASYNCHRONOUS
pthread_sigmask()	
pthread_testcancel()	
pthread_key_delete()	

When defined to 3, all pthread functions required for the Threads option of Single UNIX Specification, Version 3 are exposed, although behavior and function signatures are still based on the POSIX.4a draft 6 specification. In addition to the symbols exposed by _OPEN_THREADS 2, _OPEN_THREADS 3 adds the following symbols to the namespace:

Interfaces	Constants
pthread_atfork()	PTHREAD_CANCEL_CANCELED
pthread_attr_getguardsize()	PTHREAD_COND_INITIALIZER

Interfaces	Constants
pthread_attr_getschedparam()	PTHREAD_CREATE_DETACHED
pthread_attr_getstack()	PTHREAD_CREATE_JOINABLE
pthread_attr_getstackaddr()	PTHREAD_EXPLICIT_SCHED
pthread_attr_setguardsize()	
pthread_attr_setschedparam()	
pthread_attr_setstack()	
pthread_attr_setstackaddr()	

Thread interfaces listed above and first exposed by _OPEN_THREADS 2 or 3 are fully compliant with Single UNIX Specification, Version 3. However, the other threading interfaces in the library will not exhibit the new behavior or use function signatures changed in the new standard. Applications that define _UNIX03_THREADS or _XOPEN_SOURCE 600 will obtain threads support that complies fully with Single UNIX Specification, Version 3.

If _OPEN_THREADS is defined with _XOPEN_SOURCE 600, _OPEN_THREADS takes precedence and overrides the default threads behavior of _XOPEN_SOURCE 600. However, _OPEN_THREADS and _UNIX03_THREADS are mutually exclusive.

Note: Feature test macros _OPEN_SYS, _OPEN_SOURCE, and _ALL_SOURCE incorporate _OPEN_THREADS 1 by default, if _OPEN_THREADS has not been explicitly defined in the application, except when any of the following are present:

- _ALL_SOURCE_NO_THREADS
- _UNIX03_THREADS
- _XOPEN_SOURCE 600

_POSIX1_SOURCE

- When defined to 1, it has the same meaning as _POSIX_SOURCE.
- When defined to 2, both the POSIX.1a symbols and the POSIX.1 symbols are made visible. Additional symbols can be made visible if POSIX.1a explicitly allows the symbol to appear in the header in question.

_POSIX_C_SOURCE

- When defined to 1, it indicates that symbols required by POSIX.1 are made visible. Additional symbols can be made visible if POSIX.1 explicitly allows the symbol to appear in the header in question.
- When defined to 2, both the POSIX.1 and POSIX.2 symbols are made visible.
- When defined to 200112L, the Single UNIX Specification, Version 3 symbols are made visible, including POSIX.1 and POSIX.2. Since Version 3 is aligned with the ISO C standard (ISO/IEC 9899:1999), this definition of the feature test macro also exposes the C99 namespace.
- The _POSIX_C_SOURCE 200112L definition is available beginning with z/OS V1R9. Targeting earlier releases will result in an error during compile-time.
- Additional symbols can be made visible if POSIX.2 explicitly allows the symbol to appear in the header in question.

_POSIX_SOURCE

When defined to any value with `#define`, it indicates that symbols required by `POSIX.1` are made visible. Additional symbols can be made visible if `POSIX.1` explicitly allows the symbol to appear in the header in question.

_SHARE_EXT_VARS

This feature test macro provides access to POSIX and XPG4 external variables of an application from a dynamically loaded module such as a DLL. For those external variables that have a function to access a thread-specific value, it provides access to the thread-specific value of the external variable without having to explicitly invoke the function.

Individual variables can be externalized by using the feature test macros prefixed with _SHR_ and the feature test macros that are shown as follows. The entire set can be accessed by defining _SHARE_EXT_VARS.

Note: When an application is compiled with the XPLINK or LP64 option:

- The POSIX and XPG4 external variables will be resolved through the C runtime library side-deck in the SCEELIB data set and will be accessible from all dynamically loaded modules. See *z/OS XL C/C++ Programming Guide* for more details.
- The _SHARE_EXT_VARS feature test macro, and the following feature test macros with the _SHR_ prefix, are only necessary for accessing the thread-specific values without having to explicitly invoke the function.

_SHR_DAYLIGHT

To share access to the daylight external variable from a dynamically loaded module such as a DLL, define the _SHR_DAYLIGHT feature test macro and include time.h in your program source.

_SHR_ENVIRON

If you have declared char **environ in your program and want to access the environment variable array from a dynamically loaded module such as a DLL, define the _SHR_ENVIRON feature test macro and include stdlib.h in the program source.

_SHR_H_ERRNO

To share access to the h_errno external variable from a dynamically loaded module such as a DLL, define the _SHR_H_ERRNO feature test macro and include netdb.h in your program source.

_SHR__LOC1

To share access to the __loc1 external variable from a dynamically loaded module such as a DLL, define _SHR__LOC1 feature test macro and include libgen.h in your program source.

_SHR_LOC1

To share access to the loc1 external variable from a dynamically loaded module such as a DLL, define _SHR_LOC1 feature test macro and include regexp.h in your program source.

_SHR_LOC2

To share access to the loc2 external variable from a dynamically loaded module such as a DLL, define _SHR_LOC2 feature test macro and include regexp.h in your program source.

_SHR_LOCS

To share access to the locs external variable from a dynamically loaded module such as a DLL, define _SHR_LOCS feature test macro and include regexp.h in your program source.

_SHR_OPTARG

To share access to the optarg external variable from a dynamically loaded module such as a DLL, define the _SHR_OPTARG feature test macro and include unistd.h or stdio.h in your program source.

_SHR_OPTERR

To share access to the opterr external variable from a dynamically loaded module such as a DLL, define the _SHR_OPTERR feature test macro and include unistd.h or stdio.h in your program source.

_SHR_OPTIND

To share access to the optind external variable from a dynamically loaded module such as a DLL, define _SHR_OPTIND feature test macro and include unistd.h or stdio.h in your program source.

_SHR_OPTOPT

To share access to the optopt external variable from a dynamically loaded module such as a DLL, define the _SHR_OPTOPT feature test macro and include unistd.h or stdio.h in your program source.

_SHR_SIGNGAM

To share access to the signgam external variable from a dynamically loaded module such as a DLL, define the _SHR_SIGNGAM feature test macro and include math.h in your program source.

_SHR_T_ERRNO

To share access to the t_errno external variable from a dynamically loaded module such as a DLL, define the _SHR_T_ERRNO feature test macro and include xti.h in your program source.

_SHR_TIMEZONE

To share access to the timezone external variable from a dynamically loaded module such as a DLL, define the _SHR_TIMEZONE feature test macro and include time.h in your program source. To avoid name space pollution when _SHR_TIMEZONE is defined, the timezone variable must be referred to as _timezone.

_SHR_TZNAME

To share access to the tzname external variable from dynamically loaded module such as a DLL, define the _SHR_TZNAME feature test macro and include time.h in your program source.

__STDC_CONSTANT_MACROS

This feature test macro is required by C++ applications wishing to expose macros for integer constants as documented in <stdint.h>.

__STDC_FORMAT_MACROS

This feature test macro is required by C++ applications wishing to expose macros for format specifiers as documented in <inttypes.h>.

__STDC_WANT_DEC_FP__

This MACRO will be added to the C99 DFP specification (for C and C++). The user will define this MACRO when DFP support is wanted. It will cause all DFP-oriented definitions in <math.h> and other headers to be exposed if __IBM_DFP is defined.

__STDC_LIMIT_MACROS

This feature test macro is required by C++ applications wishing to expose limits of specified-width integer types and limits of other integer types as documented in <stdint.h>.

_TR1_C99

This feature test macro exposes the C++ TR1 C99 name space as described in Chapter 8 of *ISO/IEC DTR 19768: Draft Technical Report* on C++ Library Extensions.

When both the _TR1_C99 and _AIX_COMPATIBILITY Feature Test Macros are defined, the _AIX_COMPATIBILITY takes precedence. This affects the copysign(), scalbn(), and the floating point classification functions.

When both the _TR1_C99 and _FP_MODE_VARIABLE feature test macros are defined, float overloads are not supported for the following functions:

atan2(), copysign(), fdim(), fma(), fmax(), fmin(), fmod(), frexp(), hypot(), ldexp(), modf(), nextafter(), nexttoward(), pow(), remainder(), remquo(), scalbln(), and scalbn()

Also, when both the _TR1_C99 and _FP_MODE_VARIABLE feature test macros are set, the long double overloads are not supported for frexp() and ldexp().

This feature test macro requires the use of the z/OS V1.10 z/OS XL C++ compiler or later.

_UNIX03_SOURCE

This feature test macro exposes new Single UNIX Specification, Version 3 interfaces. It does not change the behavior of existing APIs, nor expose interfaces controlled by feature test macros such as _XOPEN_SOURCE_EXTENDED. Functions and behavior exposed by _UNIX03_SOURCE are a subset and not the complete implementation of the Single UNIX Specification, Version 3. To expose the full Single UNIX Specification, Version 3 implementation available in the C/C++ Runtime, see _XOPEN_SOURCE or _POSIX_C_SOURCE.

Release	Interfaces Exposed with _UNIX03_SOURCE
z/OS V1R6	dlclose(), dlerror(), dlopen(), dlsym()
z/OS V1R7	sched_yield(), strerror_r(), unsetenv()
z/OS V1R8	flockfile(), ftrylockfile(), funlockfile(), getc_unlocked(), getchar_unlocked(), putc_unlocked(), putchar_unlocked()
z/OS V1R9	posix_openpt(), pselect(), sockatmark()

Note: This feature test macro does not expose any new pthread interfaces. See _OPEN_THREADS and _UNIX03_THREADS to expose pthread interfaces.

_UNIX03_THREADS

This feature test exposes all pthread functions, function signatures, and behaviors required for the Threads option of Single UNIX Specification, Version 3. The macro is available for compiles targeting z/OS V1R9 or later.

Defining _UNIX03_THREADS exposes the content covered by feature test macro _OPEN_SYS_MUTEX_EXT, so that the latter is redundant and need not be defined with _UNIX03_THREADS.

It is not necessary to define this feature test macro, if _XOPEN_SOURCE is defined to 600. Unless _OPEN_THREADS is defined, _XOPEN_SOURCE 600 will make available the same interfaces and behaviors as _UNIX03_THREADS.

_UNIX03_THREADS and _OPEN_THREADS are mutually exclusive.

_UNIX03_WITHDRAWN

Defining this feature test macro exposes any language elements, previously in the Legacy Feature Group or marked obsolescent, that have been removed from Single Unix Specification, Version 3. These elements would not otherwise be visible in the namespace exposed by compiling with _XOPEN_SOURCE 600 or POSIX_C_SOURCE 200112L.

The following withdrawn symbols are exposed when _UNIX03_WITHDRAWN is defined:

Functions	**Constants**
brk()	CLOCK_TICKS
chroot()	IUCLC
cuserid()	L_cuserid
gamma()	NOSTR
getdtablesize()	OLCUC
getpagesize()	PASS_MAX
getpass()	_SC_2_ C_VERSION
getw()	_SC_PASS_MAX
putw()	_SC_XOPEN_XCU_VERSION
regcmp()	TMP_MAX
regex()	XCASE
sbrk()	YESSTR
sigstack()	
ttyslot()	
valloc()	**External Variable**
wait3()	__loc1

_VARARG_EXT_

This feature test macro allows users of the va_arg, va_end, and va_start macros to define the va_list type differently.

_XOPEN_SOURCE

This feature test macro defines the functionality defined in the XPG4 standard dated July 1992.

When defined to 500, this feature test macro makes available certain key functions that are associated with Single UNIX Specification, Version 2.

When defined to 600, this feature test macro exposes the complete implementation of the Single UNIX Specification, Version 3, including the namespace defined by _POSIX_C_SOURCE 200112L as well as namespaces associated with the X/Open System Interface (XSI) extension and these options and option groups:

- File synchronization
- Memory mapped files
- Memory protection
- Realtime signals extension
- Thread stack address attribute
- Thread stack size attribute
- Thread process-shared synchronization
- Thread-safe functions
- Threads
- Encryption option group
- Legacy option group
- XSI streams option group

The use of _XOPEN_SOURCE 600 exposes namespaces covered by several other feature test macros, and as such, makes those macros redundant. The following need not be defined when _XOPEN_SOURCE 600 is defined:

_ISOC99_SOURCE
_UNIX03_THREADS
_UNIX03_SOURCE
_XOPEN_SOURCE_EXTENDED
_POSIX_C_SOURCE
_OPEN_SYS_MUTEX_EXT
_POSIX_SOURCE

If _OPEN_THREADS is defined with _XOPEN_SOURCE 600, _OPEN_THREADS takes precedence and overrides Single UNIX Specification, Version 3 threads behavior. Whenever _OPEN_THREADS is in effect, the _OPEN_SYS_MUTEX_EXT extensions are also dropped, unless the application explicitly defines this macro.

The _XOPEN_SOURCE 600 definition is available beginning with z/OS V1R9. Targeting earlier releases will result in an error during compile-time.

Full support of Single UNIX Specification, Version 3 requires use of a C99 compliant compiler. Most of the namespace is available to older compilers, but some elements of Version 3 (such as . <complex.h> or <tgmath.h>) will not be visible.

_XOPEN_SOURCE_EXTENDED

When defined to 1, this defines the functionality defined in the XPG4 standard plus the set of “Common APIs for UNIX-based Operating Systems”, April, 1994, draft.

aio.h

The aio.h header file contains definitions for asynchronous I/O operations. It declares these functions:

aio_read() aio_write() aio_cancel()
aio_suspend() aio_error() aio_return()

Note: There are several sockets oriented extensions to asynchronous I/O available with the BPX1AIO callable service, such as asynchronous accept(), asynchronous accept_and_recv(), asynchronous forms of all five pairs of read and write type operations, and receiving I/O completion notifications via an ECB, exit program, or through a message queue. The <aio.h> header contains all the structure fields, constants, and prototypes necessary to use BPX1AIO from a C program. These extensions are exposed when the _AIO_OS390 feature test macro is defined. The BPX1AIO stub resides in SYS1.CSSLIB and must be bound with your program. For a more detailed description of asynchronous I/O services, see BPX1AIO in *z/OS UNIX System Services Programming: Assembler Callable Services Reference*.

arpa/inet.h

The arpa/inet.h header file contains definitions for internet operations.

arpa/nameser.h

The arpa/nameser.h header file contains the definitions used to support the construction of queries and the inspection of answers received from a Domain Name Server available in a network. It also contains the macros GETSHORT(), PUTSHORT(), GETLONG(), and PUTLONG() that are used to construct or inspect DNS requests.

assert.h

The assert.h header file defines the assert() macro that allows you to insert diagnostics into your code. You must include assert.h when you use assert().

cassert

The cassert header file contains definitions for C++ for enforcing assertions when functions execute. Include the standard header into a C++ program to effectively include the standard header <assert.h> within the std namespace.

```
namespace std {
#include <assert.h>
    };
```

_Ccsid.h

The _Ccsid.h header file declares functions, symbols and data types used in CCSID to codeset name conversion.

cctype

The cctype header file contains definitions for C++ for classifying characters. Include the standard header into a C++ program to effectively include the standard header <ctype.h> within the std namespace.

```
namespace std {
#include <ctype.h>
    };
```

ceeedcct.h

The ceeedcct.h header file contains C declarations of the Language Environment condition tokens.

cerrno

The cerrno header file contains definitions for C++ for testing error codes reported by library functions. Include the standard header into a C++ program to effectively include the standard header <errno.h> within the std namespace.

```
namespace std {
#include <errno.h>
    };
```

cfloat

The cfloat header file contains definitions for C++ for testing floating-point type properties. Include the standard header into a C++ program to effectively include the standard header <float.h> within the std namespace.

```
namespace std {
#include <float.h>
    };
```

cics.h

The cics.h header file declares the iscics() function, which verifies whether `cics` is running.

ciso646

The ciso646 header file contains definitions for C++ for programming in ISO646 variant character sets. Include the standard header into a C++ program to effectively include the standard header <iso646.h> within the std namespace.

```
namespace std {
#include <iso646.h>
    };
```

climits

The climits header file contains definitions for C++ for testing integer type properties. Include the standard header into a C++ program to effectively include the standard header <limits.h> within the std namespace.

```
namespace std {
#include <limits.h>
    };
```

clocale

The clocale header file contains definitions for C++ for adapting to different cultural conventions. Include the standard header into a C++ program to effectively include the standard header <locale.h> within the std namespace.

```
namespace std {
#include <locale.h>
    };
```

cmath

The cmath header file contains definitions for C++ for computing common mathematical functions. Include the standard header into a C++ program to effectively include the standard header <math.h> within the std namespace.

```
namespace std {
#include <math.h>
    };
```

collate.h

The collate.h header includes declarations of functions that allow retrieval of information regarding the current locale's collating properties. It declares these functions:

cclass()	collequiv()	collorder()	collrange()	colltostr()
getmccoll()	getwmccoll()	ismccollel()	maxcoll()	strtocoll()

For more information about the effect of locale, see setlocale(), locale.h, or look up the individual functions in this topic. For still more information, see "Internationalization: Locales and Character Sets" in *z/OS XL C/C++ Programming Guide*.

complex.h

The complex.h header file contains function declarations for all the complex math functions listed below.

cabs()	cabsf()	cabsl()	cacos()	cacosf()
cacosh()	cacoshf()	cacoshl()	cacosl()	carg()
cargf()	cargl()	casin()	casinf()	casinh()
casinhf()	casinhl()	casinl()	catan()	catanf()
catanh()	catanhf()	catanhl()	catanl()	ccos()
ccosf()	ccosh()	ccoshf()	ccoshl()	ccosl()
cexp()	cexpf()	cexpl()	cimag()	cimagf()
cimagl()	clog()	clogf()	clogl()	conj()
conjf()	conjl()	cpow()	cpowf()	cpowl()
cproj()	cprojf()	cprojl()	creal()	crealf()
creall()	csin()	csinf()	csinh()	csinhf()
csinhl()	csinl()	csqrt()	csqrtf()	csqrtl()
ctan()	ctanf()	ctanh()	ctanhf()	ctanhl()
ctanl()				

The complex.h header file defines the following macros:

complex
expands to _Complex, where _Complex is a type specifier.

_Complex_I
expands to const float _Complex with the value of the imaginary unit

I expands to _Complex_I

Compile requirement: Use of this header requires a compiler that is designed to support C99.

cpio.h

The cpio.h header file contains CPIO archive values.

csetjmp

The csetjmp header file contains definitions for C++ for executing nonlocal goto statements. Include the standard header into a C++ program to effectively include the standard header <setjmp.h> within the std namespace.

```
namespace std {
#include <setjmp.h>
    };
```

csignal

The csignal header file contains definitions for C++ for controlling various exceptional conditions. Include the standard header into a C++ program to effectively include the standard header <signal.h> within the std namespace.

```
namespace std {
#include <signal.h>
    };
```

csp.h

Restriction: This header file is not supported in AMODE 64.

The csp.h header file declares the __csplist macro, which obtains the CSP parameter list.

These macros are *not* supported under z/OS UNIX and they are not supported for C++ applications.

cstdarg

The cstdarg header file contains definitions for C++ for accessing a varying number of arguments. Include the standard header into a C++ program to effectively include the standard header <stdarg.h> within the std namespace.

```
namespace std {
#include <stdarg.h>
    };
```

cstddef

The cstddef header file contains definitions for C++ for defining several useful types and macros. Include the standard header into a C++ program to effectively include the standard header <stddef.h> within the std namespace.

```
namespace std {
#include <stddef.h>
    };
```

cstdio

The cstdio header file contains definitions for C++ for performing input and output. Include the standard header into a C++ program to effectively include the standard header <stdio.h> within the std namespace.

```
namespace std {
#include <stdio.h>
    };
```

cstdlib

The cstdlib header file contains definitions for C++ for performing a variety of operations. Include the standard header into a C++ program to effectively include the standard header <stdlib.h> within the std namespace.

```
namespace std {
#include <stdlib.h>
    };
```

cstring

The cstring header file contains definitions for C++ for manipulating several kinds of strings. Include the standard header into a C++ program to effectively include the standard header <string.h> within the std namespace.

```
namespace std {
#include <string.h>
    };
```

ctest.h

The ctest.h header file contains declarations for the functions that involve debugging and diagnostics. The diagnostic functions are:

cdump() csnap() ctest() ctrace()

ctime

The ctime header file contains definitions for C++ for converting between various time and date formats. Include the standard header into a C++ program to effectively include the standard header <time.h> within the std namespace.

```
namespace std {
#include <time.h>
    };
```

ctype.h

The ctype.h header file declares functions used in character classification. The functions declared are:

isalnum() isalpha() isblank() iscntrl() isdigit()
isgraph() islower() isprint() ispunct() isspace()
isupper() isxdigit() tolower() toupper()

_XOPEN_SOURCE

isascii() toascii() _tolower() _toupper()

For more information about the effect of locale, see setlocale(), locale.h, or look up the individual functions in this topic. For still more information, see “Internationalization: Locales and Character Sets” in *z/OS XL C/C++ Programming Guide*.

cwchar

The cwchar header file contains definitions for C++ for manipulating wide streams and several kinds of strings. Include the standard header into a C++ program to effectively include the standard header <wchar.h> within the std namespace.

```
namespace std {
#include <wchar.h>
    };
```

cwctype

The cwctype header file contains definitions for C++ for classifying wide characters. Include the standard header into a C++ program to effectively include the standard header <wctype.h> within the std namespace.

```
namespace std {
#include <wctype.h>
    };
```

decimal.h

The decimal.h header file is not supported under z/OS C++ applications.

The decimal.h header file contains declarations for those built-in functions that perform fixed-point decimal operations. The functions declared are:

decabs()   decchk()   decfix()

The header file also contains definitions of constants that specify the ranges of the `decimal` data types.

dirent.h

The dirent.h header file contains constants, prototypes, and typedef definitions for POSIX directory access functions. It declares the following functions.

_OPEN_SYS_DIR_EXT

__opendir2()   __readdir2()

_POSIX_SOURCE

closedir()   opendir()   readdir()   rewinddir()

_XOPEN_SOURCE

seekdir()   telldir()

_XOPEN_SOURCE 500

readdir_r()

This header file can be used by C++ POSIX(OFF) functions.

dlfcn.h

The dlfcn.h header file declares the following macros for use in the construction of a dlopen() mode argument::

RTLD_LAZY
Relocations are performed at an implementation-defined time.

RTLD_NOW
Relocations are performed when the object is loaded.

RTLD_GLOBAL
All symbols are available for relocation processing of other modules.

RTLD_LOCAL
All symbols are not made available for relocation processing by other modules.

dlclose() dlerror() dlsym() dlopen()

dll.h

The dll.h header file declares the following functions:

dllload() dllqueryvar() dllqueryfn() dllfree()

Use this header file when using these functions to import functions and variables from a DLL.

dynit.h

The dynit.h header file contains information for dynamic allocation routines. Specifically, it contains declarations of the dynalloc() and dynfree() functions, the definition of the dyninit() macro, declarations of related structures, and definitions of related constants.

env.h

The env.h header file is used to declare the setenv() and clearenv() functions, which are used in POSIX programs to set and clear environment variables. The env.h header file requires the `_POSIX1_SOURCE 2` feature test macro.

errno.h

The errno.h header file defines the symbolic constants that are returned in the external variable errno.

Table 3. Definitions in errno.h

EACCES	Permission denied
EADDRINUSE	Address in use

Table 3. Definitions in errno.h (continued)

EADDRNOTAVAIL	Address not available
EADV	Advertise error
EAFNOSUPPORT	Address family not supported
EAGAIN	Resource temporarily unavailable
EALREADY	Connection already in progress
EBADF	Bad file descriptor
EBADMSG	Bad message
EBUSY	Resource busy
ECANCELED	Operation canceled
ECHILD	No child processes
ECICS	Function not supported under CICS®
ECOMM	Communication error on send
ECONNABORTED	Connection aborted
ECONNREFUSED	Connection refused
ECONNRESET	Connection reset
EDEADLK	Resource deadlock avoided
EDESTADDRREQ	Destination address required
EDOM	Domain error
EDOTDOT	Cross mount point (not an error)
EDQUOT	Reserved
EEXIST	File exists
EFAULT	Bad address
EFBIG	File too large
EHOSTDOWN	Host is down
EHOSTUNREACH	Destination host can not be reached
EIBMBADCALL	A bad socket-call constant in IUCV header
EIBMBADPARM	Other IUCV header error
EIBMCANCELLED	Request canceled
EIBMCONFLICT	Conflicting call outstanding on socket
EIBMIUCVERR	Request failed due to IUCV error
EIBMSOCKINUSE	Assigned socket number already in use
EIBMSOCKOUTOFRANGE	Assigned socket number out of range
EIDRM	Identifier removed
EILSEQ	Illegal byte sequence
EINPROGRESS	Connection in progress
EINTR	Interrupted function call
EINTRNODATA	Function call interrupted before any data received
EINVAL	Invalid argument
EIO	Input/output error
EISCONN	Socket is already connected
EISDIR	Is a directory
ELEMSGERR	Message file was not found in the hierarchical file system
ELEMULTITHREAD	Function not allowed in a multithreaded environment
ELEMULTITHREADFORK	Function not allowed in child of fork() in multithreaded environment
ELENOFORK	Language Environment member language cannot tolerate a fork()
ELOOP	A loop exists in symbolic links encountered during resolution of the path argument
EMFILE	Too many open files
EMLINK	Too many links
EMSGSIZE	Message too long
EMULTIHOP	Multihop is not allowed

Table 3. Definitions in errno.h (continued)

EMVSBADCHAR	Bad character in environment variable name
EMVSCATLG	Catalog obtain error
EMVSCPLERROR	A CPL service failed
EMVSCVAF	Catalog Volume Access Facility error
EMVSDYNALC	Dynamic allocation error
EMVSERR	An MVS internal error
EMVSEXPIRE	Password has expired
EMVSINITIAL	Process initialization err
EMVSNORTL	Access to the z/OS UNIX version of the C RTL is denied
EMVSNOTXP	z/OS UNIX are not active
EMVSPARM	Bad parameters were passed to the service
EMVSPASSWORD	Password is invalid
EMVSPATHOPTS	Access mode argument conflicts with PATHOPTS parameter
EMVSPFSFILE	PDSE/X encountered a permanent file error
EMVSPFSPERM	PDSE/X encountered a system error
EMVSSAF2ERR	SAF/RACF error
EMVSSAFEXTRERR	SAF/RACF extract error
EMVSTODNOTSET	System TOD clock not set
ENAMETOOLONG	File name too long
ENETDOWN	The local interface to use or reach the destination
ENETRESET	Network dropped connection on reset
ENETUNREACH	Network unreachable
ENFILE	Too many open files in system
ENOBUFS	No buffer space available
ENODATA	No message available
ENODEV	No such device
ENOENT	No such file or directory
ENOEXEC	Exec format error
ENOLINK	The link has been severed
ENOLCK	No locks available
ENOMEM	Not enough space
ENOMSG	No message of desired type
ENONET	Machine is not on the network
ENOPROTOOPT	Protocol not available
ENOREUSE	The socket cannot be reused
ENOSPC	No space left on device
ENOSR	No stream resource
ENOSYS	Function not implemented
ENOSTR	Not a stream
ENOTBLK	Block device required
ENOTCONN	The socket is not connected.
ENOTDIR	Not a directory
ENOTEMPTY	Directory not empty
ENOTSOCK	Descriptor does not refer to a socket
ENOTSUP	Not supported.
ENOTTY	Inappropriate I/O control operation
ENXIO	No such device or address
EOFFLOADboxDOWN	Offload box down
EOFFLOADboxERROR	Offload box error
EOFFLOADboxRESTART	Offload box restarted
EOPNOTSUPP	Operation not supported on socket
EOVERFLOW	Value too large to be stored in data type
EPERM	Operation not permitted

Table 3. Definitions in errno.h (continued)

EPFNOSUPPORT	Protocol family not supported
EPIPE	Broken pipe
EPROCLIM	Too many processes
EPROTO	Protocol error
EPROTONOSUPPORT	Protocol not supported
EPROTOTYPE	The socket type is not supported by the protocol
ERANGE	Range error or pole error
EREMCHG	Remote address changed
EREMOTE	Too many levels of remote in path
EROFS	Read-only file system
ERREMOTE	Object is remote
ESHUTDOWN	Cannot send after socket shutdown
ESOCKTNOSUPPORT	Socket type not supported
ESPIPE	Invalid seek
ESRCH	No such process
ESRMNT	srmount error
ESTALE	The file handle is stale
ETIME	Stream ioctl() timeout
ETIMEDOUT	Socket not connected
ETOOMANYREFS	Too many references: cannot splice
ETXTBSY	Text file busy
EUSERS	Too many users
EWOULDBLOCK	Problem on nonblocking socket
EXDEV	A link to a file on another file system was attempted
E2BIG	Argument list too long

The errno.h header file also defines errno, which is a modifiable `lvalue` having type `int`. If you intend to test the value of errno after library function calls, first set it to 0, because the library functions do not reset the value to 0.

`strerror()` or `perror()` functions can be used to print the description of the message associated with a particular errno.

To test for the explicit error, use the macro names defined in errno.h, rather than specific values of these macros. Doing so will ensure future compatibility and portability.

errno.h also declares the __errno2() prototype.

exception

The exception header file defines several types and functions related to the handling of exceptions.

```
namespace std {
    class exception;
    class bad_exception;
    typedef void (*terminate_handler)();
    typedef void (*unexpected_handler)();
    terminate_handler
        set_terminate(terminate_handler ph) throw();
    unexpected_handler
        set_unexpected(unexpected_handler ph) throw();
    void terminate();
    void unexpected();
    bool uncaught_exception();
    };
```

bad_exception

```
class bad_exception : public exception {
    };
```

The class describes an exception that can be thrown from an unexpected handler. The value returned by what() is an implementation-defined C string. None of the member functions throw any exceptions.

exception

```
class exception {
public:
    exception() throw();
    exception(const exception& rhs) throw();
    exception& operator=(const exception& rhs) throw();
    virtual ~exception() throw();
    virtual const char *what() const throw();
    };
```

The class serves as the base class for all exceptions thrown by certain expressions and by the Standard C++ library. The C string value returned by what() is left unspecified by the default constructor, but may be defined by the constructors for certain derived classes as an implementation-defined C string. None of the member functions throw any exceptions.

terminate_handler

```
typedef void (*terminate_handler)();
```

The type describes a pointer to a function suitable for use as a terminate handler.

unexpected_handler

```
typedef void (*unexpected_handler)();
```

The type describes a pointer to a function suitable for use as an unexpected handler.

fcntl.h

The fcntl.h header file declares the following POSIX functions for creating, opening, rewriting, and manipulating files.

_POSIX_SOURCE

creat() fcntl() open()

features.h

The features.h header file contains definitions for feature test macros. For information on feature test macros, see “Feature test macros” on page 3.

fenv.h

The fenv.h header contains the following data types, macros and functions. This header is supported under IEEE Binary Floating-Point only and is required to define the feature test macro _ISOC99_SOURCE or requires the compiler that is designed to support C99 to expose the functionality. The decimal floating-point macros, prefixed by FE_DEC_ and _FE_DEC_, are used by the fe_dec_getround and fe_dec_setround functions to get and set the rounding mode of decimal floating-point operations. Decimal floating-point functionality additionally requires the __STDC_WANT_DEC_FP__ feature test macro to be defined.

Data types

fenv_t represents the entire floating-point environment.

fexcept_t
represents the floating-point status flags collectively.

Macros

FE_DIVBYZERO
defines the divide by zero exception

_FE_DEC_AWAYFROMZERO
rounds away from zero

FE_DEC_DOWNWARD
rounds towards minus infinity

_FE_DEC_PREPAREFORSHORTER
rounds to prepare for shorter precision

FE_DEC_TONEAREST
rounds to nearest

FE_DEC_TONEARESTFROMZERO
rounds to nearest, ties away from zero

_FE_DEC_TONEARESTTOWARDZERO
rounds to nearest, ties toward zero

FE_DEC_TOWARDZERO
rounds toward zero

FE_DEC_UPWARD
rounds toward plus infinity

FE_INEXACT
defines the inexact exception

FE_INVALID
defines the invalid exception

FE_OVERFLOW
defines the overflow exception

FE_UNDERFLOW
defines the underflow exception

FE_ALL_EXCEPT
defines the bitwise OR of all the exception macros.

FE_DOWNWARD
rounds towards minus infinity

FE_TONEAREST
rounds to nearest

FE_TOWARDZERO
rounds toward zero

FE_UPWARD
rounds towards plus infinity

FE_DFL_ENV
defines the floating-point environment as it was available at program start up

Functions

feclearexcept()
feraiseexcept()
feupdateenv()
fegetenv()
fesetenv()
fe_dec_getround()
fegetexceptflag()
fesetexceptflag()
fegetround()
fesetround()
fe_dec_setround()
feholdexcept()
fetestexcept()

float.h

The float.h header file contains definitions of constants listed in ANSI 2.2.4.2.2, that describe the characteristics of the internal representations of the three floating-point data types, float, double, and long double. The definitions are:

`Constant`
Description

`FLT_ROUNDS`
The rounding mode for floating-point addition.

`FLT_RADIX`
The radix for z/OS XL C applications. The FLT_RADIX value depends on the compile option FLOAT. FLOAT(HEX) gives value 16. FLOAT(IEEE) gives value 2.

`FLT_MANT_DIG`
`DBL_MANT_DIG`
`LDBL_MANT_DIG`
The number of hexadecimal digits stored to represent the significand of a fraction.

`FLT_DIG`
`DBL_DIG`
`LDBL_DIG`
The number of decimal digits, *q*, such that any floating-point number with *q* decimal digits can be rounded into a floating-point number with *p* radix FLT_RADIX digits and back again, without any change to the *q* decimal digits.

`FLT_MIN_EXP`
`DBL_MIN_EXP`
`LDBL_MIN_EXP`
The minimum negative integer such that FLT_RADIX raised to that power minus 1 is a normalized floating-point number.

`FLT_MIN_10_EXP DBL_MIN_10_EXP`
`LDBL_MIN_10_EXP`
The minimum negative integer such that 10 raised to that power is in the range of normalized floating-point numbers.

`FLT_MAX_EXP`

DBL_MAX_EXP
LDBL_MAX_EXP
The maximum integer such that FLT_RADIX raised to that power minus 1 is a representable finite floating-point number.

FLT_MAX_10_EXP
DBL_MAX_10_EXP
LDBL_MAX_10_EXP
The maximum integer such that 10 raised to that power is in the range of representable finite floating-point numbers.

FLT_MAX
DBL_MAX
LDBL_MAX
The maximum representable finite floating-point number.

FLT_EPSILON
DBL_EPSILON
LDBL_EPSILON
The difference between 1.0 and the least value greater than 1.0 that is representable in the given floating-point type.

FLT_MIN
DBL_MIN
LDBL_MIN
The minimum normalized positive floating-point number.

DECIMAL_DIG
The minimum number of decimal digits needed to represent all the significant digits for type long double.

FLT_EVAL_METHOD
Describes the evaluation mode for floating point operations. This value is 1, which evaluates all operations and constants of types float and double to type double and that of long double to type long double.

DEC_EVAL_METHOD
The decimal floating-point evaluation format.

FLT_MAXDIG10
DBL_MAXDIG10
LDBL_MAXDIG10
The number of base 10 digits required to ensure that values which differ by only one smallest unit in the last place (ulp) are always differentiated.

FLT_SUBNORM
DBL_SUBNORM
LDBL_SUBNORM
Characterize the presence or absence of subnormal numbers:

- -1: indeterminable
- 0: absent (type does not support subnormal numbers)
- 1: present (type does support subnormal numbers)

FLT_DECIMAL_DIG
DBL_DECIMAL_DIG
LDBL_DECIMAL_DIG
The number of decimal digits, n, such that any floating-point number with p radix b digits can be rounded to a floating-point number with n decimal digits and back again without change to the value.

FLT_TRUE_MIN

DBL_TRUE_MIN
LDBL_TRUE_MIN
The minimum positive floating-point number.

The float.h header file also contains constants that describe the characteristics of the internal representations of the three decimal floating-point data types, _Decimal32, _Decimal64, and _Decimal128. The prefixes DEC32_, DEC64_, and DEC128_ are used to denote the types _Decimal32, _Decimal64, and _Decimal128, respectively.

Constant
Description

DEC32_MANT_DIG
DEC64_MANT_DIG
DEC128_MANT_DIG
The number of digits in the coefficient.

DEC32_MIN_EXP
DEC64_MIN_EXP
DEC128_MIN_EXP
The minimum exponent.

DEC32_MAX_EXP
DEC64_MAX_EXP
DEC128_MAX_EXP
The maximum exponent.

DEC32_MAX
DEC64_MAX
DEC128_MAX
The maximum representable finite decimal floating number.

DEC32_EPSILON
DEC64_EPSILON
DEC128_EPSILON
The difference between 1 and the least value greater than 1 that is representable in the given floating point type.

DEC32_MIN
DEC64_MIN
DEC128_MIN
The minimum normalized positive decimal floating number.

DEC32_SUBNORMAL_MIN
DEC64_SUBNORMAL_MIN
DEC128_SUBNORMAL_MIN
The minimum subnormal (denormalized) positive Decimal Floating Point number.

DEC_EVAL_METHOD
The decimal floating-point evaluation format.

fmtmsg.h

The fmtmsg.h header file contains message display structures.

fnmatch.h

The fnmatch.h header file contains filename matching types.

fpxcp.h

The fpxcp.h header file declares floating-point exception interfaces.

__ftp.h

The __ftp.h header file contains definitions for FTP resolver functions.

ftw.h

The ftw.h header file contains file tree traversal constants.

glob.h

The glob.h header file contains pathname pattern matching types.

grp.h

The grp.h header file declares functions used to access group databases.

_POSIX_SOURCE

getgrgid()	__getgrgid1()	getgrnam()	__getgrnam1()

_OPEN_SYS

initgroups()	setgroups()

_XOPEN_SOURCE_EXTENDED 1

setgrent()	endgrent()	getgrent()

_XOPEN_SOURCE 500

getgrgid_r()	getgrnam_r()

iconv.h

The iconv.h header file declares the `iconv_open()`, `iconv()`, and `iconv_close()` functions that deal with code conversion.

_Ieee754.h

The _Ieee754.h header file declares IEEE 754 interfaces.

ims.h

Restriction: This header file is not supported in AMODE 64.

The ims.h header file declares the ctdli() function that invokes IMS™ facilities. The function is *not* supported from a z/OS UNIX program running POSIX(ON).

inttypes.h

The following macros are defined in inttypes.h. Each expands to a character string literal containing a conversion specifier which can be modified by a length modifier that can be used in the *format* argument of a formatted input/output function when converting the corresponding integer type. These macros have the general form of PRI (character string literals for the fprintf() and fwprintf() family of functions) or SCN (character string literals for the fscanf() and fwscanf() family of functions), followed by the conversion specifier, followed by a name corresponding to a similar type name in <inttypes.h>. In these names, the suffix number represents the width of the type. For example, *PRIdFAST32* can be used in a format string to print the value of an integer of type **int_fast32_t**.

This header defines the type **imaxdiv_t**.

Note: Requires long long to be available.
imaxdiv_t is a structure type that is the type of the value returned by the imaxdiv() function. It is functionally equivalent to lldiv_t.

Compile requirement:

In the following list all macros with the suffix MAX or 64 require long long to be available.

PRId8	PRId16	PRId32	PRId64
PRIdLEAST8	PRIdLEAST16	PRIdLEAST32	PRIdLEAST64
PRIdFAST8	PRIdFAST16	PRIdFAST32	PRIdFAST64
PRIdMAX			
PRIdPTR			
PRIi8	PRIi16	PRIi32	PRIi64
PRIiLEAST8	PRIiLEAST16	PRIiLEAST32	PRIiLEAST64
PRIiFAST8	PRIiFAST16	PRIiFAST32	PRIiFAST64
PRIiMAX			
PRIiPTR			

Example

```
#define _ISOC99_SOURCE
#include <inttypes.h>
#include <stdio.h>
int main(void)
{

int8_t i =  40;
printf("Demonstrating the use of the following macros:\n");
printf("Using PRId8, the printed value of 40 "
"is  %" PRId8"\n", i);
printf("Using PRIiFAST8, the printed value of 40 "
"is  %" PRIiFAST8"\n", i);
printf("Using PRIoLEAST8, the printed value of 40 "
"is  %" PRIoLEAST8 "\n", i);
return 0;
}


Output:

Demonstrating the use of the following macros:
```

```
Using PRId8, the printed value of 40 is  40
Using PRIiFAST8, the printed value of 40 is  40
Using PRIoLEAST8, the printed value of 40 is  50
```

Compile requirement:

In the following list all macros with the suffix MAX or 64 require long long to be available.

Macros for fprintf family for unsigned integers:

PRIo8	PRIo16	PRIo32	PRIo64
PRIoLEAST8	PRIoLEAST16	PRIoLEAST32	PRIoLEAST64
PRIoFAST8	PRIoFAST16	PRIoFAST32	PRIoFAST64
PRIoMAX			
PRIoPTR			
PRIu8	PRIu16	PRIu32	PRIu64
PRIuLEAST8	PRIuLEAST16	PRIuLEAST32	PRIuLEAST64
PRIuFAST8	PRIuFAST16	PRIuFAST32	PRIuFAST64
PRIuMAX			
PRIuPTR			
PRIx8	PRIx16	PRIx32	PRIx64
PRIxLEAST8	PRIxLEAST16	PRIxLEAST32	PRIxLEAST64
PRIxFAST8	PRIxFAST16	PRIxFAST32	PRIxFAST64
PRIxMAX			
PRIxPTR			
PRIX8	PRIX16	PRIX32	PRIX64
PRIXLEAST8	PRIXLEAST16	PRIXLEAST32	PRIXLEAST64
PRIXFAST8	PRIXFAST16	PRIXFAST32	PRIXFAST64
PRIXMAX			
PRIXPTR			

Example

```
#define _ISOC99_SOURCE
#include <inttypes.h>
#include <stdio.h>

int main(void)
{
uint32_t i =  24000;
printf("Demonstrating the use of the following macros:\n");
printf("Using PRIuPTR, the address of the variable "
"is  %" PRIuPTR "\n", i);
printf("Using PRIXFAST32, the printed value of 24000 "
"is  %" PRIXFAST32"\n", i);
printf("Using PRIxLEAST32, the printed value of 24000 "
"is  %" PRIxLEAST32 "\n", i);
return 0;
}
```

```
Output:

Demonstrating the use of the following macros:
Using PRIuPTR, the address of the variable is  538874544
Using PRIXFAST32, the printed value of 24000 is  5DC0
Using PRIxLEAST32, the printed value of 24000 is  5dc0
```

Compile requirement:

In the following list all macros with the suffix MAX or 64 require long long to be available.

Macros for fscanf family for signed integers:

SCNd8	SCNd16	SCNd32	SCNd64
SCNdLEAST8	SCNdLEAST16	SCNdLEAST32	SCNdLEAST64
SCNdFAST8	SCNdFAST16	SCNdFAST32	SCNdFAST64
SCNdMAX			
SCNdPTR			
SCNi8	SCNi16	SCNi32	SCNi64
SCNiLEAST8	SCNiLEAST16	SCNiLEAST32	SCNiLEAST64
SCNiFAST8	SCNiFAST16	SCNiFAST32	SCNiFAST64
SCNiMAX			
SCNiPTR			

Example

```
#define _ISOC99_SOURCE
#include <inttypes.h>
#include <stdio.h>

int main(void)
{
    int32_t i;
    printf("Enter decimal value ");
    scanf("%" SCNdFAST32, i);
    printf("Print result: %" PRIdFAST32 "\n", i);
   return 0;
}

Output:

Enter decimal value 23
Print result: 23
```

Compile requirement:

In the following list all macros with the suffix MAX or 64 require long long to be available.

Macros for fscanf family for signed integers:

SCNo8	SCNo16	SCNo32	SCNo64
SCNoLEAST8	SCNoLEAST16	SCNoLEAST32	SCNoLEAST64
SCNoFAST8	SCNoFAST16	SCNoFAST32	SCNoFAST64
SCNoMAX			
SCNoPTR			
SCNu8	SCNu16	SCNu32	SCNu64
SCNuLEAST8	SCNuLEAST16	SCNuLEAST32	SCNuLEAST64
SCNuFAST8	SCNuFAST16	SCNuFAST32	SCNuFAST64
SCNuMAX			
SCNuPTR			
SCNx8	SCNx16	SCNx32	SCNx64
SCNxLEAST8	SCNxLEAST16	SCNxLEAST32	SCNxLEAST64
SCNxFAST8	SCNxFAST16	SCNxFAST32	SCNxFAST64
SCNxMAX			
SCNxPTR			

Example

```
#define _ISOC99_SOURCE
#include <inttypes.h>
#include <stdio.h>

int main(void)
{
    intmax_t i;
    printf("Enter hex value ");
    scanf("%" SCNxMAX, i);
    printf("Print result: %020" PRIxMAX "\n", i);
    return 0;
}

Output :

Enter hex value 0x32
Print result: 00000000000000000032
```

iso646.h

The header file iso646.h allows the user to use the following macros in place of the associated operator.

Macros	**Operators**
and	&&
and_eq	&=
bitand	&
bitor	|
compl	~
not	!
not_eq	!=
or	||
or_eq	|=
xor	^
xor_eq	^=

langinfo.h

The langinfo.h header file contains the declaration for the `nl_langinfo()` function. The header file also defines the macros that, in turn, define constants used to identify the information queried by `nl_langinfo()` in the current locale. The following macros are defined:

Table 4. Item Values defined in langinfo.h

Item Name	Description
ABDAY_1	Abbreviated first day of the week
ABDAY_2	Abbreviated second day of the week
ABDAY_3	Abbreviated third day of the week
ABDAY_4	Abbreviated fourth day of the week
ABDAY_5	Abbreviated fifth day of the week
ABDAY_6	Abbreviated sixth day of the week

Table 4. Item Values defined in langinfo.h (continued)

Item Name	Description
ABDAY_7	Abbreviated seventh day of the week
ABMON_1	Abbreviated first month
ABMON_2	Abbreviated second month
ABMON_3	Abbreviated third month
ABMON_4	Abbreviated fourth month
ABMON_5	Abbreviated fifth month
ABMON_6	Abbreviated sixth month
ABMON_7	Abbreviated seventh month
ABMON_8	Abbreviated eighth month
ABMON_9	Abbreviated ninth month
ABMON_10	Abbreviated tenth month
ABMON_11	Abbreviated eleventh month
ABMON_12	Abbreviated twelfth month
ALT_DIGITS	String of semicolon separated alternative symbols for digits
AM_STR	String for morning
CODESET	Current encoded character set of the process
CRNCYSTR	Local currency symbol, preceded by '-' if the symbol should appear before the value, '+' if the symbol should appear after the value, or '.' if the symbol should replace the radix character.
D_FMT	String for formatting date
D_T_FMT	String for formatting date and time
DAY_1	Name of the first day of the week
DAY_2	Name of the second day of the week
DAY_3	Name of the third day of the week
DAY_4	Name of the fourth day of the week
DAY_5	Name of the fifth day of the week
DAY_6	Name of the sixth day of the week
DAY_7	Name of the seventh day of the week
ERA	String of semicolon separated era segments
ERA_D_FMT	String for era date format
ERA_D_T_FMT	String for era date and time format
ERA_T_FMT	String for era time format
MON_1	Name of the first month
MON_2	Name of the second month
MON_3	Name of the third month
MON_4	Name of the fourth month
MON_5	Name of the fifth month
MON_6	Name of the sixth month
MON_7	Name of the seventh month
MON_8	Name of the eighth month
MON_9	Name of the ninth month
MON_10	Name of the tenth month
MON_11	Name of the eleventh month
MON_12	Name of the twelfth month
NOEXPR	Negative response expression
NOSTR	Negative response string
PM_STR	String for afternoon
RADIXCHAR	Radix character
T_FMT	String for formatting time
T_FMT_AMPM	String for formatting time in 12-hour clock format
THOUSEP	Separator for thousands
YESEXPR	Affirmative response expression
YESSTR	affirmative response string

Note:

The YESSTR and NOSTR constants are kept for historical reasons. They were part of the Legacy Feature in Single UNIX Specification, Version 2. They have been withdrawn and are not supported as part of Single UNIX Specification, Version 3.

If it is necessary to continue using these constants in an application written for Single UNIX Specification, Version 3, define the feature test macro _UNIX03_WITHDRAWN before including any standard system headers. The macro exposes all interfaces and symbols removed in Single UNIX Specification, Version 3.

For more information about the effect of locale, see setlocale(), locale.h, or look up the individual functions in this topic. For still more information, see “Internationalization: Locales and Character Sets” in *z/OS XL C/C++ Programming Guide*.

lc_core.h

The lc_core.h header file contains locale-related data structures.

lc_sys.h

The lc_sys.h header file contains definitions used by the localedef utility for building references to locale methods in ASCII locales.

__le_api.h

The __le_api.h header file declares the following AMODE 64 C functions in Language Environment®:

__le_ceegtjs()
__le_msg_add_insert()
__le_msg_write()
__le_cib_get()
__le_msg_get()
__le_set_debug_resume_mch()
__le_condition_token_build()
__le_msg_get_and_write()

Restrictions:

- This header is supported in AMODE 64 only. For AMODE 31, the leawi.h header file should be used.

leawi.h

Restriction: This header file is not supported in AMODE 64. For AMODE 64, the __le_api.h header file should be used.

The leawi.h header file contains internal macros. The header file is required for applications that use Language Environment Application Writer Interfaces (LE AWI).

libgen.h

The libgen.h header file contains definitions for pattern matching functions.

limits.h

The limits.h header file contains symbolic names that represent standard values for limits on resources, such as the maximum value for an object of type `char`.

Table 5. Definitions of resource limits (limits.h)

ATEXIT_MAX	2048
BC_DIM_MAX	32768
BC_SCALE_MAX	32767
BC_STRING_MAX	2048
CHAR_BIT	8
CHAR_MAX	127 (_CHAR_SIGNED)
CHAR_MAX	255
CHAR_MIN	(-128) (_CHAR_SIGNED)
CHAR_MIN	0
COLL_WEIGHTS_MAX	2
__DIR_NAME_MAX	256
EXPR_NEST_MAX	32
INT_MAX	2147483647
INT_MIN	(-2147483647 - 1)
LINE_MAX	2048
LLONG_MAX	(9223372036854775807LL)
LLONG_MIN	(-LLONG_MAX-1)
LONG_MAX	2147483647
LONGLONG_MAX	(9223372036854775807LL)
LONG_MIN	(-2147483647L - 1)
LONGLONG_MIN	(-LONGLONG_MAX - 1)
MB_LEN_MAX	4
NGROUPS_MAX	300
PASS_MAX	255
_POSIX_ARG_MAX	4096
_POSIX_CHILD_MAX	25
_POSIX_DATAKEYS_MAX	32
_POSIX_LINK_MAX	8
_POSIX_MAX_CANON	255
_POSIX_MAX_INPUT	255
_POSIX_NAME_MAX	14
_POSIX_NGROUPS_MAX	8
_POSIX_OPEN_MAX	20
_POSIX_PATH_MAX	255
_POSIX_PIPE_BUF	512
_POSIX_SSIZE_MAX	32767
_POSIX_STREAM_MAX	8
POSIX_SYMLOOP	24
_POSIX_TZNAME_MAX	6
_POSIX2_BC_BASE_MAX	99
_POSIX2_BC_DIM_MAX	2048
_POSIX2_BC_SCALE_MAX	99
_POSIX2_BC_STRING_MAX	1000
_POSIX2_COLL_WEIGHTS_MAX	2
_POSIX2_EXPR_NEST_MAX	32
_POSIX2_LINE_MAX	2048
_POSIX2_RE_DUP_MAX	255
RE_DUP_MAX	255
SCHAR_MAX	127
SCHAR_MIN	(-128)

Table 5. Definitions of resource limits (limits.h) (continued)

SHRT_MAX	32767
SHRT_MIN	(-32768)
SSIZE_MAX	2147483647
UCHAR_MAX	255
UINT_MAX	4294967295
ULONG_MAX	4294967295U
ULONGLONG_MAX	(18446744073709551615ULL)
ULLONG_MAX	(18446744073709551615ULL)
USHRT_MAX	65535
When compiled with SUSV3 thread support (_UNIX03_THREADS or _XOPEN_SOURCE 600), limits.h adds the following constants:	
PTHREAD_STACK_MIN	4096 (1048576 in 64-bit)
_POSIX_THREAD_DESTRUCTOR_ITERATIONS	4
_POSIX_THREAD_KEYS_MAX	128
_POSIX_THREAD_THREADS_MAX	64

localdef.h

The localdef.h header file defines data structures for locale objects which are loaded by setlocale(). The data structures in localdef.h are not a supported programming interface.

locale.h

The locale.h header file contains declarations for the localdtconv() and localeconv() library functions, which retrieve values from the current locale, and for the setlocale() function, used to query or change locale settings for internationalized applications.

The locale.h file declares the `lconv` structure. Table 6 below shows the elements of the `lconv` structure and the defaults for the C locale.

Table 6. Elements of lconv Structure

Element	Purpose of Element	Default
char *decimal_point	Decimal-point character used to format non-monetary quantities.	"."
char *thousands_sep	Character used to separate groups of digits to the left of the decimal-point character in formatted non-monetary quantities.	""
char *grouping	String indicating the size of each group of digits in formatted non-monetary quantities. The value of each character in the string determines the number of digits in a group. A value of `CHAR_MAX` indicates that there are no further groupings. 0 indicates that the previous element is to be used for the remainder of the digits.	""

Table 6. Elements of lconv Structure (continued)

Element	Purpose of Element	Default
char *int_curr_symbol	International currency symbol for the current locale. The first three characters contain the alphabetic international currency symbol. The fourth character (usually a space) is the character used to separate the international currency symbol from the monetary quantity.	""
char *currency_symbol	Local currency symbol of the current locale.	""
char *mon_decimal_point	Decimal-point character used to format monetary quantities.	"."
char *mon_thousands_sep	Separator for digits in formatted monetary quantities.	""
char *mon_grouping	String indicating the size of each group of digits in formatted monetary quantities. The value of each character in the string determines the number of digits in a group. A value of CHAR_MAX indicates that there are no further groupings. 0 indicates that the previous element is to be used for the remainder of the digits.	""
char *positive_sign	String indicating the positive sign used in monetary quantities.	""
char *negative_sign	String indicating the negative sign used in monetary quantities.	""
char int_frac_digits	The number of displayed digits to the right of the decimal place for internationally formatted monetary quantities.	UCHAR_MAX
char frac_digits	Number of digits to the right of the decimal place in monetary quantities.	UCHAR_MAX
char p_cs_precedes	Value indicating the placement of the currency symbol in a nonnegative, formatted monetary quantity. For a list of valid values, see Table 7 on page 42.	UCHAR_MAX
char p_sep_by_space	Value indicating the use of white space in a nonnegative, formatted monetary quantity. For a list of valid values, see Table 7 on page 42.	UCHAR_MAX
char n_cs_precedes	Value indicating the placement of the currency symbol in a negative, formatted monetary quantity. For a list of valid values, see Table 7 on page 42.	UCHAR_MAX
char n_sep_by_space	Value indicating the use of white space in a negative, formatted monetary quantity. For a list of valid values, see Table 7 on page 42.	UCHAR_MAX
char p_sign_posn	Value indicating the position of the positive_sign for a nonnegative formatted monetary quantity. For a list of valid values, see Table 7 on page 42.	UCHAR_MAX

Table 6. Elements of lconv Structure (continued)

Element	Purpose of Element	Default
char n_sign_posn	Value indicating the position of the `negative_sign` for a negative formatted monetary quantity. For a list of valid values, see Table 7.	UCHAR_MAX
char *left_parenthesis	Negative-valued monetary symbol. **Note:** This element is an IBM-specific extension.	""
char *right_parenthesis	Negative-valued monetary symbol. **Note:** This element is an IBM-specific extension.	""
char *debit_sign	Debit_sign character string. **Note:** This element is an IBM-specific extension.	""
char *credit_sign	Credit_sign character string. **Note:** This element is an IBM-specific extension.	""
char int_p_cs_precedes	For international formatting, value indicating the placement of the currency symbol in a nonnegative, monetary quantity. For a list of valid values, see Table 7.	UCHAR_MAX
char int_n_cs_precedes	For international formatting, value indicating the placement of the currency symbol in a negative, monetary quantity. For a list of valid values, see Table 7.	UCHAR_MAX
char int_p_sep_by_space	For international formatting, value indicating the use of white space in a nonnegative, monetary quantity. For a list of valid values, see Table 7.	UCHAR_MAX
char int_n_sep_by_space	For international formatting, value indicating the use of white space in a negative, monetary quantity. For a list of valid values, see Table 7.	UCHAR_MAX
char int_p_sign_posn	For international formatting, value indicating the position of the positive sign for a nonnegative monetary quantity. For a list of valid values, see Table 7.	UCHAR_MAX
char int_n_sign_posn	For international formatting, value indicating the position of the negative sign for a negative monetary quantity. For a list of valid values, see Table 7.	UCHAR_MAX

For a list of valid values, see Table 7.

Table 7. Monetary Formatting Values

Element	Values	
cs_precedes	0	The `currency_symbol` succeeds the value for the formatted monetary quantity;
	1	The `currency_symbol` precedes the value for the formatted monetary quantity

Table 7. Monetary Formatting Values (continued)

Element	Values
sep_by_space	0 No space separates the currency_symbol from the formatted monetary quantity;
	1 If currency_symbol and sign string are adjacent, a space separates them from the value. Otherwise, the currency symbol and value are separated by a space.
	2 If currency_symbol and sign string are adjacent, a space separates them from each other. Otherwise, the sign string and value are separated by a space.
sign_posn	0 Parentheses surround the quantity and currency_symbol;
	1 The sign string precedes the quantity and currency_symbol;
	2 The sign string succeeds the quantity and currency_symbol;
	3 The sign string immediately precedes currency_symbol;
	4 The sign string immediately succeeds currency_symbol;
	5 Substitute debit_sign or credit_sign for negative_sign or positive_sign, respectively. **Note:** This value is an IBM-specific extension.

The locale.h file declares the dtconv structure:

```
struct dtconv {
        char *abbrev_month_names[12];   /* Abbreviated month names      */
        char *month_names[12];          /* full month names             */
        char *abbrev_day_names[7];      /* Abbreviated day names        */
        char *day_names[7];             /* full day names               */
        char *date_time_format;         /* date and time format         */
        char *date_format;              /* date format                  */
        char *time_format;              /* time format                  */
        char *am_string;                /* AM string                    */
        char *pm_string;                /* PM string                    */
        char *time_format_ampm;         /* long date format             */
        char *iso_std8601_2000;         /* ISO 8601:2000 std date format*/
};
```

Note: This structure is an IBM-specific extension.

The locale.h file also contains macro definitions for use with the setlocale() function:

LC_ALL	LC_COLLATE	LC_CTYPE	LC_MONETARY
LC_NUMERIC	LC_TIME	LC_TOD	NULL
LC_MESSAGES	LC_SYNTAX		

The aspects of a program related to national language or to cultural characteristics (such as time zone, currency symbols, and sorting order of characters) can be customized at run time using different locales, to suit users' requirements at those locales. See the internationalization topic in *z/OS XL C/C++ Programming Guide*.

math.h

The math.h header file contains function declarations for all the floating-point math functions:

No feature test macro required.

Notes:

1. nan(), nanf(), and nanl() functions are supported under IEEE only.
2. For the C99 math functions, it is required to define the feature test macro _ISO_C99_SOURCE or requires a compiler that is designed to support C99 to expose the functionality.

absf() absl() acos() acosf() acoshf()
acoshl() acosl() asin() asinf() asinhf()
asinhl() asinl() atan() atan2() atan2f()
atan2l() atanf() atanl() cbrtf() cbrtl()
ceil() ceilf() ceill() copysign() copysignf()
copysignl() cos() cosf() cosh() coshf()
coshl() cosl() exp() expf() expl()
expm1f() expm1l() exp2() exp2f() exp2l()
fabsf() fabsl() floor() floorf() floorl()
fma() fmaf() fmal() fmax() fmaxf()
fmaxl() fmin() fminf() fminl() fmod()
fmodf() fmodl() frexp() frexpf() frexpl()
hypotf() hypotl() ilogbf() ilogbl() ldexp()
ldexpf() ldexpl() lgammaf() lgammal() llrint()
llrintf() llrintl() llround() llroundf() llroundl()
log() logbf() logbl() logf() logl()
log1pf() log1pl() log10() log10f() log10l()
lrint() lrintf() lrintl() lround() lroundf()
lroundl() modf() modff() modfl() nan()
nanf() nanl() nearbyint() nearbyintf() nearbyintl()
nextafterf() nextafterl() nexttoward() nexttowardf() nexttowardl()
pow() powf() powl() remainderf() remainderl()
remquo() remquof() remquol( ) rintf() rintl()
round() roundf() roundl() scalbln() scalblnf()
scalblnl() sin() sinf() sinh() sinhf()
sinhl() sinl() sqrt() sqrtf() sqrtl()
tan() tanf() tanh() tanhf() tanhl()
tanl() tgamma() tgammaf() tgammal()

Special Behavior for C++: For C++ applications, each of the base functions in the following list is also overloaded for float, double, and long double. For example:

- float sqrt(float)
- double sqrt(double)
- long double sqrt(long double)

_XOPEN_SOURCE

erf() erfc() gamma() hypot() isnan()
jn() j0() j1() lgamma() yn()
y0() y1()

_XOPEN_SOURCE_EXTENDED 1

acosh()
asinh()
atanh()
cbrt()
expm1()
ilogb()
logb()
log1p()
nextafter()
remainder()
rint()
scalb()

__STDC_WANT_DEC_FP__

acosd32()
acosd64()
acosd128()
acoshd32()
acoshd64()
acoshd128()
asind32()
asind64()
asind128()
asinhd32()
asinhd64()
asinhd128()
atand32()
atand64()
atand128()
atan2d32()
atan2d64()
atan2d128()
atanhd32()
atanhd64()
atanhd128()
__atanpid32()
__atanpid64()
__atanpid128()
cbrtd32()
cbrtd64()
cbrtd128()
ceild32()
ceild64()
ceild128()
copysignd32()
copysignd64()
copysignd128()
cosd32()
cosd64()
cosd128()
coshd32()
coshd64()
coshd128()
__cospid32()
__cospid64()
__cospid128()
erfd32()
erfd64()
erfd128()
erfcd32()
erfcd64()
erfcd128()
expd32()
expd64()
expd128()
expm1d32()
expm1d64()
expm1d128()
exp2d32()
exp2d64()
exp2d128()
fabsd32()
fabsd64()
fabsd128()
fdimd32()
fdimd64()
fdimd128()
floord32()
floord64()
floord128()
fmad32()
fmad64()
fmad128()
fmaxd32()
fmaxd64()
fmaxd128()
fmind32()
fmind64()
fmind128()
fmodd32()
fmodd64()
fmodd128()
frexpd32()
frexpd64()
frexpd128()
hypotd32()
hypotd64()
hypotd128()
ilogbd32()
ilogbd64()
ilogbd128()
ldexpd32()
ldexpd64()
ldexpd128()
lgammad32()
lgammad64()
lgammad128()
llrintd32()
llrintd64()
llrintd128()
llroundd32()
llroundd64()
llroundd128()
logd32()
logd64()
logd128()
log1pd32()
log1pd64()
log1pd128()
log2d32()
log2d64()
log2d128()
log10d32()
log10d64()
log10d128()
logbd32()
logbd64()
logbd128()
lrintd32()
lrintd64()
lrintd128()
lroundd32()
lroundd64()
lroundd128()
modfd32()
modfd64()
modfd128()
nand32()
nand64()
nand128()
nearbyintd32()
nearbyintd64()
nearbyintd128()
nextafterd32()
nextafterd64()
nextafterd128()
nexttowardd32()
nexttowardd64()
nexttowardd128()
powd32()
powd64()
powd128()
quantexpd32()
quantexpd64()
quantexpd128()
quantized32()
quantized64()
quantized128()
remainderd32()
remainderd64()
remainderd128()
__remquod32()
__remquod64()
__remquod128()
rintd32()
rintd64()
rintd128()
roundd32()
roundd64()
roundd128()
samequantumd32()
samequantumd64()
samequantumd128()
scalblnd32()
scalblnd64()
scalblnd128()
scalbnd32()
scalbnd64()
scalbnd128()
sind32()
sind64()
sind128()
sinhd32()
sinhd64()
sinhd128()
__sinpid32()
__sinpid64()
__sinpid128()
sqrtd32()
sqrtd64()
sqrtd128()
tand32()
tand64()
tand128()
tanhd32()
tanhd64()
tanhd128()
tgammad32()
tgammad64()
tgammad128()
truncd32()
truncd64()
truncd128()

For C++ applications, the following functions are overloaded for _Decimal32, _Decimal64, and _Decimal128:

abs()
acos()
acosh()
asin()
asinh()
atan()
atan2()
atanh()
cbrt()
ceil()
copysign()
cos()
cosh()
erf()
erfc()
exp()
expm1()
exp2()
fabs()
fdim()
floor()
fma()
fmax()
fmin()
fmod()

frexp() hypot() ilogb() ldexp() lgamma()

llrint() llround() log() log1p() log2()

log10() logb() lrint() lround() modf()

nearbyint() nextafter() nexttoward() pow() quantexp()

remainder() rint() round() scalbn() scalbln()

sin() sinh() sqrt() tan() tanh()

tgamma() trunc()

For example:
- _Decimal32 ceil(_Decimal32)
- _Decimal64 ceil(_Decimal64)
- _Decimal128 ceil(_Decimal128)

_TR1_C99: The following functions provide additional overloads.

acos() acosh() asin() asinh() atan()

atanh() atan2() cbrt() ceil() copysign()

cos() cosh() erf() erfc() exp()

exp2() expm1() fabs() fdim() floor()

fma() fmax() fmin() fmod() frexp()

hypot() ilogb() ldexp() lgamma() llrint()

llround() log() log10() log1p() log2()

logb() lrint() lround() nearbyint() nextafter()

nexttoward() pow() remainder() remquo() rint()

round() scalbln() scalbn() sin() sinh()

sqrt() tan() tanh() tgamma() trunc()

- If any argument corresponding to a double parameter has type long double, all arguments corresponding to double parameters are effectively cast to long double. The return type will be the same return type as if the 'C' long double version of the function was called.
- Otherwise, if any argument corresponding to a double parameter has type double or an integer type, all arguments corresponding to double parameters are effectively cast to double. The return type will be the same return type as if the 'C' double version of the function was called.
- Otherwise, all arguments corresponding to double parameters are effectively cast to float. The return type will be the same return type as if the 'C' float version of the function was called.

Object-like macros: Additional object-like macros provided by the math.h header are described.

Note: The floating point macros and the macros INFINITY and NAN are supported under IEEE only.

DEC_INFINITY
A constant expression of type _Decimal32 representing infinity.

DEC_NAN
A quiet decimal floating NaN for the type _Decimal32.

HUGE_VAL_D32
A constant expression of type _Decimal32 representing +infinity.

HUGE_VAL_D64
A constant expression of type _Decimal64 representing +infinity.

HUGE_VAL_D128
A constant expression of type _Decimal128 representing +infinity.

HUGE_VALF
A very large positive number that expands to a float expression

HUGE_VALL
A very large positive number that expands to a long double expression.

INFINITY
A constant expression of type float representing positive infinity.

NAN A constant expression of type float representing a quiet NaN.

FP_INFINITE
The value of the macro fpclassify for an argument that is plus or minus infinity. This expands to an integer constant expression.

FP_NAN
The value of the macro fpclassify for an argument that is not-a-number (NaN). This expands to an integer constant expression.

FP_NORMAL
The value of the macro fpclassify for an argument that is finite and normalized. This expands to an integer constant expression.

FP_SUBNORMAL
The value of the macro fpclassify for an argument that is finite and denormalized. This expands to an integer constant expression.

FP_ZERO
The value of the macro fpclassify for an argument that is positive or negative. This expands to an integer constant expression.

FP_FAST_FMA
Indicates that the fma function generally executes about as fast as, or faster than, a multiply and an add of double operands.

FP_FAST_FMAF
This is the float version of FP_FAST_FMA.

FP_FAST_FMAL
This is the long double version of FP_FAST_FMA.

Note: Decimal floating-point does not support FP_FAST_FMAD32, FP_FAST_FMAD64, and FP_FAST_FMAD128.

FP_ILOGB0
The value returned by ilogb() if its argument is zero.

FP_ILOGBNAN
The value returned by ilogb() if its argument is a NaN.

MATH_ERRNO
This is defined as value 1 (one) and is used for testing the value of the macro math_errhandling to determine whether a math function reports an error by storing a nonzero value in errno.

MATH_ERREXCEPT
This is defined as value 2 and is used for testing the value of the macro math_errhandling to determine whether a math function reports an error by raising an invalid floating point exception.

math_errhandling
This macro expands to an expression that has type int and the value MATH_ERRNO, MATH_ERREXCEPT, or the bitwise OR of both. This implementation defines this macro as MATH_ERRNO.

Function-like macros or C++ function templates: Additional function-like macros or C++ function templates provided by the <math.h> header are listed.

fpclassify()	isfinite()	ininf()	isgreater()	isgreaterequal()
isless()	islessequal()	islessgreater()	isnan()	isnormal()
isunordered()	signbit()			

The header file includes declarations for the built-in functions abs() and fabs(). For information about built-in functions, see "Built-in functions" on page 93.

The math.h header file declares the macro HUGE_VAL, which expands to a positive `double` constant expression, not necessarily representable as a `float`. Similarly, the macros HUGE_VALF and HUGE_VALL are respectively `float` and `long double` analogs of HUGE_VAL.

For all mathematical functions, a *domain error* occurs when an input argument is outside the range of values allowed for that function. If a domain error occurs, errno is set to the value of EDOM.

A range error occurs if the result of the function cannot be represented in a float, double, long double, _Decimal32, _Decimal64, or _Decimal128 value. If the magnitude of the result is too large (overflow), the function returns the positive or negative value of the macro HUGE_VAL, HUGE_VALF, HUGE_VALL, HUGE_VAL_D32, HUGE_VAL_D64, or HUGE_VAL_D128, as applicable, and sets errno to ERANGE. If the result is too small (underflow), the function returns 0.

A pole error occurs if the function has an exact infinite result as the finite input arguments are approached in the limit (for example, log(0.0)). If a pole error occurs, the function sets errno to ERANGE.

float_t and double_t are floating-point types whose type depends on the value of FLT_EVAL_METHOD. FLT_EVAL_METHOD is 1 which implies both float_t and double_t are double.

Note: Decimal floating-point does not support FP_FAST_FMAD32, FP_FAST_FMAD64, and FP_FAST_FMAD128.

memory.h

The memory.h header file contains declarations for memory operations.

monetary.h

The monetary.h header file contains the declaration for the strfmon() function.

For more information about the effect of locale, see setlocale(), locale.h, or look up the individual functions in this topic. For still more information, see "Internationalization: Locales and Character Sets" in *z/OS XL C/C++ Programming Guide*.

msgcat.h

The msgcat.h header file contains message catalog structures and definitions. The data structures in msgcat.h are not a supported programming interface.

mtf.h

Restriction: This header file is not supported in AMODE 64.

The mtf.h header file contains declarations for the multitasking facility (MTF) functions:

tinit()
tsched()
tsyncro()
tterm()

tsched() is a built-in function.

This header file also contains definitions of macros for certain return values from the above functions.

This header file is supported only under z/OS C applications. The functions are *not* supported under z/OS UNIX.

_Nascii.h

The _Nascii.h header file contains the externals for the correspondence table and functions that support bimodal application development.

ndbm.h

The ndbm.h header file contains definitions for ndbm database operations.

netdb.h

The netdb.h header file contains definitions for network database operations.

net/if.h

The net/if.h header file contains network interface structures and definitions.

_OE_SOCKETS
_OPEN_SYS_IF_EXT 1

The following structures are declared:

- ifaddr
- ifconf
- ifnet
- ifreq

The following macros are declared:

- IF_ADJ
- IF_DEQUEUE
- IF_DEQUEUEIF
- IF_DROP
- IF_ENQUEUE

- IF_PREPEND
- IF_QFULL
- IFF_ALLMULTI
- IFF_BRIDGE
- IFF_BROADCAST
- IFF_CANTCHANGE
- IFF_CHECKSUM
- IFF_DEBUG
- IFF_LOOPBACK
- IFF_MULTICAST
- IFF_NOARP
- IFF_NOTRAILERS
- IFF_POINTOMULTIPT
- IFF_POINTOPOINT
- IFF_PROMISC
- IFF_RUNNING
- IFF_SNAP
- IFF_UP
- IFF_VIRTUAL
- IFNET_SLOWHZ
- IFQ_MAXLEN

_OPEN_SYS_SOCK_IPV6
_POSIX_C_SOURCE 200112L

The following structure is declared:

- if_nameindex

The following macro is declared:

- IF_NAMESIZE

The following functions are declared:

- if_freenameindex
- if_indextoname
- if_nameindex
- if_nametoindex

net/rtrouteh.h

The net/rtrouteh.h header file contains network routing structures and definitions.

netinet/icmp6.h

The netinet/icmp6.h header file defines structures and constants for ICMPv6 header options.

The following structures are exposed with this header file:

- icmp6_hdr
- nd_router_solicit
- nd_router_advert

- nd_neighbor_solicit
- nd_neighbor_advert
- nd_redirect
- nd_opt_hdr
- nd_opt_prefix_info
- nd_opt_rd_hdr
- nd_opt_mtu
- mld_hdr
- icmp6_router_renum
- rr_pco_match
- rr_pco_use
- rr_result

The following definitions are associated with the icmp6_hdr structure:
- icmp6_data32
- icmp6_data16
- icmp6_data8
- icmp6_pptr
- icmp6_mtu
- icmp6_id
- icmp6_seq
- icmp6_maxdelay

The following definitions are associated with ICMPv6 Type and Code values:
- ICMP6_DST_UNREACH
- ICMP6_PACKET_TOO_BIG
- ICMP6_TIME_EXCEEDED
- ICMP6_PARAM_PROB
- ICMP6_INFOMSG_MASK
- ICMP6_ECHO_REQUEST
- ICMP6_ECHO_REPLY
- ICMP6_DST_UNREACH_NOROUTE
- ICMP6_DST_UNREACH_ADMIN
- ICMP6_DST_UNREACH_BEYONDSCOPE
- ICMP6_DST_UNREACH_ADDR
- ICMP6_DST_UNREACH_NOPORT
- ICMP6_TIME_EXCEED_TRANSIT
- ICMP6_TIME_EXCEED_REASSEMBLY
- ICMP6_PARAMPROB_HEADER
- ICMP6_PARAMPROB_NEXTHEADER
- ICMP6_PARAMPROB_OPTION

The following definitions are associated with the nd_router_solicit structure:
- ND_ROUTER_SOLICIT
- nd_rs_type
- nd_rs_code

- nd_rs_cksum
- nd_rs_reserved

The following definitions are associated with the nd_router_advert structure:

- ND_ROUTER_ADVERT
- nd_ra_type
- nd_ra_code
- nd_ra_cksum
- nd_ra_curhoplimit
- nd_ra_flags_reserved
- ND_RA_FLAG_MANAGED
- ND_RA_FLAG_OTHER
- nd_ra_router_lifetime

The following definitions are associated with the nd_neighbor_solicit structure:

- ND_NEIGHBOR_SOLICIT
- nd_ns_type
- nd_ns_code
- nd_ns_cksum
- nd_ns_reserved

The following definitions are associated with the nd_neighbor_advert structure:

- ND_NEIGHBOR_ADVERT
- nd_na_type
- nd_na_code
- nd_na_cksum
- nd_na_flags_reserved
- ND_NA_FLAG_ROUTER
- ND_NA_FLAG_SOLICITED
- ND_NA_FLAG_OVERRIDE

The following definitions are associated with the nd_redirect structure:

- ND_REDIRECT
- nd_rd_type
- nd_rd_code
- nd_rd_cksum
- nd_rd_reserved

The following definitions are associated with the nd_opt_hdr structure:

- ND_OPT_SOURCE_LINKADDR
- ND_OPT_TARGET_LINKADDR
- ND_OPT_PREFIX_INFORMATION
- ND_OPT_REDIRECTED_HEADER
- ND_OPT_MTU

The following definitions are associated with nd_opt_prefix_info structure:

- ND_OPT_PI_FLAG_ONLINK

- ND_OPT_PI_FLAG_AUTO

The following definitions are associated with the mld_hdr structure:
- MLD_LISTENER_QUERY
- MLD_LISTENER_REPORT
- MLD_LISTENER_REDUCTION
- mld_type
- mld_code
- mld_cksum
- mld_maxdelay
- mld_reserved

The following definitions are associated with the icmp6_router_renum structure:
- ICMP6_ROUTER_RENUMBERING
- rr_type
- rr_code
- rr_cksum
- rr_seqnum
- ICMP6_RR_FLAGS_TEST
- ICMP6_RR_FLAGS_REQRESULT
- ICMP6_RR_FLAGS_FORCEAPPLY
- ICMP6_RR_FLAGS_SPECSITE
- ICMP6_RR_FLAGS_PREVDONE

The following definitions are associated with the rr_pco_match structure:
- RPM_PCO_ADD
- RPM_PCO_CHANGE
- RPM_PCO_SETGLOBAL

The following definitions are associated with the rr_pco_use structure:
- ICMP6_RR_PCOUSE_RAFLAGS_ONLINK
- ICMP6_RR_PCOUSE_RAFLAGS_AUTO
- ICMP6_RR_PCOUSE_FLAGS_DECRVLTIME
- ICMP6_RR_PCOUSE_FLAGS_DECRPLTIME
- nd_ns_reserved

The following definitions are associated with the rr_result structure:
- ICMP6_RR_RESULT_FLAGS_OOB
- ICMP6_RR_RESULT_FLAGS_FORBIDDEN

netinet/in.h

The netinet/in.h header file contains definitions for the internet protocol family.

The following structure definition is supported for IPv6:
- struct ip6_mtuinfo{};

The following functions are supported for IPv6:

- bind2addrsel()
- inet6_is_srcaddr()
- inet6_rth_space()
- inet6_rth_init()
- inet6_rth_add()
- inet6_rth_reverse()
- inet6_rth_segments()
- inet6_rth_getaddr()
- inet6_opt_init()
- inet6_opt_append()
- inet6_opt_finish()
- inet6_opt_set_val()
- inet6_opt_next()
- inet6_opt_find()
- inet6_opt_get_val()

The following macros are supported for IPv6:
IN6_IS_ADDR_LINKLOCAL
IN6_IS_ADDR_LOOPBACK
IN6_IS_ADDR_MC_GLOBAL
IN6_IS_ADDR_MC_LINKLOCAL
IN6_IS_ADDR_MC_NODELOCAL
IN6_IS_ADDR_MC_ORGLOCAL
IN6_IS_ADDR_MC_SITELOCAL
IN6_IS_ADDR_MULTICAST
IN6_IS_ADDR_SITELOCAL
IN6_IS_ADDR_UNSPECIFIED
IN6_IS_ADDR_V4COMPAT
IN6_IS_ADDR_V4MAPPED

Structures:

```
struct ip_mreq{
     struct in_addr imr_multiaddr;
     struct in_addr imr_interface;
};
```

Socket options:

- MCAST_INCLUDE
- MCAST_EXCLUDE
- IP_BLOCK_SOURCE
- IP_UNBLOCK_SOURCE
- IP_ADD_SOURCE_MEMBERSHIP
- IP_DROP_SOURCE_MEMBERSHIP
- IPV6_ADDR_PREFERENCES
- MCAST_JOIN_GROUP
- MCAST_LEAVE_GROUP
- MCAST_BLOCK_SOURCE
- MCAST_UNBLOCK_SOURCE
- MCAST_JOIN_SOURCE_GROUP
- MCAST_LEAVE_SOURCE_GROUP

Structure: Multicast filter support is accessed by defining feature test macro _OPEN_SYS_SOCKET_EXT3. The feature test also exposes symbols in sys/socket.h

```
struct ip_mreq{}
struct ip_mreq_source {};
struct group_req {};
struct group_source_req {};
setipv4sourcefilter()
getipv4sourcefilter()
setsourcefilter()
getsourcefilter()
```

netinet/ip6.h

The netinet/ip6.h header file defines structures and constants for IPv6 header options.

The following structures are exposed with this header file:

- ip6_hdr
- ip6_hbh
- ip6_dest
- ip6_rthdr
- ip6_rthdr0
- ip6_frag
- ip6_opt
- ip6_opt_jumbo
- ip6_opt_nsap
- ip6_opt_tunnel
- ip6_opt_router

The following definitions are associated with the ip6_hdr structure:

- ip6_vcf
- ip6_flow
- ip6_plen
- ip6_nxt
- ip6_hlim
- ip6_hops
- ip6_src
- ip6_dst

The following definitions are associated with the extension header structure:

- IP6F_OFF_MASK
- IP6F_RESERVED_MASK
- IP6F_MORE_FRAG

The following definitions are associated with the option header structure:

- IP6OPT_TYPE
- IP6OPT_TYPE_SKIP
- IP6OPT_TYPE_DISCARD
- IP6OPT_TYPE_FORCEICMP
- IP6OPT_TYPE_ICMP

- IP6OPT_MUTABLE
- IP6OPT_PAD1
- IP6OPT_PADN
- IP6OPT_JUMBO
- IP6OPT_NSAP_ADDR
- IP6OPT_TUNNEL_LIMIT
- IP6OPT_ROUTER_ALERT
- IP6OPT_JUMBO_LEN
- IP6_ALERT_MLD
- IP6_ALERT_RSVP
- IP6_ALERT_AN

netinet/tcp.h

The netinet/tcp.h header contains definitions for the Internet Transmission Control Protocol (TCP) .

new

The <new> header file defines several types and functions that control allocation and freeing of storage under program control.

Some of the functions declared in this header are replaceable. The implementation supplies a default version. A program can, however, define a function with the same signature to replace the default version at link time. The replacement version must satisfy the requirements of the function.

```
namespace std {
 typedef void (*new_handler)();
 class bad_alloc;
 class nothrow_t;
 extern const nothrow_t nothrow;
//      FUNCTIONS
 new_handler set_new_handler(new_handler ph) throw();
};
//      OPERATORS
void operator delete(void *p) throw();
void operator delete(void *, void *) throw();
void operator delete(void *p, const std::nothrow_t&) throw();
void operator delete[](void *p) throw();
void operator delete[](void *, void *) throw();
void operator delete[](void *p, const std::nothrow_t&) throw();
void *operator new(std::size_t n) throw(std::bad_alloc);
void *operator new(std::size_t n, const std::nothrow_t&) throw();
void *operator new(std::size_t n, void *p) throw();
void *operator new[](std::size_t n) throw(std::bad_alloc);
void *operator new[](std::size_t n, const std::nothrow_t&) throw();
void *operator new[](std::size_t n, void *p) throw();
```

The <new> header file supercedes the new.h header, which remains for compatibility as a wrapper to <new>.

`bad_alloc`

```
class bad_alloc : public exception {
    };
```

The class describes an exception thrown to indicate that an allocation request did not succeed. The value returned by what() is an implementation-defined C string. None of the member functions throw any exceptions.

```
new_handler
typedef void (*new_handler)();
```

The type points to a function suitable for use as a new handler.

```
nothrow
extern const nothrow_t nothrow;
```

The object is used as a function argument to match the parameter type nothrow_t.

```
nothrow_t
class nothrow_t {};
```

The class is used as a function parameter to operator new to indicate that the function should return a null pointer to report an allocation failure, rather than throw an exception.

new.h

The ISO/ANSI C++ Standard (ISO/IEC 14882:1998(E)) supersedes this header with the new header <new>. However, new.h remains for compatibility as a wrapper for TARGET releases of z/OS V1R2 and later.

For compilations with a TARGET release before z/OS V1R2, the new.h header file declares the set_new_handler() function, which is used for z/OS C++ exception handling (`try`, `throw`, and `catch`). This header file also declares array and non-array version of the allocation operator `new` and the deallocation operator `delete`.

nlist.h

The nlist.h header file declares the nlist() function.

nl_types.h

The nl_types.h header file defines the following types:

nl_item	int used as manifest constant by nl_langinfo()
nl_catd	pointer to a catalog descriptor structure

The header also contains these constants:

NL_SETD NL_CAT_LOCALE

The following functions are prototyped:

catclose() catgets() catopen()

No feature test macro is required for nl_item.

To expose the other definitions in this header, compile with the _XOPEN_SOURCE feature test macro defined.

For more information about the effect of locale, see setlocale(), locale.h, or look up the individual functions in this topic. For more information, see “Internationalization: Locales and Character Sets” in *z/OS XL C/C++ Programming Guide*.

poll.h

The poll.h header file contains definitions for the poll() function.

pthread.h

The pthread.h header file contains function declarations and mappings for threading interfaces and defines a number of constants used by those functions. The header includes the sched.h header. When _UNIX03_THREADS is defined, pthread.h also includes the time.h header. For _OPEN_THREADS applications, pthread.h defines the timespec structure.

There is a lot of overlap in the namespaces identified by the _OPEN_THREADS and _UNIX03_THREADS feature test macros. There are, however, behavioral differences between functions of the same name exposed by _OPEN_THREADS (POSIX.4a Draft 6) and by _UNIX03_THREADS (Single UNIX Specification, Version 3). See the individual function descriptions for specific details.

_OPEN_THREADS 1

pthread_attr_destroy()
pthread_attr_getstacksize()
pthread_attr_setdetachstate()
pthread_cancel()
pthread_cleanup_push()
pthread_condattr_init()
pthread_cond_destroy()
pthread_cond_signal()
pthread_cond_wait()
pthread_detach()
pthread_exit()
pthread_join()
pthread_kill()
pthread_mutexattr_getpshared()
pthread_mutexattr_init()
pthread_mutexattr_settype()
pthread_mutex_init()
pthread_mutex_trylock()
pthread_once()
pthread_rwlockattr_getpshared()
pthread_rwlockattr_setpshared()
pthread_rwlock_init()
pthread_rwlock_tryrdlock()
pthread_rwlock_unlock()
pthread_setintr()
pthread_setspecific()
pthread_tag_np()
pthread_attr_getdetachstate()
pthread_attr_init()
pthread_attr_setstacksize()
pthread_cleanup_pop()
pthread_condattr_destroy()
pthread_cond_broadcast()
pthread_cond_init()
pthread_cond_timedwait()
pthread_create()
pthread_equal()
pthread_getspecific()
pthread_key_create()
pthread_mutexattr_destroy()
pthread_mutexattr_gettype()
pthread_mutexattr_setpshared()
pthread_mutex_destroy()
pthread_mutex_lock()
pthread_mutex_unlock()
pthread_rwlockattr_destroy()
pthread_rwlockattr_init()
pthread_rwlock_destroy()
pthread_rwlock_rdlock()
pthread_rwlock_trywrlock()
pthread_self()
pthread_setintrtype()
pthread_testintr()
pthread_yield()

_OPEN_THREADS 2

pthread_getconcurrency()
pthread_setcancelstate()
pthread_setconcurrency()
pthread_key_delete()
pthread_setcanceltype()
pthread_testcancel()

_OPEN_THREADS 3

pthread_atfork()
pthread_attr_getguardsize()
pthread_attr_getschedparam()
pthread_attr_getstack()
pthread_attr_getstackaddr()
pthread_attr_setguardsize()
pthread_attr_setschedparam()
pthread_attr_setstack()
pthread_attr_setstackaddr()

_UNIX03_THREADS

pthread_atfork()
pthread_attr_destroy()
pthread_attr_getdetachstate()
pthread_attr_getguardsize()
pthread_attr_getschedparam()
pthread_attr_getstack()
pthread_attr_getstackaddr()
pthread_attr_getstacksize()
pthread_attr_init()
pthread_attr_setdetachstate()
pthread_attr_setguardsize()
pthread_attr_setschedparam()
pthread_attr_setstack()
pthread_attr_setstackaddr()
pthread_attr_setstacksize()
pthread_cancel()
pthread_cleanup_pop()
pthread_cleanup_push()
pthread_cond_broadcast()
pthread_cond_destroy()
pthread_cond_init()
pthread_cond_signal()
pthread_cond_timedwait()
pthread_cond_wait()
pthread_condattr_destroy()
pthread_condattr_getpshared()
pthread_condattr_init()
pthread_condattr_setpshared()
pthread_create()
pthread_detach()
pthread_equal()
pthread_exit()
pthread_getconcurrency()
pthread_getspecific()
pthread_join()
pthread_key_create()
pthread_key_delete()
pthread_mutex_destroy()
pthread_mutex_init()
pthread_mutex_lock()
pthread_mutex_trylock()
pthread_mutex_unlock()
pthread_mutexattr_destroy()
pthread_mutexattr_getpshared()
pthread_mutexattr_gettype()
pthread_mutexattr_init()
pthread_mutexattr_setpshared()
pthread_mutexattr_settype()
pthread_once()
pthread_rwlock_destroy()
pthread_rwlock_init()
pthread_rwlock_rdlock()
pthread_rwlock_tryrdlock()
pthread_rwlock_trywrlock()
pthread_rwlock_unlock()
pthread_rwlock_wrlock()
pthread_rwlockattr_destroy()
pthread_rwlockattr_getpshared()
pthread_rwlockattr_init()
pthread_rwlockattr_setpshared()
pthread_self()
pthread_setcancelstate()
pthread_setcanceltype()
pthread_setconcurrency()
pthread_setspecific()
pthread_testcancel()

PTHREAD_CANCEL_ASYNCHRONOUS
PTHREAD_CANCEL_DEFERRED
PTHREAD_MUTEX_DEFAULT
PTHREAD_MUTEX_ERRORCHECK

PTHREAD_CANCEL_DISABLE
PTHREAD_CANCEL_ENABLE
PTHREAD_CANCELED
PTHREAD_COND_INITIALIZER
PTHREAD_CREATE_DETACHED
PTHREAD_CREATE_JOINABLE
PTHREAD_EXPLICIT_SCHED
PTHREAD_INHERIT_SCHED
PTHREAD_MUTEX_INITIALIZER
PTHREAD_MUTEX_NORMAL
PTHREAD_MUTEX_RECURSIVE
PTHREAD_ONCE_INIT
PTHREAD_PROCESS_PRIVATE
PTHREAD_PROCESS_SHARED
PTHREAD_RWLOCK_INITIALIZER_NP

_OPEN_SYS

pthread_attr_getsynctype_np()
pthread_attr_setsynctype_np()
pthread_condattr_getkind_np()
pthread_join_d4_np()
pthread_mutexattr_setkind_np()
pthread_set_limit_np()
pthread_attr_getweight_np()
pthread_attr_setweight_np()
pthread_condattr_setkind_np()
pthread_mutexattr_getkind_np()
pthread_security_np()
pthread_tag_np()

_OPEN_SYS_MUTEX_EXT

pthread_condattr_getpshared()
pthread_condattr_setpshared()

The pthread.h header defines the following constants:

__COND_DEFAULT
__DETACHED
__MEDIUM_WEIGHT
__MUTEX_NONRECXRSIVE
__UNDETACHED
PRIO_INHERIT
PTHREAD_INHERIT_SCHED
PTHREAD_INTR_CONTROLLED
PTHREAD_INTR_ENABLE
PTHREAD_PROCESS_PRIVATE
PTHREAD_SCOPE_GLOBAL
SCHED_FIFO
SCHED_RR
__COND_NODEBUG
__HEAVY_WEIGHT
__MUTEX_NODEBXG
__MUTEX_RECXRSIVE
NO_PRIO_INHERIT
PTHREAD_DEFAULT_SCHED
PTHREAD_INTR_ASYNCHRONOUS
PTHREAD_INTR_DISABLE
PTHREAD_ONCE_INIT
PTHREAD_PROCESS_SHARED
PTHREAD_SCOPE_LOCAL
SCHED_OTHER
PRIO_PROTECT

Furthermore, pthread.h defines these macros:

PTHREAD_MUTEX_DEFAULT
PTHREAD_MUTEX_NORMAL
PTHREAD_MUTEX_RECURSIVE
__THDQ_LENGTH
PTHREAD_MUTEX_ERRORCHECK
PTHREAD_MUTEX_INITIALIZER
PTHREAD_RWLOCK_INITIALIZER

_OPEN_THREADS 2

PTHREAD_CANCEL_ENABLE
PTHREAD_CANCEL_DEFERRED
PTHREAD_CANCEL_DISABLE
PTHREAD_CANCEL_ASYNCHRONOUS

pwd.h

The pwd.h header file declares functions that access the user database through a password structure. The header file also defines the passwd structure.

_POSIX_SOURCE

getpwnam() getpwuid()

_XOPEN_SOURCE_EXTENDED 1

endpwent() getpwent() setpwent()

re_comp.h

The re_comp.h header file contains regular expression matching functions for re_comp().

Note:

This header is kept for historical reasons. It was part of the Legacy Feature in Single UNIX Specification, Version 2, but has been withdrawn and is not supported as part of Single UNIX Specification, Version 3. New applications should use regcomp(), regexec(), regerror() and regfree() functions and the header, which provide full internationalized regular expression functionality compatible with IEEE Std 1003.1-2001 Regular Expressions.

Applications conforming to Single UNIX Specification, Version 3 must not include the <re_comp.h> header file.

regex.h

The regex.h header file contains definitions for the following regular expression functions.

regcomp() regerror() regexec() regfree()

The regex.h header file declares the regex_t type, which can store a compiled regular expression.

The regex.h header file declares the following macros:
- Values of the *cflags* parameter of the regcomp() function: REG_EXTENDED, REG_ICASE, REG_NEWLINE, REG_NOSUB
- Values of the *eflags* parameter of the regexec() function: REG_NOTBOL, REG_NOTEOL
- Values of the *errcode* parameter of the regerror() function: REG_*.

regexp.h

The regexp.h header file contains regular expression declarations.

Note:

This header is kept for historical reasons. It was part of the Legacy Feature in Single UNIX Specification, Version 2, but has been withdrawn and is not supported as part of Single UNIX Specification, Version 3. New applications should use regcomp(), regexec(), regerror() and regfree() functions and the header, which provide full internationalized regular expression functionality compatible with IEEE Std 1003.1-2001 Regular Expressions.

Applications conforming to Single UNIX Specification, Version 3 must not include the <regexp.h> header file.

resolv.h

The resolv.h header file contains the __res_state structure and the definitions to support the IP Address Resolution functions commonly called the Resolver. It contains the prototypes for the following functions — dn_comp(), dn_expand(), dn_find(), dn_skipname(), res_init(), res_mkquery(), res_query(), res_querydomain(), res_search(), and res_send() — which are used to communicate with a Domain Name Server (DNS).

rexec.h

The rexec.h header file declares the rexec() and rexec_af() functions.

sched.h

The sched.h header file declares functions to manipulate and examine process execution scheduling.

_UNIX03_SOURCE

sched_yield()

When compiled with SUSV3 thread support (_UNIX03_THREADS or _XOPEN_SOURCE 600), sched.h defines the following symbols:

SCHED_FIFO	SCHED_OTHER	SCHED_RR

and the sched_param structure.

search.h

The search.h header file contains definitions for searching tables.

setjmp.h

The setjmp.h header file contains function declarations for longjmp() and setjmp(), which use the system stack to affect the program state. It also defines one buffer type, `jmp_buf`, that the setjmp() and longjmp() functions use to save and restore the program state.

_POSIX_SOURCE: setjmp.h declares functions siglongjmp() and sigsetjmp() and defines a buffer type `sigjmp_buf` used by siglongjmp() and sigsetjmp().

_XOPEN_SOURCE_EXTENDED 1: setjmp.h declares the functions _longjmp() and _setjmp().

signal.h

The signal.h header file defines the following values.

- Functions:

raise()
signal()

- Macros:

SIG_DFL
SIG_ERR
SIG_IGN
SIG_PROMOTE

- Signals:

SIGABND
SIGABRT
SIGFPE
SIGILL
SIGINT
SIGIOERR
SIGSEGV
SIGTERM
SIGUSR1
SIGUSR2

- The type `sig_atomic_t`, which is the largest integer type the processor can load or store automatically in the presence of asynchronous interrupts.

The following functions are supported only in a POSIX program. You must specify the POSIX(ON) runtime option for these functions.

kill()
sigaction()
__sigactionset()
sigaddset()
sigdelset()
sigemptyset()
sigfillset()
sigismember()
siglongjmp()
sigpending()
sigprocmask()
sigsuspend()
sigtimedwait()
sigwait()
sigwaitinfo()

The following values are available in z/OS UNIX only:

- Signals:

SIGALRM
SIGCHLD
SIGCLD
SIGCONT
SIGHUP
SIGIO
SIGKILL
SIGPIPE
SIGQUIT
SIGSTOP
SIGTHCONT
SIGTHSTOP
SIGTRACE
SIGTRAP
SIGTSTP
SIGTTIN
SIGTTOU

- The structures `sigaction`, `__sigactionset_t`, `__sigactionset_s`, `sigset_t`, and `pid_t`.
- *options* arguments for sigprocmask(): SIG_BLOCK, SIG_UNBLOCK, and SIG_SETMASK.
- Flags for the `sa_flags` field, available in z/OS UNIX only: SA_NOCLDSTOP and _SA_OLD_STYLE.

_XOPEN_SOURCE_EXTENDED 1:

- Signals:

SIGBUS
SIGPOLL
SIGPROF
SIGSYS
SIGURG
SIGXCPU
SIGXFSZ
SIGVTALRM
SIGWINCH

- Functions:

bsd_signal()
killpg()
sigaltstack()
sighold()
sigignore()
siginterrupt()
sigpause()
sigrelse()
sigset()
sigstack()

Note: bsd_signal() has been marked obsolescent in Single UNIX Specification, Version 3 and may be withdrawn in a future version. The sigaction() function is preferred for portability.

_OPEN_THREADS 2:

- Functions:

  pthread_sigmask()

_UNIX03_THREADS:

- pthread_kill()

spawn.h

The spawn.h header file contains spawn() constants and inheritance structure.

spc.h

Restriction: This header file is not supported in AMODE 64.

This header file is supported only for z/OS C applications.

The spc.h header file contains declarations for the functions available in the system programming environment, as described in "using the System Programming C Facility" in *z/OS XL C/C++ Programming Guide*. The functions are:

edcxregs
edcxusr
edcxusr2
__xhotc()
__xhotl()
__xhott()
__xhotu()
__xregs()
__xsacc()
__xsrvc()
__xusr()
__xusr2()
__24malc()
__4kmalc()

The spc.h header file also declares these functions, used for the allocation of storage and writing of strings, (which are described in Chapter 3, "Library functions," on page 89):

calloc()
free()
malloc()
realloc()
sprintf()

stdalign.h

The stdalign.h header file defines macros that are associated with alignment, which is introduced in the C11 (ISO/IEC 9899:2011) standard:

alignas
alignof
__alignas_is_defined
__alignof_is_defined

stdarg.h

The stdarg.h header file defines macros used to access arguments in functions with variable-length argument lists:

va_arg()
va_copy()
va_start()
va_end()

The stdarg.h header file also defines the structure `va_list`.

stdbool.h

The <stdbool.h> header defines the following macros:

bool expands to _Bool

__bool_true_false_are_defined
expands to 1

false expands to 0

true expands to 1

Restriction: This header is not supported for C++ applications.

stddef.h

The stddef.h header file contains definitions of the commonly used pointers, variables, and types, from the `typedef` statements, as listed below:

`ptrdiff_t`
The signed long type of the result of subtracting two pointers.

`size_t` `typedef` for the type of the value returned by *sizeof*.

`wchar_t`
`typedef` for a wide-character constant.

max_align_t
An object type whose alignment is as great as is supported in all contexts.

stddef.h defines the macros NULL and `offsetof`. NULL is a pointer that never points to a data object. The `offsetof` macro expands to the number of bytes between a structure member and the start of the structure. The `offsetof` macro has the form `offsetof(`*`structure_type`*`, `*`member`*`)`

stdefs.h

The stdefs.h header file contains the same information as found in <stddef.h>.

stdint.h

The stdint.h header defines integer types, limits of specified width integer types, limits of other integer types and macros for integer constant expressions.

Note: For the exact width integer types, minimum width integer types and limits of specified width integer types we support *bit sizes N* with the values 8, 16, 32, and 64.

The following exact width integer types are defined.
- int*N*_t
- uint*N*_t

The following minimum-width integer types are defined.
- int_least*N*_t
- uint_least*N*_t

The following fastest minimum-width integer types are defined. These types are the fastest to operate with among all integer types that have at least the specified width.

- int_fast*N*_t
- uint_fast*N*_t

The following greatest-width integer types are defined. These types hold the value of any signed/unsigned integer type.

Note: Requires long long to be available.

- intmax_t
- uintmax_t

The following integer types capable of holding object pointers are defined.

- intptr_t
- uintptr_t

Object-like macros for limits of integer types: Additional object-like macros provided by the stdint.h header are described.

Note: For the exact width integer limits, minimum width integer limits and limits of specified width integer types we support *bit sizes N* with the values 8, 16, 32, and 64.

Macros for limits of exact width integer types:

- INT*N*_MAX
- INT*N*_MIN
- UINT*N*_MAX

Macros for limits of minimum width integer types:

- INT_LEAST*N*_MAX
- INT_LEAST*N*_MIN
- UINT_LEAST*N*_MAX

Macros for limits of fastest minimum width integer types:

- INT_FAST*N*_MAX
- INT_FAST*N*_MIN
- UINT_FAST*N*_MAX

Macros for limits of greatest width integer types:

Note: Requires long long to be available.

- INTMAX_MAX
- INTMAX_MIN
- UINTMAX_MAX

Macros for limits of pointer integer types:

- INTPTR_MAX
- INTPTR_MIN
- UINTPTR_MAX

Macros for limits of ptrdiff_t:

- PTRDIFF_MAX
- PTRDIFF_MIN

Macros for limits of sig_atomic_t:

- SIG_ATOMIC_MAX
- SIG_ATOMIC_MIN

Macro for limit of size_t:

- SIZE_MAX

Macros for limits of wchar_t:

- WCHAR_MAX
- WCHAR_MIN

Macros for limits of wint_t:

- WINT_MAX
- WINT_MIN

Function-like macros for integer constants: Additional function-like macros provided by the stdint.h header are described.

Note: For the following macro for minimum width integer constants, we support *bit sizes N* with the values 8, 16, 32, and 64.

Macros for minimum width integer constants:

- INTN_C(value)
- UINTN_C(value)

Example:

```
#define _ISOC99_SOURCE
#include <inttypes.h>
#include <stdio.h>


 int main(void)
 {
   uint32_t a = UINT32_C(1234);
   printf("%u\n",a );

 }

Output

1234
```

Example of how the compiler expands the macro:

```
|     uint32_t a = UINT32_C(1234);
+     uint32_t a = 1234U;
```

Macros for greatest width integer constants:

Note: Requires long long to be available.

- INTMAX_C(value)
- UINTMAX_C(value)

Example:

```
/* long long required */
#define _ISOC99_SOURCE
#include <inttypes.h>
#include <stdio.h>


 int main(void)
 {
   intmax_t a = INTMAX_C(45268724);
   printf("%jd\n",a );

 }
```

```
Output

45268724
```

Example of how the compiler expands the macro with the LP64 compiler option:

```
| intmax_t a = INTMAX_C(45268724);
+ intmax_t a = 45268724L;
```

Otherwise the compiler expands to:

```
| intmax_t a = INTMAX_C(45268724);
+ intmax_t a = 45268724LL;
```

stdio.h

The stdio.h header file declares functions that deal with standard input and output. One of these functions, fdopen(), is supported only in a POSIX program.

The stdio.h header file also declares these functions:

clearerr() clrmemf() fclose() fdelrec() feof()
ferror() fflush() fgetc() fgetpos() fgets()
fldata() flocate() fopen() fprintf() fputc()
fputs() fread() freopen() fscanf() fseek()
fseeko() fsetpos() ftell() ftello() fupdate()
fwrite() getc() getchar() gets() perror()
printf() putc() putchar() puts() remove()
rename() rewind() scanf() setbuf() setvbuf()
sprintf() sscanf() svc99() tmpfile() tmpnam()
ungetc() vfprintf() vprintf() vsprintf()

Defined types in stdio.h

The `FILE` type is defined in stdio.h. Stream functions use a pointer to the `FILE` type to get access to a given stream. The system uses the information in the `FILE` structure to maintain the stream. The C standard streams `stdin`, `stdout`, and `stderr` are also defined in stdio.h.

The type `fpos_t` is defined in stdio.h for use with fgetpos() and fsetpos().

The types `__S99parms`, `__S99rbx_t`, and `__S99emparms_t` are defined in stdio.h for use with the svc99() function.

The type `fldata_t` is defined in stdio.h for use with the fldata() function.

The types `__amrc_type` and `__amrc2_type` are defined in stdio.h for use in determining error information when I/O functions fail.

Macros defined in stdio.h

You can use these macros as constants in your programs, but you should not alter their values.

`BUFSIZ` Specifies the buffer size to be used by the setbuf() library function when you are allocating buffers for stream I/O. This value is the expected size of the user's buffer supplied to setbuf(). If a larger buffer is required, for example, if blocksize is larger than BUFSIZ, or if special buffer attributes are required, z/OS XL C/C++ applications will not use the user's buffer.

`EOF` The value returned by an I/O function when the End Of File (EOF) (or in some cases, an error) is found.

`FOPEN_MAX`
The maximum number of files that can be open simultaneously.

`FILENAME_MAX`
The maximum number of characters in a filename. Can be used in the size specification of an array (for example, to hold the filename returned by fldata()).

`L_tmpnam`
The size of the longest temporary name that can be generated by the tmpnam() function.

`L_ctermid`
Maximum size of a character array for ctermid() output. This macro is supported only in a POSIX program.

NULL A pointer which never points to a data object.

`TMP_MAX`
The minimum number of unique file names that can be generated by the tmpnam() function.

The macros `SEEK_CUR`, `SEEK_END`, and `SEEK_SET` expand to integral constant expressions and can be used as the third argument to fseek().

The macros `_IOFBF`, `_IOLBF`, and `_IONBF` expand to integral constant expressions with distinct values suitable for use as the third argument to the setvbuf() function.

The following macros expand to integer constant expressions suitable for interpreting values returned by fldata(), in the `fldata_t` structure.

__APPEND
__BINARY
__BLOCKED
__DISK
__DUMMY
__ESDS
__ESDS_PATH
__HFS
__HIPERSPACE
__KSDS
__KSDS_PATH
__MEMORY
__MSGFILE
__NORLS
__NOTVSAM
__OTHER
__PRINTER
__READ
__RECORD
__RLS
__RSDS
__TAPE
__TDQ
__TERMINAL
__TEXT
__UPDATE
__WRITE

The following macros expand to integral constant expressions suitable for use as the fourth argument to the flocate() function.

__KEY_EQ
__KEY_EQ_BWD
__KEY_FIRST
__KEY_GE
__KEY_LAST
__RBA_EQ
__RBA_EQ_BWD

The following macros expand to integral constant expressions suitable for use as the argument to the function clrmemf().

__CURRENT
__CURRENT_LOWER
__LOWER

The following macros expand to integral constant expressions suitable for use to determine the last operation reported in the `__amrc_type` structure. All these macros are described in *z/OS XL C/C++ Programming Guide*.

__BSAM_BLDL
__BSAM_CLOSE
__BSAM_CLOSE_T
__BSAM_NOTE
__BSAM_OPEN
__BSAM_POINT
__BSAM_READ
__BSAM_STOW
__BSAM_WRITE
__C_CANNOT_EXTEND
__C_DBCS_SI_TRUNCATE
__C_DBCS_SO_TRUNCATE
__C_DBCS_TRUNCATE
__C_DBCS_UNEVEN
__C_FCBCHECK
__C_TRUNCATE
__CELMSGF_WRITE
__CICS_WRITEQ_TD
__HSP_CREATE
__HSP_DELETE
__HSP_EXTEND
__HSP_READ
__HSP_WRITE
__INTERCEPT_READ
__INTERCEPT_WRITE
__IO_CATALOG
__IO_DEVTYPE
__IO_INIT
__IO_LOCATE
__IO_OBTAIN
__IO_RDJFCB
__IO_RENAME
__IO_SCRATCH
__IO_TRKCALC
__IO_UNCATALOG
__LFS_CLOSE
__LFS_FSTAT
__LFS_LSEEK
__LFS_OPEN
__LFS_READ
__LFS_WRITE
__NOSEEK_REWIND
__OS_CLOSE
__OS_OPEN
__QSAM_FREEPOOL
__QSAM_GET
__QSAM_PUT
__QSAM_RELSE
__QSAM_TRUNC
__SVC99_ALLOC
__SVC99_ALLOC_NEW
__SVC99_UNALLOC
__TGET_READ
__TPUT_WRITE
__VSAM_CLOSE
__VSAM_ENDREQ
__VSAM_ERASE
__VSAM_GENCB
__VSAM_GET
__VSAM_MODCB
__VSAM_OPEN_ESDS
__VSAM_OPEN_ESDS_PATH
__VSAM_OPEN_FAIL
__VSAM_OPEN_KSDS
__VSAM_OPEN_KSDS_PATH
__VSAM_OPEN_RRDS
__VSAM_POINT
__VSAM_PUT
__VSAM_SHOWCB
__VSAM_TESTCB

stdio_ext.h

The stdio_ext.h header file contains prototypes and related definitions for the set of stdio extensions that allows access to the internal portions of the `FILE` structure.

stdlib.h

The stdlib.h header file contains declarations for the following functions:

abort()
abs()[1,3]
alloca()[1]
atexit()
atof()
atoi()
atol()
bsearch()
calloc()
cds()[1]
clearenv()
cs()[1]
csid()
div()[3]
exit()
fetch()[2]
fetchep()[2]
free()
getenv()
labs()
ldiv()
llabs()
lldiv()
__librel()
malloc()
mblen()
mbstowcs()
mbtowc()
__moservices()
qsort()
rand()
realloc()
release()[2]
rpmatch()
setenv()
srand()
strtod()
strtol()
strtoll()
strtoul()
strtoull()
system()
unatexit()
wcsid()
wcstombs()

wctomb()	strtod32()[4]	strtod64()[4]	strtod128()[4]

_UNIX03_SOURCE

unsetenv()

[1] Built-in function.

[2] Not supported under C++ applications.

[3] Special Behavior for C++: For C++ applications, the functions abs() and div() are also overloaded for the type long.

[4] The __STDC_WANT_DEC_FP__ feature test macro is required to expose decimal floating-point functionality.

Two type definitions are added to stdlib.h for the Compare and Swap functions cs() and cds(). The structures defined are `__cs_t` and `__cds_t`.

The type `size_t` is declared in the header file. It is used for the type of the value returned by `sizeof`. The type `wchar_t` is declared and used for a wide character constant. For more information on the types `size_t` and `wchar_t`, see “stddef.h” on page 65.

The stdlib.h declares `div_t` and `ldiv_t`, which define the structure types that are returned by div() and ldiv().

The stdlib.h file also contains definitions for the following macros:

NULL The NULL pointer constant (also defined in stddef.h).

EXIT_SUCCESS
Used by the atexit() function.

EXIT_FAILURE
Used by the atexit() function.

RAND_MAX
Expands to an integer representing the largest number that the `RAND` function can return.

MB_CUR_MAX
Expands to an integer representing the maximum number of bytes in a multibyte character. This value is dependent on the current locale.

If MB_CUR_MAX is set to 1, multibyte functions will behave as if all multibyte characters are one byte long; wide-character functions are *not* supported and full DBCS support is *not* provided. If MB_CUR_MAX is 4, all DBCS support provided by the library is enabled.

string.h

The string.h header file declares the string manipulation functions and their built-in versions:

No feature test macro required.

memchr()[1]
memcmp()[1]
memcpy()[1]
memmove()
memset()[1]
strcat()[1]
strchr()[1]
strcmp()[1]
strcoll()
strcpy()[1]
strcspn()
strerror()
strlen()[1]
strncat()[1]
strncmp()[1]
strncpy()[1]
strpbrk()
strrchr()[1]
strspn()
strstr()
strtok()
strxfrm()

_UNIX03_SOURCE

strerror_r

[1] Built-in function.

_XOPEN_SOURCE

memccpy()

_XOPEN_SOURCE_EXTENDED 1

strdup()

The string.h header file also defines the macro NULL and the type `size_t`. For more information see “stddef.h” on page 65.

strings.h

The strings.h header file contains definitions for string operations.

stropts.h

The stropts.h header file declares the following functions:

fattach()
fdetach()
getmsg()
getpmsg()
ioctl()
isastream()
putmsg()
putpmsg()

syslog.h

The syslog.h header file contains definitions for system error logging.

sys/acl.h

The sys/acl.h header enables users to manipulate ACLs. It also declares the following functions:

acl_create_entry()
acl_delete_entry()
acl_delete_fd()
acl_delete_file()
acl_first_entry()
acl_free()
acl_from_text()
acl_get_entry()
acl_get_fd()
acl_get_file()
acl_init()
acl_set_fd()

acl_set_file() acl_sort() acl_to_text() acl_update_entry()
acl_valid()

sys/__cpl.h

The sys/__cpl.h header contains definition for the __cpl() function. It also defines the following constants:

Table 8. Symbolic Constants defined in sys/__cpl.h

Symbolic Constant	Description
CPL_QUERY	Request data from available Coupling Facilities
CPL_CFSIZER	Request a structure size
CPL_CFSIZER_W_LVL	Request a structure size with the level of the CF

sys/file.h

The sys/file.h header file defines file manipulation constants.

sys/__getipc.h

The sys/__getipc.h header file contains definitions to get interprocess communication information.

sys/ioctl.h

The sys/ioctl.h header file contains system I/O definitions and structures.

sys/ipc.h

The sys/ipc.h header file contains definitions for the interprocess communication access structure.

sys/layout.h

The sys/layout.h header file contains declarations for supporting bidirectional (Bidi) conversion of data between Visual (MVS) and Implicit (z/OS UNIX) formats. Initial support is for Arabic and Hebrew data.

sys/mman.h

The sys/mman.h header file contains memory management declarations.

sys/__messag.h

The sys/__messag.h header file contains definitions for the __console() and __console2() functions.

sys/mntent.h

This header file is protected by the _OPEN_SYS feature test macro.

The sys/mntent.h header file declares the w_getmntent() function and it defines the structures `mnte3`, `mnte2`, and `w_mntent`, along with some related constants.

sys/modes.h

The sys/modes.h header file contains several macro definitions:

- Defined constant masks and bits for values of type `mode_t`, such as the `st_mode` field of the `stat` struct
- A defined constant mask for the `st_genvalue` field of the `stat` struct
- Function-like macros for testing values of the `st_mode` field of the `stat` struct.

Under z/OS XL C support, these definitions are included in sys/stat.h to make sys/stat.h conform with POSIX.

sys/msg.h

The sys/msg.h header file contains definitions for message queue structures.

sys/ps.h

The sys/ps.h header file declares the w_getpsent() function that provides process data and defines the structure `w_psproc` along with some related constants.

It requires the `_OPEN_SOURCE 1` feature test macro.

sys/resource.h

The sys/resource.h header file contains definitions for XSI resource operations, including declarations, constants, and structures used by the following functions:

- getpriority()
- getrlimit()
- getrusage()
- setpriority()
- setrlimit()

sys/select.h

The sys/select.h header file contains definitions for select types.

sys/sem.h

The sys/sem.h header file contains definitions for the semaphore facility.

sys/server.h

The sys/server.h header file contains definitions for using WorkLoad Manager services.

sys/shm.h

The sys/shm.h header file contains definitions for the shared memory facility.

sys/socket.h

The sys/socket.h header file contains sockets definitions.

The structure sockaddr_storage is exposed by defining the feature test macro _OPEN_SYS_SOCK_IPV6 or _OPEN_SYS_SOCK_EXT3.

sys/stat.h

The sys/stat.h header file declares the following functions related to z/OS UNIX files and their access:

chaudit()
chmod()
creat()
fchaudit()
fchmod()
fstat()
lstat()
mkdir()
mkfifo()
mknod()
__mount()
mount()
__open_stat()
stat()
umask()
umount()

sys/statfs.h

The sys/statfs.h header file declares the w_statfs() function that provides file system status and the `w_statfs` structure. It requires the _OPEN_SYS feature test macro.

sys/statvfs.h

The sys/statvfs.h header file contains definitions for file system status.

sys/time.h

The sys/time.h header file contains definitions for time types.

sys/timeb.h

The sys/timeb.h header file contains additional definitions for date and time.

sys/times.h

The sys/times.h header file declares the times() function that gets processor times for use by processes. It requires the _POSIX_SOURCE feature test macro.

sys/ttydev.h

The sys/ttydev.h header file defines constants used by the terminal I/O functions.

sys/types.h

The sys/types.h header file defines a collection of *typedef* symbols and structures.

Table 9. sys/types.h: _OE_SOCKETS or _ALL_SOURCE

u_char	unsigned char
u_int	unsigned int
ushort	unsigned short
u_short	unsigned short
u_long	unsigned long

Table 10. sys/types.h: _OE_SOCKETS or _XOPEN_SOURCE_EXTENDED 1

in_addr_t	Internet address
ip_addr_t	Internet address

Table 10. sys/types.h: _OE_SOCKETS or _XOPEN_SOURCE_EXTENDED 1 (continued)

caddr_t	Used for message data pointer

Table 11. sys/types.h: _OPEN_THREADS

pthread_t	Identify a thread
pthread_attr_t	Identify a thread attribute object
pthread_mutex_t	Mutexes
pthread_mutexattr_t	Identify a mutex attribute object
pthread_cond_t	Condition variables
pthread_condattr_t	Identify a condition attribute object
pthread_key_t	Thread-specific data keys
pthread_once_t	Dynamic package initialization

Table 12. sys/types.h: _POSIX_SOURCE

dev_t	Device numbers
gid_t	Group IDs
ino_t	File serial numbers
mode_t	Some file attributes
nlink_t	Link counts
off_t	File sizes, long
pid_t	Process IDs and process group ids
size_t	unsigned long
ssize_t	Signed long
uid_t	user IDs
time_t	Time values
clock_t	Time values, int
sigset_t	Signal set
cc_t	cc_t
tty control chars	
speed_t	tty baud rate
tcflag_t	tty modes
mtm_t	Mount requests
rdev_t	Device numbers

Table 13. sys/types.h: _XOPEN_SOURCE

key_t	Interprocess communications, long

Table 14. sys/types.h: _XOPEN_SOURCE 500

blksize_t	Block sizes
blkcnt_t	File block counts
fsblkcnt_t	Filesystem block counts
fsfilcnt_t	File serial numbers
suseconds_t	Time values in range [-1,1,000,000]

Table 15. sys/types.h: _XOPEN_SOURCE_EXTENDED 1

id_t	General identifier, can contain a pid_t or a gid_t
useconds_t	Microseconds
sa_family_t	Address family
in_port_t	AF_INET port

sys/uio.h

The sys/uio.h header file contains definitions for vector I/O operations.

sys/un.h

The sys/un.h header file contains definitions for UNIX-domain sockets.

sys/__ussos.h

The sys/__ussos.h header file contains the _SET_THLIIPADDR() macro, which sets a client's IP address for security facility authorization (SAF).

sys/utsname.h

The sys/utsname.h header file declares the utsname structure and the uname() function, which returns the name of the current operating system. It requires the _POSIX_SOURCE feature test macro.

sys/wait.h

The sys/wait.h header file declares the following functions, used for holding processes.

_POSIX_SOURCE:

wait()	waitpid()

_XOPEN_SOURCE_EXTENDED 1:

waitid()	wait3()

Note: wait3() has been withdrawn in Single UNIX Specification, Version 3.

sys/__wlm.h

The sys/__wlm.h header file contains definitions for WorkLoad Manager functions.

tar.h

The tar.h header file contains definitions for the tar utility.

terminat.h

The ISO/ANSI C++ Standard (ISO/IEC 14882:1998(E)) supersedes this header with the new header <exception>. However, terminat.h remains for compatibility as a wrapper for TARGET releases of z/OS V1R2 and later.

For compilations with a TARGET release before z/OS V1R2, the terminat.h header file, which is used for z/OS XL C++ exception handling, declares the terminate() and set_terminate() functions.

termios.h

The termios.h header file contains constants, prototypes, and typedef definitions of POSIX terminal I/O functions. It includes the __termcp structure, and declares the following functions:

cfgetispeed() cfgetospeed() cfsetispeed() cfsetospeed() tcdrain()
tcflow() tcflush() tcgetattr() __tcgetcp() tcgetsid()
tcsendbreak() tcsetattr() __tcsetcp() __tcsettables()

These functions are supported only in a POSIX program.

The termios.h header file also contains constants, prototypes and typedef definitions for the w_ioctl() function.

tgmath.h

The header tgmath.h includes the headers math.h and complex.h and defines a number of type-generic macros. This requires the compiler that is designed to support C99.

Use of the macro invokes a function whose corresponding real type and type domain are determined by the arguments for the generic parameters. If there is more than one real floating type argument, usual arithmetic conversions are applied to the real floating type arguments so that they have compatible types. Then,

- If any argument has type _Decimal128, the type determined is _Decimal128.
- Otherwise, if any argument has type _Decimal64, the type determined is _Decimal64.
- Otherwise, if any argument has type _Decimal32, the type determined is _Decimal32.
- Otherwise, if any argument has type long double, the type determined is long double.
- Otherwise, if any argument has type double or is of integer type, the type determined is double.
- Otherwise, if none of the above the type determined is float.

All the functions in math.h and complex.h have their corresponding type generic macros in this header where if for a function in math.h, there is a corresponding c prefixed function in complex.h, then the corresponding type generic macro has the same name as the one in math.h. The macros are:

acos acosh asin asinh atan atan2
atanh carg cbrt ceil cimag conj
copysign cos cosh cproj creal erf
erfc exp exp2 expm1 fabs fdim
floor fma fmax fmin fmod frexp
hypot ilogb ldexp lgamma llrint llround
log log10 log1p log2 logb lrint
lround nearbyint nextafter nexttoward pow remainder
remquo rint round scalbln scalbn sin
sinh sqrt tan tanh tgamma trunc
quantize() samequantum()

[1] The following type-generic macros are not supported for decimal-floating point types: carg(), cimag(), conj(), cproj(), creal().

Restrictions:
- This header does not support the _FP_MODE_VARIABLE feature test macro.
- This header is not supported for C++ applications.

For example:

> The macro `exp(int` *`n`*`)` invokes the function `exp(int` *`n`*`)`
>
> The macro `acosh(float` *`f`*`)` invokes the function `acoshf(float` *`f`*`)`
>
> The macro `log(float complex` *`fc`*`)` invokes the complex function `clogf(float complex fc)`
>
> The macro `pow(double complex` *`dc`*`, float` *`f`*`)` invokes `cpow(double complex` *`dc`*`, float` *`f`*`)`

)

time.h

The time.h header file declares the time and date functions:

asctime() clock() ctime() difftime() gmtime()
localtime() mktime() strftime() strptime() time()
tzset()[1]

[1] These functions are supported only in a POSIX program.

The time.h header file also provides:
- A structure `timespec` containing the following members:
  ```
  time_tv_sec;           seconds
  long tv_nsec;          nanoseconds
  ```
- A structure tm containing the components of a calendar time. See Table 16 on page 80 for a list of the members of the tm structure. This structure is used by the functions `asctime()`, `gmtime()`, `localtime()`, `mktime()`, `strftime()`, and `strptime()`.
- A macro `CLOCKS_PER_SEC` equal to the number per second of the value returned by the clock() function.
- Types `clock_t`, `time_t`, and `size_t`.
- The NULL pointer constant. For more information on NULL and the type `size_t`, see “stddef.h” on page 65.
-

  The macro CLK_TCK, which is the number of clock ticks per second, is kept for historical reasons. It was used in connection with the return value of the clock() function. CLK_TCK has been withdrawn and is not supported as part of Single UNIX Specification, Version 3. New applications should use sysconf(_SC_CLK_TCK) instead of the CLK_TCK macro.

  However, if it is necessary to continue using this symbol in an application written for Single UNIX Specification, Version 3, define the feature test macro _UNIX03_WITHDRAWN before including any standard system headers. The macro exposes all interfaces and symbols removed in Single UNIX Specification, Version 3.

Table 16. Fields of tm Structure

Field	Description
tm_sec	Seconds (0-60)
tm_min	Minutes (0-59)
tm_hour	Hours (0-23)
tm_mday	Day of month (1-31)
tm_mon	Month (0-11; January = 0)
tm_year	Year (current year minus 1900)
tm_wday	Day of week (0-6; Sunday = 0)
tm_yday	Day of year (0-365; January 1 = 0)
tm_isdst	Zero if Daylight Saving Time is not in effect; positive if Daylight Saving Time is in effect; negative if the information is not available.

The time functions are affected by the current locale selected. The LC_CTYPE category affects the behavior of the strftime(), strptime(), and wcsftime() functions. The LC_TOD category affects the behavior of the gmtime(), mktime(), and localtime() functions.

typeinfo

The typeinfo header file defines several types associated with the type-identification operator typeid, which yields information about both static and dynamic types.

```
namespace std {
    class type_info;
    class bad_cast;
    class bad_typeid;
    };
```

```
type_info
```

The class describes type information generated within the program by the implementation. Objects of this class effectively store a pointer to a name for the type, and an encoded value suitable for comparing two types for equality or collating order. The names, encoded values, and collating order for types are all unspecified and may differ between program executions.

An expression of the form typeid x is the only way to construct a (temporary) typeinfo object. The class has only a private copy constructor. Since the assignment operator is also private, you cannot copy or assign objects of class typeinfo either.

```
class type_info {
public:
    virtual ~type_info();
    bool operator==(const type_info& rhs) const;
    bool operator!=(const type_info& rhs) const;
    bool before(const type_info& rhs) const;
    const char *name() const;
private:
    type_info(const type_info& rhs);
    type_info& operator=(const type_info& rhs);
    };
```

```
type_info::operator!=
bool operator!=(const type_info& rhs) const;

The function returns !(*this == rhs).

type_info::operator==

bool operator==(const type_info& rhs) const;

The function returns a nonzero value if *this and rhs represent the
same type.

type_info::before

bool before(const type_info& rhs) const;

The function returns a nonzero value if *this precedes rhs in the
collating order for types.

type_info::name

const char *name() const;

The function returns a C string which specifies the name of the type.
```

bad_cast

```
class bad_cast : public exception {
    };
```

The class describes an exception thrown to indicate that a dynamic cast expression, of the form:

```
 dynamic_cast<type>(expression)
```

generated a null pointer to initialize a reference. The value returned by what() is an implementation-defined C string. None of the member functions throw any exceptions.

bad_typeid

```
class bad_typeid : public exception {
    };
```

The class describes an exception thrown to indicate that a typeid operator encountered a null pointer. The value returned by what() is an implementation-defined C string. None of the member functions throw any exceptions.

typeinfo.h

The ISO/ANSI C++ Standard (ISO/IEC 14882:1998(E)) supersedes this header with the new header <typeinfo>. While this header represents function that did not previously exist on the z/OS and OS/390 operating systems, it is being provided now for compatibility as a wrapper to <typeinfo>.

The typeinfo.h header file contains definitions for types associated with the type-identification operator typeid, which yields information about both static and dynamic types.

uchar.h

This header defines typedefs and macros associated with extended character data types.

The following typedefs are defined:

char16_t char32_t mbstate_t size_t

The following object-like macros are defined:

__STDC_UTF_16__ __STDC_UTF_32__

The uchar.h header file also declares functions that deal with conversions between multibyte characters and char16_t/char32_t. The following functions are declared:

mbrtoc16() mbrtoc32() c16rtomb() c32rtomb()

ucontext.h

The ucontext.h header file contains the prototypes and definitions needed by the following functions:

getcontext() setcontext() makecontext() swapcontext()

uheap.h

The uheap.h header file contains the prototypes and definitions needed by the following functions:

__ucreate() __umalloc() __ufree() __uheapreport()

ulimit.h

The ulimit.h header file contains definitions for ulimit commands.

unexpect.h

The ISO/ANSI C++ Standard (ISO/IEC 14882:1998(E)) supersedes this header with the new header <exception>. However, unexpect.h remains for compatibility as a wrapper for TARGET releases of z/OS V1R2 and later.

For compilations with a TARGET release before z/OS V1R2, the unexpect.h header file, which is used for z/OS XL C++ exception handling, declares the `unexpected()` and `set_unexpected()` functions.

unistd.h

The unistd.h header file declares a number of implementation-specific functions:

__atoe() __atoe_l() __check_resource_auth_np()
__convert_id_np() __etoa() __etoa_l() __isPosixOn()

__smf_record()
__wsinit()

XPLINK

__a2e_l()
__a2e_s()
__e2a_l()
__e2a_s()

There are also a large number of POSIX and UNIX functions declared, shown below with the minimum feature test macro needed to expose them:

access()
alarm()
chdir()
chown()
close()
ctermid()
dup()
dup2()
execl()
execle()
execlp()
execv()
execve()
execvp()
_exit()
fork()
fpathconf()
getcwd()
getegid()
geteuid()
getgid()
getgroups()
getlogin()
getpgrp()
getpid()
getppid()
getuid()
isatty()
link()
lseek()
pathconf()
pause()
pipe()
read()
rmdir()
setgid()
setpgid()
setsid()
setuid()
sleep()
sysconf()
tcgetpgrp()
tcsetpgrp()
ttyname()
unlink()
write()

__certificate()
__getlogin1()
__login()
__pid_affinity()

POSIX1_SOURCE = 2

fchown()
fsync()
ftruncate()
readlink()
setegid()
setgeuid()
symlink()

POSIX_C_SOURCE = 2

optarg
opterr
optind
optopt

External Variables

_XOPEN_SOURCE

chroot()
confstr()
crypt()
cuserid()
encrypt()
getopt()
getpass()
nice()
swab()

_XOPEN_SOURCE = 500

brk()
fchdir()
getdtablesize()
gethostid()
gethostname()
getlogin_r()
getpagesize()
getpgid()
getsid()
getwd()
lchown()
lockf()
pread()
pwrite()
sbrk()
setpgrp()
setregid()
setreuid()
sync()
truncate()
ttyname_r()
ualarm()
usleep()
vfork()

The unistd.h header file also defines many symbols to represent configuration variables and implementation features provided. Some of these are used at compile time, while others are used to interrogate the system at run time, using sysconf(), confstr(), pathconf(), or fpathconf().

utime.h

The utime.h header file declares the `utimbuf` structure and the utime() function, which is used to set file access and modification times.

The utime() function is supported only in a POSIX program.

utmpx.h

The utmpx.h header file contains user accounting database definitions.

varargs.h

The varargs.h header file contains definitions for handling variable argument lists.

Note:

This header is kept for historical reasons. It was part of the Legacy Feature in Single UNIX Specification, Version 2, but has been withdrawn and is not supported as part of Single UNIX Specification, Version 3. New applications should use the <stdarg.h> header to support variable argument list functionality compatible with IEEE Std 1003.1-2001.

Applications conforming to Single UNIX Specification, Version 3 must not include the header file.

variant.h

The variant.h header file declares the getsyntx() function, which returns LC_SYNTAX characters. It also contains the declaration of the variant structure:

```
struct variant {
      char   *codeset;          /* code set of the current locale */
      char    backslash;        /* encoding of \   */
      char    right_bracket;    /* encoding of ]   */
      char    left_bracket;     /* encoding of [   */
      char    right_brace;      /* encoding of }   */
      char    left_brace;       /* encoding of {   */
      char    circumflex;       /* encoding of ^   */
      char    tilde;            /* encoding of ~   */
      char    exclamation_mark; /* encoding of !   */
      char    number_sign;      /* encoding of #   */
      char    vertical_line;    /* encoding of |   */
      char    dollar_sign;      /* encoding of $   */
      char    commercial_at;    /* encoding of @   */
      char    grave_accent;     /* encoding of `   */
};

   struct variant *getsyntx(void);
```

For more information about the effect of locale, see setlocale(), locale.h, or look up the individual functions in this topic. For still more information, see “Internationalization: Locales and Character Sets” in *z/OS XL C/C++ Programming Guide*.

wchar.h

The wchar.h header file contains the declaration for the supported subset of the ISO/C Multibyte Support extensions introduced in ISO/IEC 9899:1990/ Amendment 1:1993(E) extensions. The following functions are declared in wchar.h:

btowc()
fgetwc()
fgetws()
fputwc()
fputws()
fwide()
fwprintf()
fwscanf()
getwc()
getwchar()
mbrlen()
mbrtowc()
mbsinit()
mbsrtowcs()
putwc()
putwchar()
swprintf()
swscanf()
ungetwc()
vfwprintf()
vfwscanf()
vswprintf()
vswscanf()
vwprintf()
vwscanf()
wcrtomb()
wcscat()
wcschr()
wcscmp()
wcscoll()
wcscpy()
wcscspn()
wcsftime()
wcslen()
wcsncat()
wcsncmp()
wcsncpy()
wcspbrk()
wcsrchr()
wcsrtombs()
wcsspn()
wcsstr()
wcstod()
wcstok()
wcstol()
wcstoll()
wcstoul()
wcstoull()
wcswidth()
wcsxfrm()
wctob()
wcwidth()
wmemchr()
wmemcpy()
wmemcmp()
wmemmove()
wmemset()
wprintf()
wscanf()
wcstod32()[1]
wcstod64()[1]
wcstod128()[1]

[1] The __STDC_WANT_DEC_FP__ feature test macro is required to expose decimal floating-point functionality.

wmemchr(), wmemcpy(), wmemcmp(), and wmemset() are also available as their built-in versions.

You don't need to include stdio.h and stdarg.h to use the header file.

The header file wchar.h contains definitions of the following types:

`mbstate_t`
Conversion-state information needed when converting between sequences of multibyte characters and wide characters.

`size_t` `typedef` for the type of the value returned by *sizeof*.

`wchar_t`
`typedef` for a wide-character constant.

`win_t` An integral type unchanged by integral promotions that can hold any value corresponding to members of the extended character set, as well as WEOF (see below).

`FILE` The `FILE` structure type is defined in both stdio.h and wchar.h. Stream functions use a pointer to the `FILE` type to get access to a given stream. The system uses the information in the `FILE` structure to maintain the stream. The C standard streams `stdin`, `stdout`, and `stderr` are also defined in stdio.h.

`va_list`
This type is defined in both stdarg.h and wchar.h.

The header file wchar.h also contains definitions of the following constants:

NULL A pointer that never points to a data object.

WEOF Expands to a constant expression of type `wint_t`, whose value does not correspond to any member of the extended character set. It indicates End Of File (EOF).

WCHAR_MIN
Defines the lower limit of the wchar_t type.

WCHAR_MAX
Defines the upper limit of the wchar_t type.

You can perform wide-character input/output on the streams described in the ISO/IEC 9899:1990 standard, subclause 7.9.2. This standard expands the definition of a stream to include an *orientation* for both text and binary streams. For more information about DBCS orientation, see the section on Double-Byte Characters Sets in *z/OS XL C/C++ Programming Guide*.

The wide-character string functions are also declared in wcstr.h for compatibility with previous releases of C/370™, although wcstr.h may be withdrawn in the future.

For more information about the effect of locale, see setlocale(), locale.h, or look up the individual functions in this topic. For still more information, see the "Internationalization: Locales and Character Sets" in *z/OS XL C/C++ Programming Guide*.

wcstr.h

The wcstr.h header file declares the following multibyte functions:

wcscat() wcschr() wcscmp() wcscpy() wcscspn()
wcslen() wcsncat() wcsncmp() wcsncpy() wcspbrk()
wcsrchr() wcsspn() wcswcs()

wcstr.h also defines the types `size_t`, NULL, `wchar_t`, and `wint_t`.

The wide-character string functions are also declared in wchar.h for compatibility with previous releases of C/370. wcstr.h may be withdrawn in future releases of the z/OS XL C/C++ product.

wcstr.h is a non-standard header provided for compatibility with previous releases of C/370. Functions in wcstr.h are exposed by compiling with LANGLVL(EXTENDED). The wcstr.h header may be withdrawn in future releases of the z/OS XL C/C++ product. The wide character functions in wcstr.h are also declared in the wchar.h header, which is the standard interface.

wctype.h

The wctype.h header declares functions that deal with wide character properties. The following are declared as functions and are also defined as macros:

iswalnum() iswalpha() iswblank() iswcntrl() iswctype()
iswdigit() iswgraph() iswlower() iswprint() iswpunct()
iswspace() iswupper() iswxdigit() towlower() towupper()
wctype()

The following are declared as prototypes only:

towctrans() wctrans() wctype()

The wctype.h header defines the types `wctrans_t`, `wctype_t` and `wint_t`. The wctype.h defines the macro WEOF, which expands to a constant expression of type `wint_t`, whose value does not correspond to any member of the extended character set. The macro WEOF indicates End Of File (EOF).

wordexp.h

The wordexp.h header file contains definitions for word expansion types.

xti.h

The xti.h header file declares the following under the _XOPEN_SOURCE_EXTENDED 1 feature test macro:

Table 17. Symbolic Constants defined in xti.h

Symbolic Constant	Description
TBADADDR	Incorrect addr format
TBADOPT	Incorrect option format
TACCES	Incorrect permissions
TBADF	Illegal transport fd
TNOADDR	Could not allocate addr
TOUTSTATE	Out of state
TBADSEQ	Bad call sequence number
TSYSERR	System error
TLOOK	Event requires attention
TBADDATA	Illegal amount of data
TBUFOVFLW	Buffer not large enough
TFLOW	Flow control
TNODATA	No data
TNODIS	Discon_ind not found on queue
TNOUDERR	unitdata error not found
TBADFLAG	Bad flags
TNOREL	No ord rel found on queue
TNOTSUPPORT	Primitive not supported
TSTATECHNG	State currently changing
TNOSTRUCTYPE	unknown struct-type requested
TBADNAME	Invalid transport name
TBADQLEN	Qlen is zero
TADDRBUSY	Address in use
TINDOUT	Outstanding connect indications
TPROVMISMATCH	Transport provider mismatch
TRESQLEN	Resfd specified to accept w/qlen>0
TRESADDR	Resfd not bound to same addr as fd
TQFULL	Incoming connection queue full
TPROTO	XTI protocol error
T_LISTEN	Connection indication received
T_CONNECT	Connect confirmation received
T_DATA	Normal data received
T_EXDATA	Expedited data received
T_DISCONNECT	Disconnect received
T_UDERR	Datagram error indication
T_ORDREL	Orderly release indication
T_GODATA	Sending normal data is possible
T_GOEXDATA	Sending expedited data is possible
T_EVENTS	Event mask

Table 17. Symbolic Constants defined in xti.h (continued)

Symbolic Constant	Description
T_MORE	More data
T_EXPEDITED	Expedited data
T_NEGOTIATE	Set opts
T_CHECK	Check opts
T_DEFAULT	Get default opts
T_SUCCESS	Successful
T_FAILURE	Failure
T_CURRENT	Current opts
T_PARTSUCCESS	Partial success
T_READONLY	Read-only
T_NOTSUPPORT	Not supported
T_BIND	S

Chapter 3. Library functions

This topic describes the z/OS XL C/C++ Runtime Library functions, including the built-in library functions used by the z/OS XL C/C++ compilers.

Names

Identifiers (function names, macros, types) defined by the various standards in the headers are reserved. Also reserved are:

- Identifiers that begin with an underscore and either an uppercase letter or another underscore.
- Identifiers that end with "_t".

Do not use these reserved identifiers for any purpose other than those defined in the documentation.

All identifiers other than the ISO C identifiers comprise the *user's name space*. You are free to use any of these names. However, a number of names in the z/OS XL C/C++ Runtime Library encroach on the user's name space. This is a result of our desire to provide names that are meaningful and easy to remember, or to support industry-defined names, for example: fetchep() or pthread_cancel(). The header files cause these names to be renamed into reserved names and these in turn are mapped onto the external entry point names that usually are operating-system specific.

If you want to use names in the z/OS XL C/C++ Runtime Library which are in the user's name space as defined, just include the appropriate header. If you cannot include the appropriate header because it would bring in other names that collide with your own private names, but you still want to use some of the functions defined there, you can refer to these functions by their reserved internal names. These reserved names are unique, not longer than 8 characters, and usually start with a double underscore.

The IBM z/OS XL C/C++ compiler automatically maps all underscores and lowercase letters in external identifiers in source code to '@' characters and uppercase characters in the object deck. Thus, to refer to the fetchep() function without including the stdlib.h header, you can use its reserved internal name __ftchep(), which is then automatically mapped to the external entry point `@@FTCHEP`. For C++ functions, you must ensure C linkage by declaring the functions as `extern "C"`.

Functions that are mapped this way have the external entry point listed in the function description in this part under the heading, "External Entry Point".

See also the following section s in *z/OS XL C/C++ Language Reference* for more information about external names:

- "#pragma csect"
- "#pragma map"
- "External Name Mapping"

See also the following section s in *z/OS XL C/C++ User's Guide*:

- "Prelinking a C Application"
- The LONGNAME compiler option

See also “Naming Conventions” in “Using Environment Variables”, in *z/OS XL C/C++ Programming Guide* for details about external names.

Unsupported functions and external variables in AMODE 64

All examples have been tested to work in 31-bit mode. Some examples might not work in 64-bit mode (AMODE 64). As examples are updated for AMODE 64, a statement of AMODE 64 support will be added to the description of the example.

The following functions are not supported in AMODE 64:
- advance()
- brk()
- compile()
- __console()
- __csplist
- ctdli()
- fortrc()__openMvsRel()
- __pcblist
- pthread_quiesce_and_get_np()
- re_comp()
- re_exec()
- regcmp()
- regex()
- sbrk()
- sock_debug_bulk_perf0()
- sock_do_bulkmode()
- step()
- tinit()
- tsched()
- tsetsubt()
- tsyncro()
- tterm()
- valloc()

The following external variables are not supported in AMODE 64:
- __loc1
- loc1
- loc2
- locs

Standards

Each function description begins with a table to indicate the standards/extensions, language support, and dependencies. See the table below for more details:

Standards / Extensions	C or C++	Dependencies
ISO C	C only	POSIX(ON)
ISO C Amendment	C++ only	OS/390 V2R6

Standards / Extensions	C or C++	Dependencies
POSIX.1	both	OS/390 V2R7
POSIX.1a		OS/390 V2R8
POSIX.2		OS/390 V2R9
POSIX.4a		OS/390 V2R10
POSIX.4b		z/OS V1R1
BSD 4.3		z/OS V1R2
XPG4		z/OS V1R3
XPG4.2		z/OS V1R4
SAA		z/OS V1R5
C Library		z/OS V1R6
Language Environment		z/OS V1R7
z/OS UNIX		z/OS V1R8
Single UNIX Specification, Version 2		z/OS V1R9
ISO/ANSI C++		AMODE 64
RFC2292		
RFC2553		
RFC3678		
ANSI/IEEE Standard P754		
C99		
Single UNIX Specification, Version 3		
C/C++ DFP		
C++ TR1 C99		

By indicating a standard, we refer to the origin of the function, not necessarily the compliance. For example, functions that are enriched by features from XPG4 have XPG4 listed.

These are the standards referred to:

- Standards/extensions
  1. *ISO C* refers to ISO/IEC 9899:1990(E).
  2. ISO C Amendment refers to a subset of the ISO/IEC 9899:1990/Amendment 1:1993(E).
  3. POSIX
     - *POSIX.1* refers to ISO/IEC 9945-1:1990/IEEE POSIX 1003.1-1990.
     - *POSIX.1a* refers to a subset of IEEE POSIX 1003.1a, Draft 7, May 1992.
     - *POSIX.2* refers to IEEE Portable Operating System Interface (POSIX) Part 2, P1003.2 draft 12.
     - *POSIX.4a* refers to a subset of IEEE POSIX 1003.4a, Draft 6, Feb. 26,1992.
  4. *XPG4* refers to X/Open Common Applications Environment Specification, System Interfaces and Headers, Issue 4.
  5. *XPG4.2* refers to X/Open Common Applications Environment Specification, System Interfaces and Headers, Issue 4, Version 2.

6. *ISO/ANSI C++* refers to the ISO/ANSI C++ Standard (ISO/IEC 14882:1998(E)).
7. RFC2553 refers to the Basic Socket Interface Extensions for IPv6 (draft-ietf-ipngwg-rfc2253bis-05.txt, dated February 2002).

   Note: Not all of the support described in this draft is available on z/OS.
8. RFC2292 refers to the Advanced Sockets API for IPv6 (draft-ietf-ipngwg-rfc2292bis-06.txt, dated February 25, 2002).

   Note: Not all of the support described in this draft is available on z/OS.
9. C99 refers to ISO/IEC 9899:1999(E).
10. Single UNIX Specification, Version 2 refers to IEEE Std 1003.1-1997.
11. Single UNIX Specification, Version 3 refers to IEEE Std 1003.1-2001.
12. *Extensions* refers to one of the following:
    a. *SAA* refers to the IBM Systems Application Architecture® Common Programming Interface (SAA CPI) Level 2 definition of the C language.
    b. *C Library* refers to the functions that are extensions to the runtime library, before the Language Environment product.
    c. *Language Environment* refers to functions that are extensions to the conventional standards.
    d. *z/OS UNIX* refers to functions that provide z/OS UNIX support beyond the defined standards.
13. C/C++ DFP refers to:
    a. ISO/IEC TR24732 -- Extensions for the programming language C to support decimal floating point arithmetic.
    b. ISO/IEC TR24733 -- Extensions for the programming language C++ to support decimal floating point arithmetic.
14. C++ TR1 C99 refers to ISO/IEC DTR 19768 - Draft Technical Report on C++ library Extensions, chapter 8, C compatibility.

- Language support

  C or C++ refers to whether the function is supported for the z/OS XL C compiler, the z/OS XL C++ compiler, or both.
- Dependencies

  Some functions have the following dependencies identified. If the dependencies are not met, then the function fails, and returns an errno of EMVSNORTL. Functions defined by the standards that cannot fail, will cause abnormal termination and return Language Environmen condition CEE5001.

  – POSIX(ON) *required* refers to whether the enclave can run with the POSIX semantics.

    POSIX is an application characteristic that is maintained at the enclave level. After you have established the characteristic during enclave initialization, you cannot change it.

    When you set POSIX to ON, you can use functions that are unique to POSIX, such as pthread_create().

    One of the effects of POSIX(ON) is the enablement of POSIX signal handling semantics, which interact closely with the z/OS Language Environment condition handling semantics. Where ambiguities exist between ANSI and POSIX semantics, the POSIX runtime setting indicates that the POSIX semantics will be followed.

These standards do have some overlap, as illustrated in Figure 1.

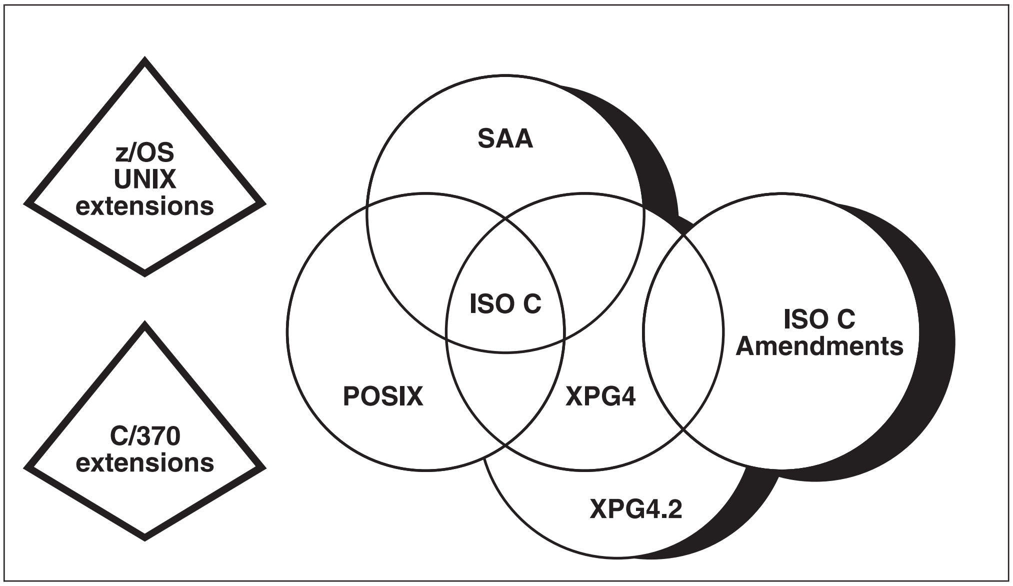


Figure 1. Overlap of C Standards and Extensions

The C library contains several functions that are extensions to the SAA CPI Level 2 definition. These library functions are available only if the LANGLVL(EXTENDED) compile-time option is in effect. As indicated, some of the *stub* routines for the extensions are available if you specify LANGLVL(ANSI). They are made available for compatibility with Version 1; they may not be available in the future. (Within runtime libraries, a *stub routine* is a routine that contains the minimum lines of code required to locate a given routine at run time.)

Many of the symbols that are defined in headers are “protected” by a feature test macro. For information on the relationships between feature test macros and the standards, see “Feature test macros” on page 3.

Using C include files from C++

If you need to use an old C header file in a C++ program, use `extern`, like this:

```
extern "C" {
  #include "myhdr.include"
}
```

Built-in functions

Built-in functions are ones for which the compiler generates inline code at compile time. Every call to a built-in function eliminates a runtime call to the function having the same name in the dynamic library.

Built-in functions are used by application code, while it is running, without reference to the dynamic library. Although built-in functions increase the size of a generated application slightly, this should be offset by the improved performance resulting from reducing the overhead of the dynamic calls. Built-in functions can be used with the System Programming C (SPC) Facilities to generate free-standing C applications.

Restriction: The SPC facility is not supported in AMODE 64.

Table 18 shows all of the built-in functions. In the listing of library functions, each built-in function is labelled as such.

Table 18. Built-in Library Functions

abs()	alloca()	cds()	cs()	decabs()
decchk()	decfix()	fabs()	fortrc()	memchr()
memcmp()	memcpy()	memset()	strcat()	strchr()
strcmp()	strcpy()	strlen()	strncat()	strncmp()
strncpy()	strrchr()	tsched()	wmemchr()	wmemcmp()
wmemcpy()	wmemset()			

The built-in versions of these functions are accessed by preprocessor macros defined in the standard header files. They are not used unless the appropriate header file (such as `decimal.h`, `math.h`, `stdlib.h`, `string.h` or `wchar.h`) is included in the source file.

The built-in versions of functions wmemchr(), wmemcmp(), wmemcpy(), and wmemset() are only available under ARCH(7) when LP64 is not used.

Your program will use the built-in version of a standard function only if you include the associated standard header file. However, decfix(), decabs(), and decchk() are implemented only as built-in functions. They are not available without including the header file.

If you are using the standard header file, but want to use the function in the dynamic library instead of the built-in function, you can force a call to the dynamic library by putting parentheses around the function name in your source code:

```
(memcpy)(buf1, buf2, len)
```

The built-in functions are documented in Using hardware built-in functions in *z/OS XL C/C++ Programming Guide*.

If you will never use the built-in version, you can also use `#undef` with the function name. For example, `#undef memcpy` causes all calls to `memcpy` in the compilation unit to make a dynamic call to the function rather than using the built-in version.

IEEE binary floating-point

Starting with OS/390 V2R6 (including the Language Environment and C/C++ components), support has been added for IEEE binary floating-point (IEEE floating-point) as defined by the ANSI/IEEE Standard 754-1985, IEEE Standard for Binary Floating-Point Arithmetic.

Notes:

1. You must have OS/390® Release 6 or higher to use the IEEE Binary Floating-Point instructions. In Release 6, the base control program (BCP) is enhanced to support the new IEEE Binary Floating-Point hardware in the IBM S/390® Generation 5 Server. This enables programs running on OS/390 Release 6 to use the IEEE Binary Floating-Point instructions and 16 floating-point registers.In addition, the BCP provides simulation support for all the new floating-point hardware instructions. This enables applications that make light use of IEEE Binary Floating-Point, and can tolerate the overhead of software simulation, to execute on OS/390 V2R6 without requiring an IBM S/390 Generation 5 Server.

2. The terms *binary floating point* and *IEEE binary floating point* are used interchangeably. The abbreviations BFP and HFP, which are used in some function names, refer to binary floating point and hexadecimal floating point respectively.
3. Under Hexadecimal Floating-Point format, the rounding mode is set to round toward 0. Under IEEE Binary Floating-Point format, the rounding mode is to round toward the nearest integer.

The z/OS XL C/C++ compiler provides a FLOAT option to select the format of floating-point numbers produced in a compile unit. The FLOAT option allows you to select either IEEE Binary Floating-Point or hexadecimal floating-point format. For details on the z/OS XL C/C++ support, see the description of the FLOAT option in *z/OS XL C/C++ User's Guide*. In addition, two related sub-options have been introduced, ARCH(3) and TUNE(3). The two sub-options support the new G5 processor architecture, and IEEE binary floating-point data. Refer to the ARCHITECTURE and TUNE compiler options in *z/OS XL C/C++ User's Guide* for details.

The C/C++ runtime library interfaces, which formerly supported only hexadecimal floating-point format, have been changed in OS/390 V2R6 to support both IEEE Binary Floating-Point and hexadecimal floating-point formats. These interfaces are documented in the z/OS XL C/C++ Runtime Library Reference.

The primary documentation for the IEEE Binary Floating-Point support is contained in *z/Architecture Principles of Operation* and *z/OS XL C/C++ User's Guide*.

IEEE binary floating point support provides interoperability and portability between platforms. It is anticipated that the support will be most commonly used for new and ported applications and in emerging environments, such as Java. Customers should not migrate existing applications that use hexadecimal floating point to binary floating point, unless there is a specific reason.

IBM does not recommend mixing floating-point formats in an application. However, for applications which must handle both formats, the C/C++ runtime library does offer some support. Reference information for IEEE Binary Floating-Point can also be found in *z/OS XL C/C++ Language Reference*.

IEEE decimal floating-point

Starting with z/OS V1R9 (including the Language Environment and C/C++ components), support has been added for IEEE decimal floating-point as defined by the ANSI/IEEE Standard P754/D0.15.3, IEEE Standard for Floating-Point Arithmetic.

Note:

1. You must have z/OS V1R9 or higher to use IEEE decimal floating-point, the hardware must have the Decimal Floating Point Facility installed and the __STDC_WANT_DEC_FP__ feature test macro must be defined.
2. The abbreviation DFP refers to IEEE Decimal Floating-Point.
3. IEEE decimal floating-point is not supported in a CICS environment.

   The z/OS XL C/C++ compiler provides a DFP option to include support for IEEE Decimal Floating-Point numbers. For details on the z/OS XL C/C++ support, see the description of the DFP option in *z/OS XL C/C++ User's Guide*. New C/C++ runtime library interfaces, which support IEEE Decimal Floating Point numbers have been added for z/OS V1R9 and other existing interfaces have been updated to support DFP. These interfaces are documented in the *z/OS XL C/C++ Runtime Library Reference*. The primary documentation for the IEEE decimal floating-point support is contained in *z/Architecture Principles of Operation* and *z/OS XL C/C++ User's Guide*. Reference information for IEEE floating-point can also be found in *z/OS XL C/C++ Language Reference*.

4. When one or more input values for a Decimal Floating Point (DFP) library function are not in the preferred Densely Packed Decimal (DPD) encoding, it is not defined whether or not the output values are converted to the preferred DPD coding. Applications should not rely on the current behavior of library functions regarding the DPD recoding of output values.

External variables

The POSIX 1003.1 and X/Open CAE Specification 4.2 (XPG4.2) require that the C system header files define certain external variables. Additional variables are defined for use with POSIX or XPG4.2 functions. If you define one of the POSIX or XPG4 feature test macros and include one of these headers, the external variables will be defined in your program. These external variables are treated differently compared with other global variables in a multithreaded environment (values are thread-specific) and across a call to a fetched module (values are propagated).

To access the global variable values the following must be specified during C/C++ compiles and z/OS bind:

Non-XPLINK (non-thread-safe)
C code must be compiled with the RENT or DLL option (C++ code needs no additional options). The SCEEOBJ autocall library must be specified during the bind.

Non-XPLINK (thread-safe)
No additional options are required for either C or C++. The _SHARE_EXT_VARS feature test macro, or the necessary _SHR_ prefixed feature test macros must be used.

Equivalently, the necessary thread-specific functions can be called directly (as documented below under each external variable).

XPLINK (non-thread-safe)
No additional options (besides XPLINK) are required for either C or C++. The C runtime library side-deck, member CELHS003 of the SCEELIB data set, must be included during the bind. (c89/cc/c++ automatically include this side-deck when the XPLINK link edit option (for example, c89 -Wl,XPLINK ...) is used.)

XPLINK (thread-safe)
No additional options (besides XPLINK) are required for either C or C++. The C runtime library side-deck, member CELHS003 of the SCEELIB dataset, must be included during the bind.

The _SHARE_EXT_VARS feature test macro, or the necessary _SHR_ prefixed feature test macros must be used. Equivalently, the necessary thread-specific functions can be called directly (as documented in the later sections under each external variable).

LP64 (non-thread-safe)
No additional options (besides LP64) are required for either C or C++. The C runtime library side-deck, member CELQS003 of the SCEELIB dataset, must be included during the bind.

The _SHARE_EXT_VARS feature test macro, or the necessary _SHR_ prefixed feature test macros must be used.

Equivalently, the necessary thread-specific functions can be called directly (as documented in the later sections under each external variable).

LP64 (thread-safe)

No additional options (besides LP64) are required for either C or C++. The C runtime library side-deck, member CELQS003 of the SCEELIB dataset, must be included during the bind. (c89/cc/c++ automatically include this side-deck when the LP64 link edit option (for example, c89 -Wl, LP64 ...) is used.)

The _SHARE_EXT_VARS feature test macro, or the necessary _SHR_ prefixed feature test macros must be used.

Equivalently, the necessary thread-specific functions can be called directly (as documented in the later sections under each external variable).

errno

When a runtime library function fails, the function may do any of the following to identify the error:

- Set errno to a documented value.
- Set errno to a value that is not documented. You can use strerror() or perror() to get the message associated with the errno.
- Not set errno.
- Clear errno.

See also **errno.h**.

daylight

Daylight savings time flag set by tzset(). Note that other time zone sensitive functions such as ctime(), localtime(), mktime(), and strftime() implicitly call tzset().

Note: Use the `__dlght()` function to access the thread-specific value of `daylight`. See also **time.h**.

getdate_err

The variable is set to the value below when an error occurs in the getdate() function.

1. The DATEMSK environment variable is NULL or undefined.
2. The template file cannot be opened for reading.
3. Failed to get file status information.
4. The template file is not a regular file.
5. An error was encountered while reading the template file.
6. Memory allocation failed (not enough memory available).
7. No line in the template file matches the input specification.
8. Non-valid input specification. For example, February 31; or a time that can not be represented in a time_t (representing the time is seconds since Epoch - midnight, January 1, 1970 (UTC)).
9. Unable to determine current time.

Any changes to errno are unspecified.

Note: This value is unique for z/OS UNIX.

The getdate64() interface affects the same pointer to the thread-specific value of getdate_err and uses the same getdate_err values as the getdate() interface.

Note: Use the**__gderr()** function to access the thread-specific value of `getdate_err`. The getdate64() function affects the same pointer to the thread-specific value of getdate_err as the getdate() function does.

See also **time.h**.

h_errno

An integer which holds the specific error code when the network name server encounters an error. The network name server is used by the gethostbyname() and gethostbyaddr() functions.

Note: Use the**__h_errno()** function to access the thread-specific value of `h_errno`. See also **netdb.h**.

Note: This variable is kept for historical reasons. However, it is used only in connection with the functions gethostbyaddr() and gethostbyname(), which are obsolescent in Single UNIX Specification, Version 3, so that the h_errno variable may also be withdrawn in the future.

__loc1

Restriction: This external variable is not supported in AMODE 64

A global character pointer which is set by the regex() function to point to the first matched character in the input string. Use the **____loc1()** function to access the thread-specific value of `__loc1`.

Note:

This variable is kept for historical reasons. It was part of the Legacy Feature in Single UNIX Specification, Version 2, but has been withdrawn and is not supported as part of Single UNIX Specification, Version 3. New applications should use the interfaces supported by the <regex.h> header, which provide full internationalized regular expression functionality compatible with IEEE Std 1003.1-2001 Regular Expressions.

If it is necessary to continue using this symbol in an application written for Single UNIX Specification, Version 3, define the feature test macro _UNIX03_WITHDRAWN before including any standard system headers. The macro exposes all interfaces and symbols removed in Single UNIX Specification, Version 3. See also **libgen.h**.

loc1

Restriction: This external variable is not supported in AMODE 64.

A pointer to characters matched by regular expressions used by step(). The value is not propagated across a call to a fetched module.

Note:

This variable is kept for historical reasons. It was part of the Legacy Feature in Single UNIX Specification, Version 2, but has been withdrawn and is not supported as part of Single UNIX Specification, Version 3. New applications should use the interfaces supported by the <regex.h> header, which provide full internationalized regular expression functionality compatible with IEEE Std 1003.1-2001 Regular Expressions. See also **regexp.h**.

loc2

Restriction: This external variable is not supported in AMODE 64

A pointer to characters matched by regular expressions used by step(). The value is not propagated across a call to a fetched module.

Note:

This variable is kept for historical reasons. It was part of the Legacy Feature in Single UNIX Specification, Version 2, but has been withdrawn and is not supported as part of Single UNIX Specification, Version 3. New applications should use the interfaces supported by the <regex.h> header, which provide full internationalized regular expression functionality compatible with IEEE Std 1003.1-2001 Regular Expressions. See also **regexp.h**.

locs

Restriction: This external variable is not supported in AMODE 64

Used by advance() to stop regular expression matching in a string. The value is not propagated across a call to a fetched module. See also **regexp.h**.

Note:

This variable is kept for historical reasons. It was part of the Legacy Feature in Single UNIX Specification, Version 2, but has been withdrawn and is not supported as part of Single UNIX Specification, Version 3. New applications should use the interfaces supported by the <regex.h> header, which provide full internationalized regular expression functionality compatible with IEEE Std 1003.1-2001 Regular Expressions.

optarg

Character pointer used by getopt() for options parsing variables.

Note: Use the**__opargf()** function to access the thread-specific value of `optarg`.

Note: This variable has been removed from stdio.h by Single UNIX Specification, Version 3 and is exposed for Version 3 only in unistd.h . See also **stdio.h** and **unistd.h**.

opterr

Error value used by getopt().

Note: Use the **__operrf()** function to access the thread-specific value of `opterr`.

Note: This variable has been removed from stdio.h by Single UNIX Specification, Version 3 and is exposed for Version 3 only in unistd.h. See also **stdio.h** and **unistd.h**.

optind

Integer pointer used by getopt() for options parsing variables.

Note: Use the**__opindf()** function to access the thread-specific value of `optind`.

Note: This variable has been removed from stdio.h by Single UNIX Specification, Version 3 and is exposed for Version 3 only in unistd.h. See also **stdio.h** and **unistd.h**.

optopt

Integer pointer used by getopt() for options parsing variables.

Note: Use the**__opoptf()** function to access the thread-specific value of `optopt`.

Note: This variable has been removed from stdio.h by Single UNIX Specification, Version 3 and is exposed for Version 3 only in unistd.h. See also **stdio.h** and **unistd.h**.

signgam

Storage for sign of lgamma(). This function defaults to thread-specific. See also **math.h**.

stderr

Standard Error stream. The external variable will be initialized to point to the enclave-level stream pointer for the standard error file. There is no multithreaded function. See also **stdio.h**.

stdin

Standard Input stream. The external variable will be initialized to point to the enclave-level stream pointer for the standard input file. There is no multithreaded function. See also **stdio.h**.

stdout

Standard Output stream. The external variable will be initialized to point to the enclave-level stream pointer for the standard output file. There is no multithreaded function. See also **stdio.h**.

t_errno

An integer which holds the specific error code when a failure occurs in one of the X/Open Transport Interface (XTI) functions. Use the **__t_errno()** function to access the thread-specific value of `t_errno`.

Note: Use the**__t_errno()** function to access the thread-specific value of `t_errno`. See also **xti.h**.

timezone

Long integer difference from UTC and standard time as set by tzset(). Note that other time zone sensitive functions such as ctime(), localtime(), mktime(), and strftime() implicitly call tzset().

Note: Use the __tzone() function to access the thread-specific value of `timezone`. See also **time.h**.

tzname

Character pointer to unsized array of timezone strings used by tzset() and ctime(). The *tzname variable contains the Standard and Daylight Savings time zone names. If the TZ environment variable is present and correct, tzname will be set from TZ. Otherwise tzname will be set from the LC_TOD locale category. See the tzset() function for a description. There is no multithreaded function. See also **time.h**.

The __restrict__ macro

The restrict keyword is being made available in the form of a macro named __restrict__ which can be used for coding before the availability of a compiler that is designed to support C99. Once the compiler support is available, only a recompile will be necessary. Applications need to include <features.h> before using the __restrict__ macro.

The __noreturn__ macro

The _Noreturn keyword is being made available in the form of a macro named __noreturn__ which can be used for coding before the availability of a compiler that is designed to support C11. Once the compiler support is available, only a recompile will be necessary. Applications need to include <features.h> before using the __noreturn__ macro.

abort() — Stop a program

Standards

Standards / Extensions	C or C++	Dependencies
ISO C POSIX.1 XPG4 XPG4.2 C99 C11 Single UNIX Specification, Version 3	both	

Format

```
#include <stdlib.h>

__noreturn__ void abort(void);
```

General description

Causes an abnormal program termination and returns control to the host environment. The abort() function flushes all buffers and closes all open files. Be aware that abnormal termination will *not* run the atexit() list functions.

If the abort() function is called and the user has a handler for SIGABRT, then SIGABRT is raised; however, SIGABRT is raised again when the handler associated with the default action is returned. The code path only passes through the user's handler once, even if the handler is reset. The same thing occurs if SIGABRT is ignored; abnormal termination occurs.

The abort() function will not result in program termination if SIGABRT is caught by a signal handler, and the signal handler does not return. You can avoid returning by “jumping” out of the handler with setjmp() and longjmp(). In z/OS XL C programs, you can jump out of the handler with sigsetjmp() and siglongjmp().

For more information see the process termination sections in the chapter “Using Runtime User Exits” in *z/OS XL C/C++ Programming Guide*.

Special behavior for POSIX C: To obtain access to the special POSIX behavior for abort(), the POSIX runtime option must be set ON.

Calls to abort() raise the SIGABRT signal, using pthread_kill(), so that the signal is directed to the same thread. A SIGABRT signal generated by abort() cannot be blocked.

Under POSIX, the handler can use siglongjmp() to restore the environment at a place in the code where a sigsetjmp() was previously issued. In this way, an application can avoid the process for termination.

Special behavior for C++: If abort() is called from a z/OS XL C++ program, the program will be terminated immediately, without leaving the current block. Functions passed to atexit(), and destructors for static and local (automatic) objects, will not be called.

By default, the z/OS XL C++ function terminate() calls abort().

Returned value

The abnormal termination return code for z/OS is 2000.

Example

CELEBA01

```
/* CELEBA01

   This example tests for successful opening of the file myfile.
   If an error occurs, an error message is printed and the program
   ends with a call to the abort() function.

 */
#include <stdio.h>
#include <stdlib.h>

int main(void)
{
   FILE *stream;
   unlink("myfile.dat");
   if ((stream = fopen("myfile.dat", "r")) == NULL)
   {
      printf("Could not open data file\n");
      abort();

      printf("Should not see this message\n");
   }
}
```

Related information

- “stdlib.h” on page 70
- “assert() — Verify condition” on page 191
- “atexit() — Register program termination function” on page 199
- “exit() — End program” on page 443
- “pthread_kill() — Send a signal to a thread” on page 1283
- “raise() — Raise signal” on page 1371
- “signal() — Handle interrupts” on page 1643

abs(), absf(), absl() — Calculate integer absolute value

Standards

Standards / Extensions	C or C++	Dependencies
ISO C POSIX.1 XPG4 XPG4.2 C99 Single UNIX Specification, Version 3 C/C++ DFP C++ TR1 C99	both	

Format

```
#include <stdlib.h>

int abs(int n);
long abs(long n);                   /* C++ only */

#include <math.h>

double abs(double n);               /* C++ only */
float abs(float n);                 /* C++ only */
long double abs(long double n);     /* C++ only */
float absf(float n);
long double absl(long double n);
```

DFP:

```
#define __STDC_WANT_DEC_FP__
#include <math.h>

_Decimal32  abs(_Decimal32 x);   /* C++ only */
_Decimal64  abs(_Decimal64 x);   /* C++ only */
_Decimal128 abs(_Decimal128 x);  /* C++ only */
```

C++ TR1 C99:

```
#define _TR1_C99
#include <inttypes.h>
intmax_t abs(intmax_t n);

#define _TR1_C99
#include <stdlib.h>
long long abs(long long n);
```

General description

The functions abs(), absf(), and absl() return the absolute value of an argument *n*.

For the integer version of abs(), the minimum allowable integer is `INT_MIN+1`. (`INT_MIN` is a macro that is defined in the `limits.h` header file.) For example, with the z/OS XL C/C++ compiler, `INT_MIN+1` is -2147483648.

For the double, float, and long double versions of abs(), the minimum allowable values are `DBL_MIN+1`, `FLT_MIN+1`, and `LDBL_MIN+1`, respectively. (The floating-point macro constants are defined in the `float.h` header file.)

If the value entered cannot be represented as an integer, the abs(), absf(), and absl() functions return the same value.

Note: These functions work in both IEEE Binary Floating-Point and hexadecimal floating-point formats. See “IEEE binary floating-point” on page 94 for more information about IEEE Binary Floating-Point.

Special behavior for C++: For C++ applications, abs() is also overloaded for the types long, float, and long double.

Returned value

The returned value is the absolute value, if the absolute value is possible to represent.

Otherwise the input value is returned.

There are no errno values defined.

Example

CELEBA02

```
/* CELEBA02

   This example calculates the absolute value of an integer
   x and assigns it to y.

 */
#include <stdio.h>
#include <stdlib.h>

int main(void)
{
    int x = -4, y;

    y = abs(x);
    printf("The absolute value of %d is %d.\n", x, y);
}
```

Output

```
The absolute value of    -4 is    4.
```

Related information

- “float.h” on page 29
- “limits.h” on page 39
- “math.h” on page 44
- “stdlib.h” on page 70
- “fabs(), fabsf(), fabsl() — Calculate floating-point absolute value” on page 460
- “labs() — Calculate long absolute value” on page 935

accept() — Accept a new connection on a socket

Standards

Standards / Extensions	C or C++	Dependencies
XPG4.2 Single UNIX Specification, Version 3	both	

Format

X/Open:

```
#define _XOPEN_SOURCE_EXTENDED 1
#include <sys/socket.h>

int accept(int socket, struct sockaddr *__restrict__ address,
           socklen_t *__restrict__address_len);
```

Berkeley sockets:

```
#define _OE_SOCKETS
#include <sys/types.h>
#include <sys/socket.h>

int accept(int socket, struct sockaddr *address, int *address_len);
```

General description

The accept() call is used by a server to accept a connection request from a client. When a connection is available, the socket created is ready for use to read data from the process that requested the connection. The call accepts the first connection on its queue of pending connections for the given socket *socket*. The accept() call creates a new socket descriptor with the same properties as *socket* and returns it to the caller. If the queue has no pending connection requests, accept() blocks the caller unless *socket* is in nonblocking mode. If no connection requests are queued and *socket* is in nonblocking mode, accept() returns -1 and sets the error code to EWOULDBLOCK. The new socket descriptor cannot be used to accept new connections. The original socket, *socket*, remains available to accept more connection requests.

Parameter
Description

socket The socket descriptor.

address The socket address of the connecting client that is filled in by accept()

before it returns. The format of *address* is determined by the domain that the client resides in. This parameter can be NULL if the caller is not interested in the client address.

address_len
: Must initially point to an integer that contains the size in bytes of the storage pointed to by *address*. On return, that integer contains the size required to represent the address of the connecting socket. If this value is larger than the size supplied on input, then the information contained in sockaddr is truncated to the length supplied on input. If *address* is NULL, *address_len* is ignored.

The *socket* parameter is a stream socket descriptor created with the socket() call. It is usually bound to an address with the bind() call. The listen() call marks the socket as one that accepts connections and allocates a queue to hold pending connection requests. The listen() call places an upper boundary on the size of the queue.

The *address* parameter is a pointer to a buffer into which the connection requester's address is placed. The *address* parameter is optional and can be set to be the NULL pointer. If set to NULL, the requester's address is not copied into the buffer. The exact format of *address* depends on the addressing domain from which the communication request originated. For example, if the connection request originated in the AF_INET domain, *address* points to a **sockaddr_in** structure, or if the connection request originated in the AF_INET6 domain, *address* points to a **sockaddr_in6** structure. The **sockaddr_in** and **sockaddr_in6** structures are defined in **netinet/in.h**. , The *address_len* parameter is used only if the address is not NULL. Before calling accept(), you must set the integer pointed to by *address_len* to the size of the buffer, in bytes, pointed to by *address*. On successful return, the integer pointed to by *address_len* contains the actual number of bytes copied into the buffer. If the buffer is not large enough to hold the address, up to *address_len* bytes of the requester's address are copied. If the actual length of the address is greater than the length of the supplied **sockaddr**, the stored address is truncated. The **sa_len** member of the store structure contains the length of the untruncated address.

Notes:

1. This call is used only with SOCK_STREAM sockets. There is no way to screen requesters without calling accept(). The application cannot tell the system the requesters from which it will accept connections. However, the caller can choose to close a connection immediately after discovering the identity of the requester.
2. The accept() function has a dependency on the level of the Enhanced ASCII Extensions. See “Enhanced ASCII support” on page 2113 for details.

A socket can be checked for incoming connection requests using the select() call.

Special behavior for C++: To use this function with C++, you must use the _XOPEN_SOURCE_EXTENDED 1 feature test macro.

Returned value

If successful, accept() returns a nonnegative socket descriptor.

If unsuccessful, accept() returns -1 and sets errno to one of the following values:

Error Code
Description

EAGAIN
If during an accept call that changes identity, the UID of the new identity is already at MAXPROCUID, the accept call fails.

EBADF
The *socket* parameter is not within the acceptable range for a socket descriptor.

EFAULT
Using *address* and *address_len* would result in an attempt to copy the address into a portion of the caller's address space into which information cannot be written.

EINTR
A signal interrupted the accept() call before any connections were available.

EINVAL
listen() was not called for socket descriptor *socket*.

EIO There has been a network or transport failure.

EMFILE
An attempt was made to open more than the maximum number of file descriptors allowed for this process.

EMVSERR
Two consecutive accept calls that cause an identity change are not allowed. The original identity must be restored (close() the socket that caused the identity change) before any further accepts are allowed to change the identity

ENFILE
The maximum number of file descriptors in the system are already open.

ENOBUFS
Insufficient buffer space is available to create the new socket.

ENOTSOCK
The *socket* parameter does not refer to a valid socket descriptor.

EOPNOTSUPP
The socket type of the specified socket does not support accepting connections.

EWOULDBLOCK
The socket descriptor *socket* is in nonblocking mode, and no connections are in the queue.

Example

The following are two examples of the accept() call. In the first, the caller wishes to have the requester's address returned. In the second, the caller does not wish to have the requester's address returned.

```
int clientsocket;
int s;
struct sockaddr clientaddress;
int address_len;
int accept(int s, struct sockaddr *addr, int *address_len);
/* socket(), bind(), and listen()
have been called */
```

```
/* EXAMPLE 1: I want the address now */
address_len = sizeof(clientaddress);
clientsocket = accept(s, &clientaddress, &address_len);
/* EXAMPLE 2: I can get the address later using getpeername() */
clientsocket = accept(s, (struct sockaddr *) 0,
(int *) 0);
```

Related information

- “sys/socket.h” on page 74
- “sys/types.h” on page 75
- “bind() — Bind a name to a socket” on page 210
- “connect() — Connect a socket” on page 308
- “getpeername() — Get the name of the peer connected to a socket” on page 748
- “listen() — Prepare the server for incoming client requests” on page 972
- “socket() — Create a socket” on page 1688

accept_and_recv() — Accept connection and receive first message

Standards

Standards / Extensions	C or C++	Dependencies
z/OS UNIX	both	

Format

X/Open:

```
#define _OPEN_SYS_SOCK_EXT2
#include <sys/socket.h>

int accept_and_recv(int socket, int *accept_socket,
                    struct sockaddr *remote_address,
                    socklen_t *remote_address_len,
                    struct sockaddr *local_address,
                    socklen_t *local_address_len,
                    void *buffer, size_t buffer_len);
```

General description

The accept_and_recv() function extracts the first connection on the queue of pending connections. It either reuses the specified socket (if **accept_socket* is not -1) or creates a new socket with the same socket type, protocol, and address family as the listening socket (if **accept_socket* is -1). It then returns the first block of data sent by the peer and returns the local and remote socket addresses associated with the connection.

The function takes the following arguments:

Parameter
Description

socket Specifies a socket that was created with socket(), has been bound to an address with bind(), and has issued a successful call to listen().

accept_socket
Pointer to an `int` which specifies the socket on which to accept the incoming connection. The socket must not be bound or connected. Use of this parameter lets the application reuse the accepting socket. It is possible that the system may choose to reuse a different socket than the one the

application specified by this argument. In this case, the system will set **accept_socket* to the socket actually reused.

A value of -1 for **accept_socket* indicates that the accepting socket should be assigned by the system and returned to the application in this parameter. It is recommended that a value of -1 be used on the first call to accept_and_recv(). For more details, see "Usage notes" on page 110.

remote_address
: Either a NULL pointer or a pointer to a `sockaddr` structure where the address of the connecting socket will be returned.

remote_address_len
: Points to a `socklen_t`. On input, this specifies the length of the supplied `sockaddr` structure. On output, this contains the length of the stored address.

local_address
: Either a NULL pointer or a pointer to a `sockaddr` structure where the address of the local socket will be returned.

local_address_len
: Points to a `socklen_t`. On input, this specifies the length of the supplied `sockaddr` structure. On output, this contains the length of the stored address.

buffer
: Either a NULL pointer, or a pointer to a buffer where the message should be stored. If this is a NULL pointer, no receive is performed, and accept_and_recv() completes when the incoming connection is received.

buffer_len
: Specifies the length in bytes of the buffer pointed to by the *buffer* argument.

If **accept_socket* is not -1, the incoming connection will be accepted on the socket specified by **accept_socket*. The system may choose to reuse a different socket. If it does, the system will change **accept_socket* to reflect the socket actually used.

If *remote_address* is not a NULL pointer, the address of the peer for the accepted connection is stored in the `sockaddr` structure pointed to by *remote_address*, and the length of this address is stored in the object pointed to by *remote_address_len*. If the actual length of the address is greater than the length of the supplied socket address structure, the stored address will be truncated.

If *local_address* is not a NULL pointer, the address of the local socket associated with this connection is stored in the `sockaddr` structure pointed to by *local_address*, and the length of this address is stored in the object pointed to by *local_address_len*. If the actual length of the address is greater than the length of the supplied socket address structure, the stored address will be truncated.

Nonblocking mode is not supported for this function. If `O_NONBLOCK` is set on the socket file descriptor, the function will return with -1 and errno will be set to `EOPNOTSUPP`.

If the listen queue is empty of connection requests, accept_and_recv() will not return until an incoming connection is received.

If *buffer* is not NULL, accept_and_recv will not return until the first block of data on the connection has been received, otherwise accept_and_recv() returns 0 after the connection is established.

Note: The accept_and_recv() function has a dependency on the level of the Enhanced ASCII Extensions. See "Enhanced ASCII support" on page 2113 for details.

Usage notes

1. On the first call to accept_and_recv(), it is recommended that the application set the socket pointed to by *accept_socket* to -1. This will cause the system to assign the accepting socket. The application then passes the assigned value into the next call to accept_and_recv() (by setting *accept_socket* = *socket_ptr*).

   To take full advantage of the performance improvements offered by the accept_and_recv() function, a process/thread model different from the one where a parent accepts in a loop and spins off child process threads is needed. The parent/process thread is eliminated. Multiple worker processes/threads are created, and each worker process/thread then executes the accept_and_recv() function in a loop. The performance benefits of accept_and_recv() include fewer buffer copies, recycled sockets, and optimal scheduling.

Returned value

If successful, accept_and_recv() returns the number of bytes (zero or more) stored in the buffer pointed to by the *buffer* argument. Zero can be returned when buffer is NULL or when the client closes the socket without sending any data.

A partial success is achieved with *accept_socket being assigned, a return value of -1 and errno set to one of the following values:

Error Code
  Description

EINTRNODATA
  The accept_and_recv() function was interrupted by a signal that was caught after a valid connection was established, but before the first block of data arrived.

EWOULDBLOCK
  A new connection was established, but the SO_RCVTIMEO timeout value was reached before data was available.

If unsuccessful, accept_and_recv() sets *accepted_socket to -1, returns -1 and sets errno to one of the following values:

Error Code
  Description

EBADF
  One of two errors occurred:
  1. *socket* is not a valid descriptor.
  2. *accept_socket* does not point to a valid descriptor.

ECONNABORTED
  A connection has been aborted.

ECONNRESET
  A connection was forcibly closed by a peer.

EFAULT
The data buffer pointed to by *accept_socket*, *remote_address*, *remote_address_len*, *local_address*, *local_address_len*, or *buffer* was not valid.

EINTR
The accept_and_recv() function was interrupted by a signal that was caught before a valid connection was established.

EINVAL
The *socket* is not accepting connections.

EIO An I/O error occurred.

EISCONN
The *accept_socket* is either bound or connected already.

EMFILE
`OPEN_MAX` descriptors are already open in the calling process.

ENOBUFS
No buffer space is available.

ENOMEM
There was insufficient memory available to complete the operation.

ENOREUSE
Socket reuse is not supported.

ENOSR
There were insufficient STREAMS resources available for the operation to complete.

ENOTSOCK
The *socket* argument does not refer to a socket, or *accept_socket* does not point to a socket.

EOPNOTSUPP
One of errors occurred:

1. The type of the socket does not support accepting connections.
2. `O_NONBLOCK` is set for the socket and nonblocking is not supported for this function.

Related information

- "sys/socket.h" on page 74
- "accept() — Accept a new connection on a socket" on page 105
- "getpeername() — Get the name of the peer connected to a socket" on page 748
- "getsockname() — Get the name of a socket" on page 775
- "read() — Read from a file or socket" on page 1376

access() — Determine whether a file can be accessed

Standards

Standards / Extensions	C or C++	Dependencies
ISO C XPG4 XPG4.2 C99 Single UNIX Specification, Version 3	both	

Format

```
#define _POSIX_SOURCE
#include <unistd.h>

int access(const char *pathname, int how);
```

General description

Determines how an HFS file can be accessed. When checking to see if a process has appropriate permissions, access() looks at the *real* user ID (UID) and group ID (GID), not the effective IDs.

pathname is the name of the file whose accessibility you want to test. The *how* argument indicates the access modes you want to test. The following symbols are defined in the unistd.h header file for use in the *how* argument:

F_OK Tests whether the file exists.

R_OK Tests whether the file can be accessed for reading.

W_OK
Tests whether the file can be accessed for writing.

X_OK Tests whether the file can be accessed for execution.

You can take the bitwise inclusive-OR of any or all of the last three symbols to test several access modes at once. If you are using F_OK to test for the file's existence, you cannot use OR with any of the other symbols.

Returned value

If the specified access is permitted, access() returns 0.

If the given file cannot be accessed in the specified way, access() returns -1 and sets errno to one of the following values:

Error Code
Description

EACCES
The process does not have appropriate permissions to access the file in the specified way, or does not have search permission on some component of the *pathname* prefix.

EINVAL
The value of *how* is incorrect.

ELOOP
A loop exists in the symbolic links.

ENAMETOOLONG
pathname is longer than **PATH_MAX** characters. The **PATH_MAX** value is determined using pathconf().

ENOENT
There is no file named *pathname*, or the *pathname* argument is an empty string.

ENOTDIR
Some component of the *pathname* prefix is not a directory.

EROFS
The argument *how* has specified write access for a file on a read-only file system.

Returned value for POSIX C

The following errno values behave differently when a program is running with POSIX(ON):

Error Code
Description

ELOOP
A loop exists in the symbolic links. This error is issued if the number of symbolic links detected in the resolution is greater than POSIX_SYMLOOP (a value defined in the limits.h header file).

ENAMETOOLONG
pathname is longer than **PATH_MAX** characters, or some component of *pathname* is longer than **NAME_MAX**, when _POSIX_NO_TRUNC (defined in the unistd.h header file) is in effect. The **PATH_MAX** and **NAME_MAX** values are determined using pathconf().

Example

CELEBA03

```
/* CELEBA03

   The following example determines how a file is accessed.

 */
#define _POSIX_SOURCE
#include <stdio.h>
#undef _POSIX_SOURCE
#include <unistd.h>

main() {
  char path[]="/";

  if (access(path, F_OK) != 0)
    printf("'%s' does not exist!\n", path);
  else {
    if (access(path, R_OK) == 0)
      printf("You have read access to '%s'\n", path);
    if (access(path, W_OK) == 0)
      printf("You have write access to '%s'\n", path);
    if (access(path, X_OK) == 0)
      printf("You have search access to '%s'\n", path);
  }
}
```

Output: From a non-superuser:

```
You have read access to '/'
You have search access to '/'
```

Related information

- “limits.h” on page 39
- “unistd.h” on page 82
- “chmod() — Change the mode of a file or directory” on page 272
- “stat() — Get file information” on page 1720

acl_create_entry() — Add a new extended ACL entry to the ACL

Standards

Standards / Extensions	C or C++	Dependencies
z/OS UNIX	both	z/OS V1R3

Format

```
#define _OPEN_SYS 1
#include <sys/acl.h>

int acl_create_entry(lacl_t *acl_p, acl_entry_t entry_p, int version);
```

General description

Use access control lists (ACLs) in conjunction with permission bits to control access to files and directories. Currently, ACLs are supported by the HFS, TFS, and zFS file systems. You must know whether your security product supports ACLs and what rules are used when determining file access. See *z/OS UNIX System Services Planning* for details.

The acl_create_entry() function creates a new extended ACL entry in the ACL pointed to by the contents of the pointer argument *acl_p*. The contents of the acl_entry are specified by *entry_p*. ACL working storage is allocated as needed. The first call to acl_get_entry() following the call to acl_create_entry() will obtain the first extended ACL entry in the ACL, as ordered by the system.

The version tells the function the version of ACL entry. See “sys/acl.h” on page 72 for ACL entry mapping.

Returned value

Upon successful completion, the function returns a value of zero.

If any of the following conditions occur, the acl_create_entry() function returns -1 and sets errno to the corresponding value:

Error Code
  Description

EINVAL
  Argument *acl_p* does not point to a pointer to an ACL structure. Argument *entry_d* does not point to a valid extended ACL entry.

ENOMEM
  The ACL working storage requires more memory than is available.

Related information

- “sys/acl.h” on page 72

acl_delete_entry() — Delete an extended ACL entry from the ACL

Standards

Standards / Extensions	C or C++	Dependencies
z/OS UNIX	both	z/OS V1R3

Format

```
#define _OPEN_SYS 1
#include <sys/acl.h>

int acl_delete_entry(lacl_t acl_d, acl_entry_t entry_d);
```

General description

Use access control lists (ACLs) in conjunction with permission bits to control access to files and directories. Currently, ACLs are supported by the HFS, TFS, and zFS file systems. You must know whether your security product supports ACLs and what rules are used when determining file access. See *z/OS UNIX System Services Planning* for details.

The acl_delete_entry() function removes the extended ACL entry indicated by *entry_d* in the ACL pointed to by argument *acl_d*. The first call to acl_get_entry() following the call to acl_delete_entry() will obtain the first extended ACL entry in the ACL, as ordered by the system.

Returned value

Upon successful completion, the function returns a value of zero.

If any of the following conditions occur, the acl_delete_entry() function returns -1 and sets errno to the corresponding value:

Error Code
 Description

EINVAL
 Argument *acl_d* does not point to a pointer to an ACL structure. Argument *entry_d* does not point to a valid extended ACL entry or not within the given ACL structure.

Related information

- "sys/acl.h" on page 72

acl_delete_fd() — Delete an ACL by file descriptor

Standards

Standards / Extensions	C or C++	Dependencies
z/OS UNIX	both	z/OS V1R3

Format

```
#define _OPEN_SYS 1
#include <sys/acl.h>

int     acl_delete_fd (int fd, int type_d);
```

General description

Use access control lists (ACLs) in conjunction with permission bits to control access to files and directories. Currently, ACLs are supported by the HFS, TFS, and zFS file systems. You must know whether your security product supports ACLs and what rules are used when determining file access. See *z/OS UNIX System Services Planning* for details.

The acl_delete_fd() function deletes the *type_d* ACL. That means that all extended ACL entries are deleted for *type_d* ACL. A file/dir subject must match the owner of the directory/file or the subject must have appropriate privileges.

The effective UID of the process must match the owner of the directory/file or the process must have appropriate privileges. If the *type_d* is the directory/file default and the object referred to by *fd* is not a directory, then the function will fail. An attempt to delete an ACL from a file that does not have that ACL is not considered an error.

Upon successful completion, the acl_delete_fd() will delete the type ACL associated with the file referred by argument *fd*. If unsuccessful, the type ACL associated with the file object referred by argument *fd* will not be changed.

Returned value

Upon successful completion, the function returns a value of zero.

If any of the following conditions occur, the acl_delete_fd() function returns -1 and sets errno to the corresponding value:

Error Code
  Description

EBADF
  The *fd* argument is not a valid file descriptor.

EINVAL
  Argument *type_d* is not a valid ACL type.

ENOTDIR
  The type specifies directory/file default ACL and the argument *fd* does not refer to a directory object.

EACCES
  The process does not have appropriate privilege to delete the type ACL.

Related information

- “sys/acl.h” on page 72
- “acl_delete_file() — Delete an ACL by file name”

acl_delete_file() — Delete an ACL by file name

Standards

Standards / Extensions	C or C++	Dependencies
z/OS UNIX	both	z/OS V1R3

Format

```
#define _OPEN_SYS 1
#include <sys/acl.h>

int     acl_delete_file (const char *path_p, int type_d);
```

General description

Use access control lists (ACLs) in conjunction with permission bits to control access to files and directories. Currently, ACLs are supported by the HFS, TFS, and zFS file systems. You must know whether your security product supports ACLs and what rules are used when determining file access. See *z/OS UNIX System Services Planning* for details.

The acl_delete_file() function deletes the *type_d* ACL. That means that all extended ACL entries are deleted for *type_d* ACL. A file/directory always has base ACL entries so they cannot be deleted. The effective UID of the process must match the owner of the directory/file or the process must have appropriate privileges.

If the type_d is the directory/file default and the object referred to by *fd* is not a directory, then the function will fail. An attempt to delete an ACL from a file that does not have that ACL is not considered an error.

Upon successful completion, the acl_delete_file() will delete the type ACL associated with the file referred by argument *path_p*. If unsuccessful, the type ACL associated with the file object referred by argument *path_p* will not be changed.

Returned value

Upon successful completion, the function returns a value of zero.

If any of the following conditions occur, the acl_delete_file() function returns -1 and sets errno to the corresponding value:

Error Code
Description

EACCES
Search permission is denied for a component of the path prefix or the object exists and the subject does not have appropriate access rights.

EINVAL
Argument *type_d* is not a valid ACL type.

ENAMETOOLONG
The length of the path name argument exceeds PATH_MAX, or a path name component is longer than NAME_MAX and {_POSIX_NO_TRUNC} is in effect for that file. For symbolic links, the length of the path name string substituted for a symbolic link exceeds PATH_MAX. PATH_MAX and NAME_MAX values can be determined by using pathconf().

ENOENT
The named object does not exist or the *path_p* argument points to an empty string.

ENOTDIR
The type specified was directory/file default but the argument *path_p* is not a directory or a component of the path prefix is not a directory.

Related information

- “sys/acl.h” on page 72
- “acl_delete_fd() — Delete an ACL by file descriptor” on page 115

acl_first_entry() — Return to beginning of ACL working storage

Standards

Standards / Extensions	C or C++	Dependencies
z/OS UNIX	both	z/OS V1R3

Format

```
#define _OPEN_SYS 1
#include <sys/acl.h>

int     acl_first_entry (lacl_t acl_d);
```

General description

Use access control lists (ACLs) in conjunction with permission bits to control access to files and directories. Currently, ACLs are supported by the HFS, TFS, and zFS file systems. You must know whether your security product supports ACLs and what rules are used when determining file access. See *z/OS UNIX System Services Planning* for details.

A call to acl_first_entry() sets the internal ACL entry offset descriptor for the *acl_d* argument such that a subsequent call to acl_get_entry() using the same *acl_d* argument obtains the first extended ACL entry in the ACL.

Returned value

Upon successful completion, the function returns a value of zero.

If any of the following conditions occur, the acl_init() function returns -1 and sets errno to the corresponding value:

Error Code
 Description

EINVAL
 Argument *acl_d* does not point to an ACL structure.

Related information

- "sys/acl.h" on page 72
- "acl_get_entry() — Get an ACL entry" on page 121

acl_free() — Release memory allocated to an ACL data object

Standards

Standards / Extensions	C or C++	Dependencies
z/OS UNIX	both	z/OS V1R3

Format

```
#define _OPEN_SYS 1
#include <sys/acl.h>

int     acl_free (lacl_t acl_d);
```

General description

Use access control lists (ACLs) in conjunction with permission bits to control access to files and directories. Currently, ACLs are supported by the HFS, TFS, and zFS file systems. You must know whether your security product supports ACLs and what rules are used when determining file access. See *z/OS UNIX System Services Planning* for details.

The acl_free() function frees any releasable memory currently allocated to the ACL data object identified by *acl_d*. Use of the object reference pointed to by *acl_d* after the memory has been released is undefined.

Returned value

Upon successful completion, the function returns a value of zero.

If any of the following conditions occur, the acl_free() function returns -1 and sets errno to the corresponding value:

Error Code
Description

EINVAL
The value of the *acl_d* argument is not valid.

Related information

- “sys/acl.h” on page 72
- “acl_init() — Initialize ACL working storage” on page 125

acl_from_text() — Create an ACL from text

Standards

Standards / Extensions	C or C++	Dependencies
z/OS UNIX	both	z/OS V1R3

Format

```
#define _OPEN_SYS 1
#include <sys/acl.h>

int  acl_from_text (const char *buf_p, short OpType, acl_all_t ptr, char **ret);
```

General description

Use access control lists (ACLs) in conjunction with permission bits to control access to files and directories. Currently, ACLs are supported by the HFS, TFS, and zFS file systems. You must know whether your security product supports ACLs and what rules are used when determining file access. See *z/OS UNIX System Services Planning* for details.

The acl_from_text() function converts the text form of an ACL referred to by *buf_p* into the acl_t form of an ACL. It parses both the extended and base ACL entries.

If successful, the structure that *ptr* points to will be updated with ACL entries for the 3 types of ACLs. The structure members that *ptr* points to must either be NULL or point to a valid *acl_t* structure. If the structure member is not NULL, ACL entries contained in *buf_p* will be merged in. New storage for those structure members may be allocated as needed and in that case passed-in storage will be freed, so structure members may point to a different storage than the one originally supplied.

If ptr is NULL, the acl_from_text will fail. If the *buf_p* has no ACL entries (such as empty string, only empty lines, etc), acl_from_text() will fail with errno set to EINVAL and *ret* will point to the null terminator in *buf_p*.

Working storage is allocated for the individual ACLs structures as needed and need to be freed using acl_free().

If the function is unsuccessful due to error encountered in parsing, ret will contain the address of beginning of extended/base ACL entry where the error was found in *buf_p* and errno will be set to EINVAL. Otherwise *ret* will be NULL.

The text form of the ACL referred to by buf_p may be incomplete or may be a non-valid ACL as defined by acl_valid(). The *buf_p* must be null terminated. The first call to acl_get_entry() following the call to acl_from_text() obtains the first entry in the ACL as ordered by the system.

For OpType =ACL_DELETE, the extended ACL entries will be updated with the flag bit to be removed from the ACL when acl_set_fd() or acl_set_file() is called. The base ACL entries are parsed but not put into the structure that ptr points to since you cannot delete base ACL entries.

The *buf_p* cannot have a mixture of ACL entry delimiters (newline and comma). All ACL entries in the buf must use the same delimiter, either a newline or comma. For a mixture of delimiters, acl_from_text() will fail with errno set to EINVAL and ret parameter will point to the delimiter in error.

acl_all_t :

index 0
access acl

index 1
file default acl

index 2
directory default acl

Valid text input format based on OpType:

tag fdefault, default (access if nothing is specified)

type user, group, other

id uid, gid, username, groupname

perm rwx (or '-' for no permission), octal (0-7) , +/^ (where + is turn on and ^ is turn off)

Pound sign (#) is used to designate a comment. When the input is separated by commas, everything past # is treated as a comment. When the input is separated by a newline, everything after # till the newline is considered a comment. Comments are ignored and are not stored in the buffer.

```
ACL_ADD  tag:type:id:perm // extended ACL entry
   type::perm     // base ACL entry
ACL_MODIFY same as ACL_ADD
ACL_DELETE tag:type:id  // extended ACL entry
```

Note: The extended ACL entries must have the type (group or user) and the id (uid/gid). The base ACL entries do not have a value for the id field. The id field or the lack of one is what distinguishes the base ACL entry from the extended ACL entry. The base ACL entries do not have the tag fields since they only apply to access ACL.

The acl_from_text() allows for trailing ACL entry separator (newline or comma). For relative permission settings, only one of '+' or '^' is allowed per ACL entry. When using relative permissions you must have at least one of r, w, or x. For example: +rw or ^rwx.

Returned value

Upon successful completion, the function returns a zero.

If any of the following conditions occur, the acl_from_text() function returns -1 and sets errno to the corresponding value:

Error Code
Description

EINVAL
Argument *buf_p* cannot be translated into an ACL.

ENOMEM
The ACL working storage requires more memory than is available.

E2BIG The number of base ACL entries exceeded allowable 3.

The ret will contain the address in *buf_p* where the error was found.

Related information

- “sys/acl.h” on page 72
- “acl_free() — Release memory allocated to an ACL data object” on page 118
- “acl_to_text() — Convert an ACL to text” on page 132

acl_get_entry() — Get an ACL entry

Standards

Standards / Extensions	C or C++	Dependencies
z/OS UNIX	both	z/OS V1R3

Format

```
#define _OPEN_SYS 1
#include <sys/acl.h>

int     acl_get_entry (lacl_t acl_d, acl_entry_t *entry_p);
```

General description

Use access control lists (ACLs) in conjunction with permission bits to control access to files and directories. Currently, ACLs are supported by the HFS, TFS, and zFS file systems. You must know whether your security product supports ACLs and what rules are used when determining file access. See *z/OS UNIX System Services Planning* for details.

The acl_get_entry() function obtains a descriptor to the next extended ACL entry of the ACL indicated by argument *acl_d*. Upon successful execution, the acl_get_entry() function returns a descriptor for the extended ACL entry via entry_p. Argument *acl_d* must refer to a valid acl_t structure.

The first call to acl_get_entry() following a call to acl_first_entry(), acl_from_text(), acl_get_fd(), acl_get_file(), acl_set_fd(), acl_set_file(), or acl_valid() obtains the first extended ACL entry in the ACL, as ordered by the system. Subsequent calls to acl_get_entry() obtain successive extended ACL entries, until the last entry is obtained. After the last extended ACL entry has been obtained from the *acl_d* the value NULL is returned via *entry_p*.

To determine if ACL has any base ACL entries, check acl_d->lacl_base, which gives the number of base ACL entries present. Then the process can access the base ACL entries directly in the acl_d. (For example: acl_d->lacl_base_entries[0].acle_type is the type field of the first base ACL entry.)

Returned value

If the function successfully obtains a pointer to the extended ACL entry, the function returns a value of one. If the last extended ACL entry in the ACL has already been returned by a previous call to acl_get_entry() or if ACL has no extended ACL entries, the function returns a value of zero.

If any of the following conditions occur, the acl_get_entry() function returns -1 and sets errno to the corresponding value:

Error Code
Description

EINVAL
Argument *acl_d* does not point to an ACL structure.

Related information

- "sys/acl.h" on page 72
- "acl_init() — Initialize ACL working storage" on page 125
- "acl_get_file() — Get ACL by file name" on page 124
- "acl_first_entry() — Return to beginning of ACL working storage" on page 118

acl_get_fd() — Get ACL by file descriptor

Standards

Standards / Extensions	C or C++	Dependencies
z/OS UNIX	both	z/OS V1R3

Format

```
#define _OPEN_SYS 1
#include <sys/acl.h>

int   acl_get_fd (int fd, acl_type_t type_d, lacl_t acl_d, int *num);
```

General description

Use access control lists (ACLs) in conjunction with permission bits to control access to files and directories. Currently, ACLs are supported by the HFS, TFS, and zFS file systems. You must know whether your security product supports ACLs and what rules are used when determining file access. See *z/OS UNIX System Services Planning* for details.

The acl_get_fd() function retrieves an ACL based on *type_d* argument for an object associated with the file descriptor *fd*. The ACL is retrieved into the supplied working storage pointed to by *acl_d*. For the *type_d* = ACL_ACCESS, acl_get_fd() will get both the base ACL entries and extended ACL entries. (The base ACL entries only apply to the ACL_ACCESS ACL.)

The working storage should be allocated using the acl_init() function. If the buffer is not big enough, the acl_get_fd() will fail with errno=E2BIG and *num* will be filled with the number of ACLs in the ACL pointed to by fd. The user can get a bigger acl_t structure buffer using the num value and reissue the acl_get_fd().

If the object associated with the file descriptor does not have the specified ACL, then an ACL containing zero ACL entries will be returned. If the argument *fd* refers to an object other than a directory and the value of *type_d* is a directory/file default, then the function will fail.

The first call to acl_get_entry() following the call to acl_get_fd() obtains the first extended ACL entry in the ACL as ordered by the system.

The result of acl_get_fd() can be used to set that same ACL using acl_set_fd() or acl_set_file() using OpType = ACL_ADD.

Returned value

Upon successful completion, the function returns zero.

If any of the following conditions occur, the acl_get_fd() function returns a value of NULL and sets errno to the corresponding value:

Error Code
Description

EACCES
The required access to the file referred to by *fd* is denied.

EBADF
The fd argument is not a valid file descriptor.

EINVAL
Argument *type_d* is not a valid ACL type. Argument *acl_d* does not point to an ACL structure.

ENOTDIR
The type specified was directory/file default but the argument *fd* does not refer to a directory.

E2BIG
The supplied buffer is too small for all extended ACL entries. *num* value has the number of ACL entries that need to fit in the buffer.

Related information

- “sys/acl.h” on page 72
- “acl_free() — Release memory allocated to an ACL data object” on page 118
- “acl_get_entry() — Get an ACL entry” on page 121
- “acl_set_fd() — Set an ACL by file descriptor” on page 126

acl_get_file() — Get ACL by file name

Standards

Standards / Extensions	C or C++	Dependencies
z/OS UNIX	both	z/OS V1R3

Format

```
#define _OPEN_SYS 1
#include <sys/acl.h>

int  acl_get_file (const char *path_p, acl_type_t type_d, lacl_t acl_d, int *num);
```

General description

Use access control lists (ACLs) in conjunction with permission bits to control access to files and directories. Currently, ACLs are supported by the HFS, TFS, and zFS file systems. You must know whether your security product supports ACLs and what rules are used when determining file access. See *z/OS UNIX System Services Planning* for details.

The acl_get_file() function retrieves an ACL based on *type_d* argument for an object associated with the object via file name. The ACL is retrieved into the specified working storage pointed to by *acl_d*. For the *type_d* = ACL_ACCESS, acl_get_file() will get both the base ACL entries and extended ACL entries. The base ACL entries only apply to the ACL_ACCESS ACL.

The working storage should be allocated using the acl_init() function. If the buffer is not big enough, the acl_get_fd() will fail with errno=E2BIG and *num* will be filled with the number of ACLs in the ACL pointed to by *fd*. The user can get a bigger acl_t structure buffer using the num value and reissue the acl_get_fd().

If the object associated with the file descriptor does not have the specified ACL, then an ACL containing zero ACL entries will be returned. If the argument *fd* refers to an object other than a directory and the value of *type_d* is a directory/file default, then the function will fail.

The first call to acl_get_entry() following the call to acl_get_fd() obtains the first extended ACL entry in the ACL as ordered by the system. The result of acl_get_fd() can be used to set that same ACL using acl_set_fd() or acl_set_file() using OpType = ACL_ADD.

Returned value

Upon successful completion, the function returns zero.

If any of the following conditions occur, the acl_get_file() function returns a value of NULL and sets errno to the corresponding value:

Error Code
Description

EACCES
Search permission is denied for a component of the path prefix or the object exists and the subject does not have appropriate access rights.

EINVAL
Argument *type_d* is not a valid ACL type. Argument *acl_d* does not point to an ACL structure.

ENAMETOOLONG
The length of the path name argument exceeds PATH_MAX, or a path name component is longer than NAME_MAX and {_POSIX_NO_TRUNC} is in effect for that file. For symbolic links, the length of the path name string substituted for a symbolic link exceeds PATH_MAX. PATH_MAX and NAME_MAX values can be determined by using pathconf().

ENOENT
The named object does not exist or the *path_p* argument points to an empty string.

ENOTDIR
The type specified was directory/file default but the argument *path_p* is not a directory or a component of the path prefix is not a directory.

E2BIG The supplied buffer is too small for all extended ACL entries. Num value has the number of ACL entries that need to fit in the buffer.

Related information

- "sys/acl.h" on page 72
- "acl_free() — Release memory allocated to an ACL data object" on page 118
- "acl_set_file() — Set an ACL by file name" on page 128
- "acl_get_entry() — Get an ACL entry" on page 121

acl_init() — Initialize ACL working storage

Standards

Standards / Extensions	C or C++	Dependencies
z/OS UNIX	both	z/OS V1R3

Format

```
#define _OPEN_SYS 1
#include <sys/acl.h>

lacl_t   acl_init (int count);
```

General description

Use access control lists (ACLs) in conjunction with permission bits to control access to files and directories. Currently, ACLs are supported by the HFS, TFS, and zFS file systems. You must know whether your security product supports ACLs and what rules are used when determining file access. See *z/OS UNIX System Services Planning* for details.

The acl_init() function allocates and initializes working storage for an ACL of at least *count* extended ACL entries. A pointer to the working storage is returned. The working storage allocated to contain the ACL must be freed by a call to acl_free(). The working storage contains an ACL with no ACL entries. The count must be greater than 0.

The acl_init() function initializes fields in the *lacl_t* it returns. When those fields are destroyed (for example, using memset or overwriting storage), the results are unpredictable. To re-use the buffer, acl_entry_delete() all extended ACL entries and set lacl_base = 0 or acl_free() the existing buffer and acl_init() for a new one.

Returned value

Upon successful completion, the function returns a pointer to the working storage.

If any of the following conditions occur, the acl_init() function returns NULL and sets errno to the corresponding value:

Error Code
 Description

ENOMEM
 The *lacl_t* to be returned requires more memory than is available.

EINVAL
 The count is less than or equal to zero.

Related information

- “sys/acl.h” on page 72
- “acl_free() — Release memory allocated to an ACL data object” on page 118

acl_set_fd() — Set an ACL by file descriptor

Standards

Standards / Extensions	C or C++	Dependencies
z/OS UNIX	both	z/OS V1R3

Format

```
#define _OPEN_SYS 1
#include <sys/acl.h>

int     acl_set_fd (int fd, acl_type_t type_d, lacl_t acl_d, short OpType,
  acl_entry_t *entry_p);
```

General description

Use access control lists (ACLs) in conjunction with permission bits to control access to files and directories. Currently, ACLs are supported by the HFS, TFS, and zFS file systems. You must know whether your security product supports ACLs and what rules are used when determining file access. See *z/OS UNIX System Services Planning* for details.

The acl_set_fd() function associates the type_d ACL with the object referred to by *fd*. The effective UID of the subject must match the owner of the object or the subject must have appropriate privileges.

If the *type_d* is the directory/file default and the object referred to by *fd* is not a directory, then the function will fail.

The acl_set_fd() function will succeed only if the ACL is valid as defined by the acl_valid() function.

Upon successful completion, acl_set_fd() will set the ACL of the object. For type_d = ACL_ACCESS, acl_set_fd() will also set the base ACL entries. The base ACL entries only apply to ACl_ACCESS ACL type, so for any other type the base ACL entries are ignored.

The *OpType* determines whether the ACL is updated or replaced.

OpType
Action

ACL_ADD
replace the whole ACL with the given extended and base ACL entries

ACL_MODIFY
update the ACL with the given extended and/or base ACL entries (if individual extended ACL entries are marked for deletion, than ACL_MODIFY removes them)

ACL_DELETE
delete from the ACL the specified extended ACL entries; marks the individual extended ACL entries for deletion (cannot delete base ACL entries)

If *Optype* is ACL_MODIFY, the setting will modify the existing extended ACL entries and add new ones if they do not exist. Both ACL entry's mask and value are used to determine ACL entry's permission to set.

If *OpType* is ACL_ADD, the existing ACL is replaced by the new one. Only extended ACL entry's value is used to determine permissions to set. The object's previous ACL will no longer be in effect. If the object had no ACL, a new one is added for both ACL_MODIFY and ACL_ADD.

Similarly, for OpType = ACL_ADD, base ACL entries are replaced with the new values specified (mask field is ignored). All three base ACL entries (ACL_USER, ACL_GROUP, and ACL_OTHER) must be specified. For OpType = ACL_MODIFY, the base ACL entries are modified with the specified values (both mask and value fields are used).

For Optype = ACL_MODIFY only the base ACL entries to be changed need to be specified. The Optype = ACL_DELETE does not apply to base ACL entries since they cannot be removed. Every file always has base ACL entries.

If the acl_set_fd() is unsuccessful, the ACL of the object referred to by argument *fd* is not changed.

The ordering of entries within ACL referred to by *acl_d* may be changed. The first call to acl_get_entry() following the call to acl_set_fd() obtains the first extended ACL entry in the ACL as ordered by the system.

Returned value

Upon successful completion, the function returns a value of zero.

If any of the following conditions occur, the acl_set_file() function returns a value of -1 and sets errno to the corresponding value:

Error Code
Description

EACCES
Search permission is denied for a component of the path prefix or the object exists and the subject does not have appropriate access rights.

E2BIG The ACL has more extended ACL entries than is allowed.

ENOENT
The named object does not exist or the *path_p* argument points to an empty string.

EINVAL
Argument *acl_d* does not point to a valid ACL structure.

ENOSPC
The directory or file system that would contain the new ACL cannot be extended or the file system is out of space.

ENOTDIR
The *type_d* specified was directory/file default but the argument *path_p* does not refer to a directory.

ENAMETOOLONG
The length of the path name argument exceeds PATH_MAX, or a path name component is longer than NAME_MAX and {_POSIX_NO_TRUNC} is in effect for that file. For symbolic links, the length of the path name string substituted for a symbolic link exceeds PATH_MAX. PATH_MAX and NAME_MAX values can be determined by using pathconf().

EMVSERR
Other internal RACF® error (more information in errno2)

The function will return -2 and set errno to EINVAL if the base ACL entry is not unique or is not a valid type or for ACL_ADD, there are less than 3 base ACL entries. The *entry_p* will be NULL.

The function will return -3 and set errno to EINVAL if the extended ACL entry is not unique or is not a valid type. The *entry_p*, if not NULL, will point to the extended ACL entry in error.

Related information

- "sys/acl.h" on page 72
- "acl_get_fd() — Get ACL by file descriptor" on page 122

acl_set_file() — Set an ACL by file name

Standards

Standards / Extensions	C or C++	Dependencies
z/OS UNIX	both	z/OS V1R3

Format

```
#define _OPEN_SYS 1
#include <sys/acl.h>

int     acl_set_file (char *path_p, acl_type_t type_d, lacl_t acl_d, short OpType,
  acl_entry_t *entry_p);
```

General description

Use access control lists (ACLs) in conjunction with permission bits to control access to files and directories. Currently, ACLs are supported by the HFS, TFS, and zFS file systems. You must know whether your security product supports ACLs and what rules are used when determining file access. See *z/OS UNIX System Services Planning* for details.

The acl_set_file() function associates the type_d ACL with the object referred to by file name *path_p*. The effective UID of the subject must match the owner of the object or the subject must have appropriate privileges.

If the type_d is the directory/file default and the object referred to by file name *path_p* is not a directory, then the function will fail.

The acl_set_file() function will succeed only if the ACL is valid as defined by the acl_valid() function.

Upon successful completion, acl_set_file() will set the ACL of the object. For type_d = ACL_ACCESS, acl_set_file() will also set the base ACL entries. The base ACL entries only apply to ACl_ACCESS ACL type, so for any other type the base ACL entries are ignored.

The *OpType* determines whether the ACL is updated or replaced.

OpType
: **Action**

ACL_ADD
: replace the whole ACL with the given extended and base ACL entries

ACL_MODIFY
: update the ACL with the given extended and/or base ACL entries (if individual extended ACL entries are marked for deletion, then ACL_MODIFY removes them)

ACL_DELETE
: delete from the ACL the specified extended ACL entries; marks the individual extended ACL entries for deletion (cannot delete base ACL entries)

If *Optype* is ACL_MODIFY, the setting will modify the existing extended ACL entries and add new ones if they do not exist. Both ACL entry's mask and value are used to determine ACL entry's permission to set.

If *OpType* is ACL_ADD, the existing ACL is replaced by the new one. Only extended ACL entry's value is used to determine permissions to set. The object's previous ACL will no longer be in effect. If the object had no ACL, a new one is added for both ACL_MODIFY and ACL_ADD.

Similarly, for OpType = ACL_ADD, base ACL entries are replaced with the new values specified (mask field is ignored). All three base ACL entries (ACL_USER, ACL_GROUP, and ACL_OTHER) must be specified. For OpType = ACL_MODIFY, the base ACL entries are modified with the specified values (both mask and value fields are used).

For Optype = ACL_MODIFY only the base ACL entries to be changed need to be specified. The Optype = ACL_DELETE does not apply to base ACL entries since they cannot be removed. Every file always has base ACL entries.

If the acl_set_file() is unsuccessful, the ACL of the object referred to by argument *path_p* is not changed.

The ordering of entries within ACL referred to by acl_d may be changed. The first call to acl_get_entry() following the call to acl_set_file() obtains the first extended ACL entry as ordered by the system.

Returned value

Upon successful completion, the function returns a value of zero.

If any of the following conditions occur, the acl_set_file() function will return a value of -1 and set errno to the corresponding value:

Error Code
    Description

EACCES
    Search permission is denied for a component of the path prefix or the object exists and the subject does not have appropriate access rights.

E2BIG The ACL has more extended ACL entries than are allowed.

ENOENT
    The named object does not exist or the *path_p* argument points to an empty string.

EBADF
    The *fd* argument is not a valid file descriptor.

EINVAL
    Argument *acl_d* does not point to a valid ACL structure.

ENOSPC
    The directory or file system that would contain the new ACL cannot be extended or the file system is out of space.

ENOTDIR
    The *type_d* specified was directory/file default but the argument *path_p* does not refer to a directory.

ENAMETOOLONG
    The length of the path name argument exceeds PATH_MAX, or a path name component is longer than NAME_MAX and {_POSIX_NO_TRUNC} is in effect for that file. For symbolic links, the length of the path name string substituted for a symbolic link exceeds PATH_MAX. PATH_MAX and NAME_MAX values can be determined by using pathconf().

EMVSERR
    Other internal RACF error (more information in errno2)

The function will return -2 and set errno to EINVAL if the base ACL entry is not unique or is not a valid type or for ACL_ADD, there are less than 3 base ACL entries. The *entry_p* will be NULL.

The function will return -3 and set errno to EINVAL if the extended ACL entry is not unique or is not a valid type. The *entry_p*, if not NULL, will point to the extended ACL entry in error.

Related information

- “sys/acl.h” on page 72
- “acl_get_fd() — Get ACL by file descriptor” on page 122
- “acl_valid() — Validate an ACL” on page 134

acl_sort() — Sort the extended ACL entries

Standards

Standards / Extensions	C or C++	Dependencies
z/OS UNIX	both	z/OS V1R3

Format

```
#define _OPEN_SYS 1
#include <sys/acl.h>

int acl_sort(lacl_t acl_d);
```

General description

Use access control lists (ACLs) in conjunction with permission bits to control access to files and directories. Currently, ACLs are supported by the HFS, TFS, and zFS file systems. You must know whether your security product supports ACLs and what rules are used when determining file access. See *z/OS UNIX System Services Planning* for details.

The acl_sort() function sorts the extended ACL entries in the following orders:

- ACL_USER lowest to highest uid
- ACL_GROUP lowest to highest gid

Returned value

Upon successful completion, the function returns a value of zero.

If any of the following conditions occur, the acl_sort() returns -1 and sets errno to the corresponding value:

Error Code
Description

EINVAL
Argument *acl_d* does not point to a valid ACL structure.

Related information

- “sys/acl.h” on page 72

acl_to_text() — Convert an ACL to text

Standards

Standards / Extensions	C or C++	Dependencies
z/OS UNIX	both	z/OS V1R3

Format

```
#define _OPEN_SYS 1
#include <sys/acl.h>

char * acl_to_text (const lacl_t acl_d, ssize_t *len_p, acl_type_t type_d,
  char delim);
```

General description

Use access control lists (ACLs) in conjunction with permission bits to control access to files and directories. Currently, ACLs are supported by the HFS, TFS, and zFS file systems. You must know whether your security product supports ACLs and what rules are used when determining file access. See *z/OS UNIX System Services Planning* for details.

The acl_to_text() function translates the extended ACL entries in an ACL pointed to by argument *acl_d* into a NULL terminated character string. This function allocates any memory necessary to contain the string and returns a pointer to the string. The memory allocated to contain the string must be freed. If the pointer *len_p* is not NULL, then the function returns the full length of the string (not including the NULL terminator) in the location pointed to by *len_p*. The *delim* parameter determines the delimiter used to separate the ACL entries (usually newline or comma).

The mask field in the base and extended ACL entry is ignored and only the ACL entry value field is used to display the ACL entry permissions.

For *acl_d* with no extended ACL entries, acl_to_text() returns NULL. When acl_to_text() cannot convert uid/gid to username/groupname, it leaves the uid/gid in the string.

The format of the text string:

```
<acl_entry>delim<acl_entry> ... <acl_entry>
```

where acl_entry may be:

base_acl_tag::permissions
user:user_name:permissions
group:group_name:permissions
default:user_name:permissions

base_acl_entry:
user, group, or other

permissions:
rwx (with '-' for no permission)

default:
fdefault - file default
default - directory default

Returned value

Upon successful completion, the function returns a pointer to the text form of an ACL.

If any of the following conditions occur, the acl_to_text() returns NULL and sets errno to the corresponding value:

Error Code
Description

EINVAL
Argument *acl_d* does not point to a valid ACL structure. The ACL referenced by acl_d contains one or more improperly formed ACL entries, or for some other reason cannot be translated into the string form of ACL.

ENOMEM
The character string to be returned requires more memory than is available.

Related information

- "sys/acl.h" on page 72
- "acl_free() — Release memory allocated to an ACL data object" on page 118
- "acl_from_text() — Create an ACL from text" on page 119

acl_update_entry() — Update the extended ACL entry

Standards

Standards / Extensions	C or C++	Dependencies
z/OS UNIX	both	z/OS V1R3

Format

```
#define _OPEN_SYS 1
#include <sys/acl.h>

int acl_update_entry(lacl_t acl_d, acl_entry_t entry_s, acl_entry_t entry_d
   int version);
```

General description

Use access control lists (ACLs) in conjunction with permission bits to control access to files and directories. Currently, ACLs are supported by the HFS, TFS, and zFS file systems. You must know whether your security product supports ACLs and what rules are used when determining file access. See *z/OS UNIX System Services Planning* for details.

The acl_update_entry() function updates the extended ACL entry *entry_s* with the values from *entry_d*. The version tells the function the version of ACL entry. See "sys/acl.h" on page 72 for ACL entry mapping.

Returned value

Upon successful completion, the function returns a value of zero.

If any of the following conditions occur, the acl_create_entry() function returns -1 and sets errno to the corresponding value:

Error Code
Description

EINVAL
Argument *entry_s* or *entry_d* do not point to a valid extended ACL entry. Argument *acl_d* does not point to a valid ACL structure.

Related information

- "sys/acl.h" on page 72

acl_valid() — Validate an ACL

Standards

Standards / Extensions	C or C++	Dependencies
z/OS UNIX	both	z/OS V1R3

Format

```
#define _OPEN_SYS 1
#include <sys/acl.h>

int     acl_valid (lacl_t acl_d, acl_entry_t *entry_p);
```

General description

Use access control lists (ACLs) in conjunction with permission bits to control access to files and directories. Currently, ACLs are supported by the HFS, TFS, and zFS file systems. You must know whether your security product supports ACLs and what rules are used when determining file access. See *z/OS UNIX System Services Planning* for details.

The acl_valid() function checks the access ACL, file default ACL, or directory default ACL referred to by the argument *acl_d* for validity.

The ACL_USER, ACL_GROUP, and ACL_OTHER can only exist once in base ACL entries. The ACL_OTHER only applies to base ACL entries.

The tag type (user, group) must contain valid values for the extended ACL entries. The qualifier field (uid, gid) must be unique among all extended ACL entries of the same ACL except for the extended ACL entries that are mapped for deletion (see ACL entry mapping in "sys/acl.h" on page 72 for more information). The ordering of base and/or extended ACL entries within ACL referred by the *acl_d* may be changed.

The first call to acl_get_entry() following the call to acl_valid() obtains the first extended ACL entry in the ACL as ordered by the system.

Returned value

Upon successful completion, the function returns a value of zero.

If any of the following conditions occur, the location referred to by *entry_p* will be undefined and the acl_valid() function returns -1 and sets errno to the corresponding value:

Error Code
Description

EINVAL
Argument *acl_d* does not point to an ACL structure.

If any of the following conditions occur, the acl_valid() function will set the location referred to by entry_p to one of the ACL entries in error, return -2 and set errno to the appropriate value.

Error Code
Description

EINVAL
The ACL contains extended ACL entries that are not unique or is not a valid ACL entry type.

If any of the following conditions occur, the acl_valid() function will return -3 and set errno to the appropriate value.

Error Code
Description

EINVAL
The ACL contains base ACL entries that are not unique or is not a valid ACL entry type. Only one base ACL entry of the same tag type (ACL_USER, ACL_GROUP, ACL_OTHER) may exist.

Related information

- "sys/acl.h" on page 72
- "acl_init() — Initialize ACL working storage" on page 125
- "acl_get_fd() — Get ACL by file descriptor" on page 122
- "acl_get_file() — Get ACL by file name" on page 124
- "acl_set_fd() — Set an ACL by file descriptor" on page 126
- "acl_set_file() — Set an ACL by file name" on page 128

acos(), acosf(), acosl() — Calculate arccosine

Standards

Standards / Extensions	C or C++	Dependencies
ISO C POSIX.1 XPG4 XPG4.2 ISO/ANSI C++ C99 Single UNIX Specification, Version 3 C++ TR1 C99	both	

Format

```
#include <math.h>

double acos(double x);
float acos(float x);                  /* C++ only */
long double acos(long double x);      /* C++ only */
float acosf(float x);
long double acosl(long double x);
```

General description

Calculates the arccosine of *x*, expressed in radians, in the range 0 to pi.

The value of *x* must be between -1 and 1 inclusive.

Note: These functions work in both IEEE Binary Floating-Point and hexadecimal floating-point formats. See “IEEE binary floating-point” on page 94 for more information about IEEE Binary Floating-Point.

Returned value

Special behavior for C/370: If *x* is less than -1 or greater than 1, the function sets errno to EDOM and returns 0. If the correct value would cause underflow, zero is returned and the value ERANGE is stored in errno.

Special behavior for XPG4.2: If successful, the function returns the arccosine of *x*, in the range [0,pi] radians.

If the value of *x* is not in the range [-1,1], the function returns 0.0 and sets errno to the following value. No other errors will occur.

Error Code
Description

EDOM
The value *x* is not in the range [-1,1].

Special behavior for IEEE: If successful, the function returns the arccosine of the argument *x*.

If *x* is less than -1 or greater than 1, the function sets errno to EDOM and returns NaNQ (Not a Number Quiet). No other errors will occur.

Example

CELEBA04

```
/* CELEBA04

   This example prompts for a value for x.
   It prints an error message if x is greater than 1 or
   less than -1; otherwise, it assigns the arccosine of
   x to y.

 */
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#define MAX  1.0
#define MIN -1.0
```

```
int main(void)
{
  double x, y;

  printf( "Enter x\n" );
  scanf( "%lf", &x );

                  /* Output error if not in range */
  if ( x > MAX )
    printf( "Error: %f too large for acos\n", x );
  else if ( x < MIN )
    printf( "Error: %f too small for acos\n", x );
  else {
    y = acos( x );
    printf( "acos( %f ) = %f\n", x, y );
  }
}
```

Output

Expected result if 0.4 is entered:

```
Enter x
acos( 0.400000 ) = 1.159279
```

Related information

- "math.h" on page 44
- "acosh(), acoshf(), acoshl() — Calculate hyperbolic arccosine" on page 139
- "asin(), asinf(), asinl() — Calculate arcsine" on page 185
- "asinh(), asinhf(), asinhl() — Calculate hyperbolic arcsine" on page 188
- "atan(), atanf(), atanl(), atan2(), atan2f(), atan2l() — Calculate arctangent" on page 192
- "atanh(), atanhf(), atanhl() — Calculate hyperbolic arctangent" on page 195
- "cos(), cosf(), cosl() — Calculate cosine" on page 328
- "cosh(), coshf(), coshl() — Calculate hyperbolic cosine" on page 331
- "sin(), sinf(), sinl() — Calculate sine" on page 1672
- "sinh(), sinhf(), sinhl() — Calculate hyperbolic sine" on page 1674
- "tan(), tanf(), tanl() — Calculate tangent" on page 1815
- "tanh(), tanhf(), tanhl() — Calculate hyperbolic tangent" on page 1818

acosd32(), acosd64(), acosd128() - Calculate arccosine

Standards

Standards / Extensions	C or C++	Dependencies
C/C++ DFP	both	z/OS V1.10

Format

```
#define __STDC_WANT_DEC_FP__
#include <math.h>

_Decimal32  acosd32(_Decimal32 x);
_Decimal64  acosd64(_Decimal64 x);
_Decimal128 acosd128(_Decimal128 x);
_Decimal32  acos(_Decimal32 x);      /* C++ only */
_Decimal64  acos(_Decimal64 x);      /* C++ only */
_Decimal128 acos(_Decimal128 x);     /* C++ only */
```

General description

Calculates the arccosine of *x*, expressed in radians, in the range 0 to pi.

The value of *x* must be between -1 and 1 inclusive.

These functions work in IEEE decimal floating-point format. See "IEEE decimal floating-point" on page 95 for more information.

Note: To use IEEE decimal floating-point, the hardware must have the Decimal Floating-Point Facility installed.

Returned value

If successful, the function returns the arccosine of the argument x.

If *x* is less than -1 or greater than 1, the function sets errno to EDOM and returns NaNQ. No other errors will occur.

Example

CELEBA11

```
/* CELEBA11

   The example illustrates the acosd32() function.

   This example prompts for a value for x.
   It prints an error message if x is greater than 1 or
   less than -1; otherwise, it assigns the arccosine of
   x to y.

*/

#define __STDC_WANT_DEC_FP__
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#define MAX  1.0DF
#define MIN -1.0DF

int main(void)
{
   _Decimal32 x, y;

   printf( "Enter x\n" );
   scanf( "%Hf", &x );

   /* Output error if not in range */
   if ( x > MAX )
      printf( "Error: %f too large for acosd32\n", x );
   else if ( x < MIN )
      printf( "Error: %f too small for acosd32\n", x );
   else {
      y = acosd32( x );
      printf( "acosd32( %Hf ) = %Hf\n", x, y );
   }
}
```

Related information

- "math.h" on page 44

- “acoshd32(), acoshd64(), acoshd128() - Calculate hyperbolic arccosine” on page 140
- “asind32(), asind64(), asind128() - Calculate arcsine” on page 187
- “asinhd32(), asinhd64(), asinhd128() - Calculate hyperbolic arcsine” on page 189
- “atand32(), atand64(), atand128(), atan2d32(), atan2d64(), atan2d128() - Calculate arctangent” on page 194
- “atanhd32(), atanhd64(), atanhd128() - Calculate hyperbolic arctangent” on page 197
- “cosd32(), cosd64(), cosd128() — Calculate cosine” on page 329
- “coshd32(), coshd64(), coshd128() - Calculate hyperbolic cosine” on page 332
- “sind32(), sind64(), sind128() — Calculate sine” on page 1673
- “sinhd32(), sinhd64(), sinhd128() - Calculate hyperbolic sine” on page 1676
- “tand32(), tand64(), tand128() - Calculate tangent” on page 1817
- “tanhd32(), tanhd64(), tanhd128() - Calculate hyperbolic tangent” on page 1820

acosh(), acoshf(), acoshl() — Calculate hyperbolic arccosine

Standards

Standards / Extensions	C or C++	Dependencies
XPG4.2 C99 Single UNIX Specification, Version 3 C++ TR1 C99	both	

Format

```
#define _XOPEN_SOURCE_EXTENDED 1
#include <math.h>

double acosh(double x);
```

C99:

```
#define _ISOC99_SOURCE
#include <math.h>

float acoshf(float x);
long double acoshl(long double x);
```

C++ TR1 C99:

```
#define _TR1_C99
#include <math.h>
float acosh(float x);
long double acosh(long double x);
```

General description

The acosh functions compute the (nonnegative) arc hyperbolic cosine of *x*. A domain error occurs for arguments less than 1.

Note: The following table shows the viable formats for these functions. See “IEEE binary floating-point” on page 94 for more information about IEEE Binary Floating-Point.

Function	Hex	IEEE
acosh	X	X

Function	Hex	IEEE
acoshf	X	X
acoshl	X	X

Returned value

If successful, acosh() returns the hyperbolic arccosine of its argument *x*.

If the value of *x* is less than 1.0, then the function returns 0.0 and sets errno to EDOM.

Special behavior for IEEE: If successful, the function returns the hyperbolic arccosine of its argument *x*.

If *x* is less than 1.0, the function sets errno to EDOM and returns NaNQ.

Related information

- "math.h" on page 44
- "acos(), acosf(), acosl() — Calculate arccosine" on page 135
- "asin(), asinf(), asinl() — Calculate arcsine" on page 185
- "asinh(), asinhf(), asinhl() — Calculate hyperbolic arcsine" on page 188
- "atan(), atanf(), atanl(), atan2(), atan2f(), atan2l() — Calculate arctangent" on page 192
- "atanh(), atanhf(), atanhl() — Calculate hyperbolic arctangent" on page 195
- "cos(), cosf(), cosl() — Calculate cosine" on page 328
- "cosh(), coshf(), coshl() — Calculate hyperbolic cosine" on page 331
- "sin(), sinf(), sinl() — Calculate sine" on page 1672
- "sinh(), sinhf(), sinhl() — Calculate hyperbolic sine" on page 1674
- "tan(), tanf(), tanl() — Calculate tangent" on page 1815
- "tanh(), tanhf(), tanhl() — Calculate hyperbolic tangent" on page 1818

acoshd32(), acoshd64(), acoshd128() - Calculate hyperbolic arccosine

Standards

Standards / Extensions	C or C++	Dependencies
C/C++ DFP	both	z/OS V1.10

Format

```
#define __STDC_WANT_DEC_FP__
#include <math.h>

_Decimal32  acoshd32(_Decimal32 x);
_Decimal64  acoshd64(_Decimal64 x);
_Decimal128 acoshd128(_Decimal128 x);
_Decimal32  acosh(_Decimal32 x);      /* C++ only */
_Decimal64  acosh(_Decimal64 x);      /* C++ only */
_Decimal128 acosh(_Decimal128 x);     /* C++ only */
```

General description

The acosh functions compute the (nonnegative) arc hyperbolic cosine of *x*. A domain error occurs for arguments less than 1.

These functions work in IEEE decimal floating-point format. See “IEEE decimal floating-point” on page 95 for more information.

Note: To use IEEE decimal floating-point, the hardware must have the Decimal Floating-Point Facility installed.

Returned value

If successful, the function returns the hyperbolic arccosine of its argument x.

If *x* is less than 1.0, the function sets errno to EDOM and returns NaNQ.

Example

CELEBA12

```
/* CELEBA12

   This example illustrates the acoshd64() function.

*/

#define  __STDC_WANT_DEC_FP__
#include <math.h>
#include <stdio.h>

int main(void)
{
   _Decimal64 x, y;

   x = 100.0DD;
   y = acoshd64(x);

   printf("acoshd64(%Df) = %Df\n", x, y);
}
```

Related information

- “math.h” on page 44
- “acosd32(), acosd64(), acosd128() - Calculate arccosine” on page 137
- “asind32(), asind64(), asind128() - Calculate arcsine” on page 187
- “asinhd32(), asinhd64(), asinhd128() - Calculate hyperbolic arcsine” on page 189
- “atand32(), atand64(), atand128(), atan2d32(), atan2d64(), atan2d128() - Calculate arctangent” on page 194
- “atanhd32(), atanhd64(), atanhd128() - Calculate hyperbolic arctangent” on page 197
- “cosd32(), cosd64(), cosd128() — Calculate cosine” on page 329
- “coshd32(), coshd64(), coshd128() - Calculate hyperbolic cosine” on page 332
- “sind32(), sind64(), sind128() — Calculate sine” on page 1673
- “sinhd32(), sinhd64(), sinhd128() - Calculate hyperbolic sine” on page 1676
- “tand32(), tand64(), tand128() - Calculate tangent” on page 1817
- “tanhd32(), tanhd64(), tanhd128() - Calculate hyperbolic tangent” on page 1820

advance() — Pattern match given a compiled regular expression

Standards

Standards / Extensions	C or C++	Dependencies
XPG4 XPG4.2	both	

Format

```
#define _XOPEN_SOURCE
#include <regexp.h>

int advance(const char *string, const char *expbuf);

extern char *loc2, *locs;
```

General description

Restriction: This function is not supported in AMODE 64.

The advance() function attempts to match an input string of characters with the compiled regular expression which was obtained by an earlier call to compile().

The first parameter *string* is a pointer to a string of characters to be checked for a match.

expbuf is the pointer to the regular expression which was previously obtained by a call to compile().

The external variable *loc2* will point to the next character in *string* after the last character that matched the regular expression.

The external variable *locs* can be optionally set to point to some point in the input regular expression string to cause the advance() function to exit its back up loop.

Note: The external variables *cirf*, *sed*, and *nbra* are reserved.

During the pattern matching operation, when advance() encounters a * or \{\} sequence in the regular expression, it will advance its pointer to the string to be matched as far as possible and will recursively call itself trying to match the rest of the string to the rest of the regular expression. As long as there is no match, advance() will back up along the string until it finds a match or reaches the point in the string that initially matched the * or \{\}. It is sometime desirable to stop this backing up before the initial point in the string is reached. If the external character pointer *locs* is equal to the point in the string at some time during the back up process, advance() will break out of the loop that backs up and will return 0 (a failure indication).

Notes:

1. The application must provide the proper serialization for the compile(), step(), and advance() functions if they are run under a multithreaded environment.
2. The compile(), step(), and advance() functions are provided for historical reasons. These functions were part of the Legacy Feature in Single UNIX Specification, Version 2. They have been withdrawn and are not supported as part of Single UNIX Specification, Version 3. New applications should use the newer functions fnmatch(), glob(), regcomp() and regexec(), which provide full internationalized regular expression functionality compatible with IEEE Std 1003.1-2001.

Returned value

If the initial substring of *string* matches the regular expression in *expbuf*, advance() returns nonzero.

If there is no match, advance() returns 0.

If there is a match, advance() sets an external character pointer, *loc2*, as a side effect. The variable *loc2* points to the next character in *string* after the last character that matched the regular expression.

Related information

- "regexp.h" on page 61
- "compile() — Compile regular expression" on page 301
- "fnmatch() — Match file name or path name" on page 564
- "glob() — Generate path names matching a pattern" on page 807
- "regcomp() — Compile regular expression" on page 1415
- "regexec() — Execute compiled regular expression" on page 1421
- "step() — Pattern match with regular expression" on page 1726

__ae_correstbl_query() — Return coded character set ID type (ASCII and EBCDIC)

Standards

Standards / Extensions	C or C++	Dependencies
z/OS UNIX	both	z/OS V1R2

Format

```
#include <_Nascii.h>

__argument_t __ae_correstbl_query(void *search_argument, int src_arg_type,
                                  _AE_correstbl_t **ebcdic_entry_ptr,
                                  _AE_correstbl_t **ascii_entry_ptr);
```

General description

The __ae_correstbl_query() function is a method by which the user can obtain coded character set id (CCSID), type, and correspondence information from the EBCDIC/ASCII Correspondence and CCSID/Codeset Name Lookup Table, CEL4CTBL.

The user must provide the following:

Argument
    Description

***search_argument**
    A Codeset Name or CCSID search argument.

src_arg_type
    Specifies whether *search_argument* is a Codeset Name or a CCSID, using one of the defined values from <_Nascii.h>:

- _AE_CODESET_SRCH_ARG
- _AE_CCSID_SRCH_ARG

****ebcdic_entry_ptr**
    The address of an _AE_correstbl_t pointer for storing the EBCDIC table entry.

****ascii_entry_ptr**
    The address of an _AE_correstbl_t pointer for storing the ASCII table entry.

The function will then populate the supplied pointers with the address of _AE_correstbl_t structures containing the requested codeset's table entry as well as the address of the corresponding codeset's entry. However, not every EBCDIC codeset in the table has a corresponding ASCII encoding and vice versa. When a corresponding codeset does not exist, the pointer value returned in that argument is zero.

For consistency, the first _AE_correstbl_t pointer address argument will be populated with the EBCDIC entry address, and the second _AE_correstbl_t pointer will be populated with the ASCII entry address, regardless of which was the requested codeset and which was the corresponding codeset.

The __argument_t return value for __ae_correstbl_query() indicates the EBCDIC or ASCII type of the provided codeset.

Returned value

If successful, __ae_correstbl_query() returns either:

- _AE_EBCDIC_TYPE, when the requested entry is EBCDIC.
- _AE_ASCII_TYPE, when the requested entry is ASCII.

If unsuccessful, because the correspondence table cannot be loaded or the provided Codeset Name or CCSID is not valid, __ae_correstbl_query() returns _AE_UNKNOWN_TYPE.

Related information

- "_Ccsid.h" on page 17
- "_Nascii.h" on page 49
- "__CcsidType() — Return coded character set ID type" on page 246
- "__CSNameType() — Return codeset name type" on page 350
- "__toCcsid() — Convert codeset name to coded character set ID" on page 1896
- "__toCSName() — Convert coded character set ID to codeset name" on page 1897

aio_cancel() — Cancel an asynchronous I/O request

Standards

Standards / Extensions	C or C++	Dependencies
Single UNIX Specification, Version 2 Single UNIX Specification, Version 3	both	OS/390 V2R7

Format

```
#define _XOPEN_SOURCE 500
#include <aio.h>

int aio_cancel(int fildes, struct aiocb *aiocbp);
```

General description

The aio_cancel() function attempts to cancel one or more asynchronous I/O requests currently outstanding against file descriptor *fildes*. The *aiocbp* argument points to an *aiocb* structure for a particular request to be canceled or is NULL to cancel all outstanding cancelable requests against *fildes*.

Normal asynchronous notification occurs for asynchronous I/O operations that are successfully canceled. The associated error status is set to ECANCELED and the return status is set to -1 for the canceled requests.

For requests that cannot be canceled, the normal asynchronous completion process takes place when their I/O completes. In this case the *aiocb* is not modified by aio_cancel().

An asynchronous operation is cancelable if it is currently blocked or becomes blocked. Once an outstanding request can be completed it is allowed to complete. For example, an aio_read() will be cancelable if there is no data available at the time that aio_cancel() is called.

fildes must be a valid file descriptor, but when *aiocbp* is not NULL *fildes* does not have to match the file descriptor with which the asynchronous operation was initiated. For maximum portability, though, it should match.

The aio_cancel() function always waits for the request being canceled to either complete or be canceled. When control returns from aio_cancel(), the program may safely free the original request's aiocb and buffer. If a signal was specified on the original request, the signal handler for that request's I/O complete notification may run before, during, or after control returns from aio_cancel(), so coordination may be necessary between the signal handler and the caller of aio_cancel(). This is particularly unpredictable when aio_cancel() is called from a different thread than the original request, unless the original thread no longer exists.

Canceling all requests on a given descriptor does not stop new requests from being made or otherwise effect the descriptor. The program may start again or close the descriptor depending on why it issued the cancel.

An individual request can only be canceled once. Subsequent attempts to explicitly cancel the same request will fail with EALREADY.

Returned value

aio_cancel() returns one of the following values:

- AIO_CANCELED if the requested operations were canceled.
- AIO_NOTCANCELED if at least one of the requested operations cannot be canceled because it is in progress.

  In this case, the state of the other operations, if any, referenced in the call to aio_cancel() is not indicated by the return value of aio_cancel(). The application can determine the status of these operations by using aio_error().
- AIO_ALLDONE if all of the operations have already completed. This is returned when there are no outstanding requests found that match the criteria specified. This is also the result returned when a file associated with *fildes* does not support the asynchronous I/O function because there are no outstanding requests to be found that match the criteria specified.
- -1 if there was an error. aio_cancel() sets errno to one of the following values:

  Error Code
  Description

  EALREADY
  The operation to be canceled is already being canceled.

EBADF
The *fildes* argument is not a valid file descriptor.

Related information

- "aio.h" on page 17
- "aio_error() — Retrieve error status for an asynchronous I/O operation"
- "aio_read() — Asynchronous read from a socket" on page 147
- "aio_return() — Retrieve status for an asynchronous I/O operation" on page 150
- "aio_write() — Asynchronous write to a socket" on page 153

aio_error() — Retrieve error status for an asynchronous I/O operation

Standards

Standards / Extensions	C or C++	Dependencies
Single UNIX Specification, Version 2 Single UNIX Specification, Version 3	both	OS/390 V2R7

Format

```
#define _XOPEN_SOURCE 500
#include <aio.h>

int aio_error(const struct aiocb *aiocbp);
```

General description

The aio_error() function returns the error status associated with the *aiocb* structure referenced by the *aiocbp* argument. The error status for an asynchronous I/O operation is the errno value that would be set by the corresponding read(), or write() operation. If the operation has not yet completed, then the error status will be equal to EINPROGRESS.

Returned value

If the asynchronous I/O operation has completed successfully, aio_error() returns 0.

If the asynchronous I/O operation has completed unsuccessfully, aio_error() returns the error status as described for read(), or write().

If the asynchronous I/O operation has not yet completed, then EINPROGRESS is returned.

aio_error() does not set errno.

When the errno is returned is not EINPROGRESS and not zero, the errno2 set by either read() or write() can be retrieved by using the __errno2() function.

Related information

- "aio.h" on page 17
- "aio_read() — Asynchronous read from a socket" on page 147
- "aio_return() — Retrieve status for an asynchronous I/O operation" on page 150
- "aio_suspend() — Wait for an asynchronous I/O request" on page 151

aio_read() — Asynchronous read from a socket

Standards

Standards / Extensions	C or C++	Dependencies
Single UNIX Specification, Version 2 Single UNIX Specification, Version 3	both	OS/390 V2R7

Format

```
#define _XOPEN_SOURCE 500
#include <aio.h>

int aio_read(struct aiocb *aiocbp);
```

General description

The aio_read() function initiates an asynchronous read operation as described by the *aiocb* structure (the asynchronous I/O control block).

The *aiocbp* argument points to the `aiocb` structure. This structure contains the following members:

aio_fildes
file descriptor

aio_offset
file offset

aio_buf
location of buffer

aio_nbytes
length of transfer

aio_reqprio
request priority offset

aio_sigevent
signal number and value

aio_lio_opcode
operation to be performed

The operation reads up to *aio_nbytes* from the socket or file associated with *aio_fildes* into the buffer pointed to by *aio_buf*. The call to aio_read() returns when the request has been initiated or queued to the file or device (even when the data cannot be delivered immediately).

Asynchronous I/O is currently only supported for sockets. The *aio_offset* field may be set but it will be ignored.

With a stream socket an asynchronous read may be completed when the first packet of data arrives and the application may have to issue additional reads, either asynchronously or synchronously, to get all the data it wants. A datagram socket has message boundaries and the operation will not complete until an entire message has arrived.

The *aiocbp* value may be used as an argument to aio_error() and aio_return() functions in order to determine the error status and return status, respectively, of the asynchronous operation. While the operation is proceeding, the error status retrieved by aio_error() is EINPROGRESS; the return status retrieved by aio_return() however is unpredictable.

If an error condition is encountered during the queueing, the function call returns without having initiated or queued the request.

When the operation completes asynchronously the program can be notified by a signal as specified in the *aio_sigevent* structure. It is significantly more efficient to receive these notifications with sigwaitinfo() or sigtimedwait() than to let them drive a signal handler. At this time the return and error status will have been updated to reflect the outcome of the operation. The sigevent structure's notification function fields are not supported. If a signal is not desired the program can occasionally poll the aiocb with aio_error() until the result is no longer EINPROGRESS.

Be aware that the operation may complete, and the signal handler may be delivered, before control returns from the call to aio_read(). Even when the operation does complete this quickly the return value from the call to aio_read() will be zero, reflecting the queueing of the I/O request not the results of the I/O itself.

An asynchronous operation may be canceled with aio_cancel() before its completion. Canceled operations complete with an error status of ECANCELED and any specified signal will be delivered. Due to timing, the operation may still complete naturally, either successfully or unsuccessfully, before it can be canceled by aio_cancel().

If the file descriptor of this operation is closed, the operation will be deleted if it has not completed or is not just about to complete. Signals specified for deleted operations will not be delivered. Close() will wait for asynchronous operations in progress for the descriptor to be deleted or completed.

You may use aio_suspend() to wait for the completion of asynchronous operations.

Sockets must be in blocking state or the operation may fail with EWOULDBLOCK.

If the control block pointed by *aiocbp* or the buffer pointed to by *aio_buf* becomes an illegal address before the asynchronous I/O completion, then the behavior of aio_read() is unpredictable.

If the thread that makes the aio_read() request terminates before the I/O completes the *aiocb* structure will still be updated with the return and error status, and any specified signal will be delivered to the process in which the thread was running. If thread related storage was used on the request the results are quite unpredictable.

Simultaneous asynchronous operations using the same *aiocbp*, asynchronous operations using a non-valid *aiocbp*, or any system action, that changes the process memory space while asynchronous I/O is outstanding to that address range, will produce unpredictable results

Simultaneous aio_read() operations on the same socket should not be done in general. With stream sockets, the I/O complete notifications may not be delivered in the same order as the bytes to which they refer, and so the byte stream may appear out of order. With UDP sockets, each datagram will complete one aio_read() operation, but you should not use multiple aio_reads for UDP sockets because this can cause significantly more system overhead as data arrives than a single outstanding request would.

There are several sockets oriented extensions to asynchronous I/O available with the BPX1AIO callable service, such as asynchronous accept(), asynchronous accept_and_recv(), asynchronous forms of all five pairs of read and write type operations, and receiving I/O completion notifications via an ECB, exit program, or through a message queue. The <aio.h> header contains all the structure fields, constants, and prototypes necessary to use BPX1AIO from a C program. These extensions are exposed when the _AIO_OS390 feature test macro is defined. The BPX1AIO stub resides in SYS1.CSSLIB and must be bound with your program. For a more detailed description of asynchronous I/O services, see BPX1AIO in *z/OS UNIX System Services Programming: Assembler Callable Services Reference*.

The *aio_lio_opcode* field is set to LIO_READ by the function aio_read().

_POSIX-PRIORITIZED_IO is not supported. The *aio_reqprio* field may be set but it will be ignored.

_POSIX_SYNCHRONIZED_IO is not supported.

Returned value

If successful, aio_read() returns 0 to the calling process.

If unsuccessful, aio_read() returns -1 and sets errno to one of the following values:

Error Code
    Description

EAGAIN
    The requested asynchronous I/O operation was not queued due to system resource limitations.

ENOSYS
    The file associated with *aio_fildes* does not support the aio_read() function.

Each of the following conditions may be detected synchronously at the time of the call to aio_read(), or asynchronously. If any of the conditions below are detected synchronously, aio_read() returns -1 and sets the errno to the corresponding value. If any of the conditions below are detected asynchronously, the return status of the asynchronous operation is set to -1, and the error status of the asynchronous operation will be set to the corresponding value.

Error Code
    Description

EBADF
    The *aio_fildes* argument is not a valid file descriptor open for reading.

EINVAL
    aio_sigevent contains an non-valid value.

EWOULDBLOCK
    The file associated with *aio_fildes* is in nonblocking state and there is no data available.

In the case where the aio_read() function successfully queues the I/O operation but the operation is subsequently canceled or encounters an error, the return status of the asynchronous operations is set to -1, and the error status of the asynchronous operation will be set to the error status normally set by the read() function call, or to the following value:

Error Code
Description

ECANCELED
The requested I/O was canceled before the I/O completed due to an explicit call to aio_cancel().

Related information

- "aio.h" on page 17
- "aio_cancel() — Cancel an asynchronous I/O request" on page 144
- "aio_error() — Retrieve error status for an asynchronous I/O operation" on page 146
- "aio_return() — Retrieve status for an asynchronous I/O operation"
- "aio_suspend() — Wait for an asynchronous I/O request" on page 151
- "aio_write() — Asynchronous write to a socket" on page 153

aio_return() — Retrieve status for an asynchronous I/O operation

Standards

Standards / Extensions	C or C++	Dependencies
Single UNIX Specification, Version 2 Single UNIX Specification, Version 3	both	OS/390 V2R7

Format

```
#define _XOPEN_SOURCE 500
#include <aio.h>

int aio_return(const struct aiocb *aiocbp);
```

General description

The aio_return() function returns the return status associated with the *aiocb* structure referenced by the *aiocbp* argument. The return status for an asynchronous I/O operation is the value that would be set by the corresponding read() or write() operation. While the operation is proceeding, the error status retrieved by aio_error() is EINPROGRESS; the return status retrieved by aio_return() however is unpredictable. The aio_return() function may be called to retrieve the return status of a given asynchronous operation; once aio_error() has returned with 0.

Returned value

If the asynchronous I/O operation has completed successfully, aio_return() returns the status as described for read() or write().

If the asynchronous I/O operation has not yet completed, then the return status is unpredictable.

aio_return() does not set errno.

Related information

- "aio.h" on page 17
- "aio_error() — Retrieve error status for an asynchronous I/O operation" on page 146
- "aio_read() — Asynchronous read from a socket" on page 147
- "aio_suspend() — Wait for an asynchronous I/O request"

aio_suspend() — Wait for an asynchronous I/O request

Standards

Standards / Extensions	C or C++	Dependencies
Single UNIX Specification, Version 2 Single UNIX Specification, Version 3	both	OS/390 V2R7

Format

```
#define _XOPEN_SOURCE 500
#include <aio.h>

int aio_suspend(const struct aiocb *const list[],
                int nent, const struct timespec *timeout);
```

General description

The aio_suspend() function suspends the calling thread when the *timeout* is a NULL pointer until at least one of the asynchronous I/O operations referenced by the *list* argument has completed, or until a signal interrupts the function. Or, if *timeout* is not NULL, it is suspended until the time interval specified by *timeout* has passed. If the time interval indicated in the *timespec* structure pointed to by *timeout* passes before any of the I/O operations referenced by *list*, then aio_suspend() returns with an error. If any of the *aoicb* structures in the list correspond to completed asynchronous I/O operations (that is, the error status for the operation is not equal to EINPROGRESS) at the time of the call, the function returns without suspending the calling thread.

The *list* argument is an array of pointers to asynchronous I/O control blocks (AIOCBs). The *nent* argument indicates the number of elements in the array. Each *aiocb* structure pointed to will have been used in initiating an asynchronous I/O request. This array may contain NULL pointers, which are ignored. If this array contains pointers that refer to *aiocb* structures that have not been used in submitting asynchronous I/O or *aiocb* structures that are not valid, the results are unpredictable.

Returned value

If successful, aio_suspend() returns 0.

If unsuccessful, aio_suspend() returns -1. The application may determine which asynchronous I/O completed by scanning the associated error and return status using aio_error() or aio_return(), respectively. aio_suspend() sets errno to one of the following values:

Error Code
Description

EAGAIN
No asynchronous I/O indicated in the list referenced by *list* completed in the time interval indicated by *timeout*.

EINTR
A signal interrupted the aio_suspend() function. Note that, since each asynchronous I/O operation may possibly provoke a signal when it completes, this error return may be caused by the completion of one (or more) of the very I/O operations being awaited.

ENOSYS
z/OS UNIX System Services does not support the aio_suspend() function.

Usage notes

1. The AIOCBs represented by the list of AIOCB pointers must reside in the same storage key as the key of the invoker of aio_suspend. If the AIOCB Pointer List or any of the AIOCBs represented in the list are not accessible by the invoker an EFAULT may occur.
2. AIOCB pointers in the list with a value of zero will be ignored.
3. A timeout value of zero (seconds+nanoseconds) means that the aio_suspend() call will not wait at all. It will check for any completed asynchronous I/O requests. If none are found it will return with a EAGAIN. If at least one is found aio_suspend() will return with success.
4. A timeout value of a *timespec* with the tv_sec field set with INT_MAX, as defined in <limits.h>, will cause the aio_suspend service to wait until a asynchronous I/O request completes or a signal is received.

   If the Macro _AIO_OS390 is defined then the following may also apply.
5. The number of pointers to AIOCBs that use application supplied event control block (ECB) pointers for invocations of asynchronous I/O is limited to 253. There is no limit when not using the _AIO_OS390 Feature Test Macro. See *z/OS UNIX System Services Programming: Assembler Callable Services Reference* under the BPX1AIO for information on supplying user-defined ECBs in the AIOCB data area.
6. The AIOCBs passed to aio_suspend() must not be freed or reused by other threads in the process while this service is still in progress. This service may use the AIOCBs even after the asynchronous I/O completes. This restriction excludes multiple threads from doing aio_suspend() on the same AIOCB at the same time. Modifying the AIOCB during an aio_suspend() will produce unpredictable results.
7. The use of these extensions will require macros from SYS1.CSSLIB. Make sure that it is included in the SYSLIB concatenation during the compile.

Related information

- “aio.h” on page 17
- “aio_error() — Retrieve error status for an asynchronous I/O operation” on page 146
- “aio_read() — Asynchronous read from a socket” on page 147
- “aio_return() — Retrieve status for an asynchronous I/O operation” on page 150

aio_write() — Asynchronous write to a socket

Standards

Standards / Extensions	C or C++	Dependencies
Single UNIX Specification, Version 2 Single UNIX Specification, Version 3	both	OS/390 V2R7

Format

```
#define _XOPEN_SOURCE 500
#include <aio.h>

int aio_write(struct aiocb *aiocbp);
```

General description

The aio_write() function initiates an asynchronous write as described by the *aiocb* structure (the asynchronous I/O control block).

The *aiocbp* argument points to the `aiocb` structure. This structure contains the following members:

aio_fildes
file descriptor

aio_offset
file offset

aio_buf
location of buffer

aio_nbytes
length of transfer

aio_reqprio
request priority offset

aio_sigevent
signal number and value

aio_lio_opcode
operation to be performed

The operation will write *aio_nbytes* from the buffer pointed to by *aio_buf* to the socket or file associated with *aio_fildes*. The call to aio_write() returns when the request has been initiated or queued to the file or device (even when the data cannot be delivered immediately).

Asynchronous I/O is currently only supported for sockets. The *aio_offset* field may be set but it will be ignored.

The *aiocbp* value may be used as an argument to aio_error() and aio_return() functions in order to determine the error status and return status, respectively, of the asynchronous operation. While the operation is proceeding, the error status retrieved by aio_error() is `EINPROGRESS`; the return status retrieved by aio_return() however is unpredictable.

If an error condition is encountered during the queueing, the function call returns without having initiated or queued the request.

When the operation completes asynchronously the program can be notified by a signal as specified in the *aio_sigevent* structure. It is significantly more efficient to receive these notifications with sigwaitinfo() or sigtimedwait() than to let them drive a signal handler. At this time the return and error status will have been updated to reflect the outcome of the operation. The sigevent structure's notification function fields are not supported. If a signal is not desired the program can occasionally poll the aiocb with aio_error() until the result is no longer EINPROGRESS.

Be aware that the operation may complete, and the signal handler may be delivered, before control returns from the call to aio_read(). Even when the operation does complete this quickly the return value from the call to aio_read() will be zero, reflecting the queueing of the I/O request not the results of the I/O itself.

An asynchronous operation may be canceled with aio_cancel() before its completion. Canceled operations complete with an error status of ECANCELED and any specified signal will be delivered. Due to timing, the operation may still complete naturally, either successfully or unsuccessfully, before it can be canceled by aio_cancel().

If the file descriptor of this operation is closed, the operation will be deleted if it has not completed or is not just about to complete. Signals specified for deleted operations will not be delivered. Close() will wait for asynchronous operations in progress for the descriptor to be deleted or completed.

You may use aio_suspend() to wait for the completion of asynchronous operations.

Sockets must be in blocking state or the operation may fail with EWOULDBLOCK.

If the control block pointed by *aiocbp* or the buffer pointed to by *aio_buf* becomes an illegal address before the asynchronous I/O completion, then the behavior of aio_read() is unpredictable

If the thread that makes the aio_read() request terminates before the I/O completes, the *aiocb* structure will still be updated with the return and error status, and any specified signal will be delivered to the process in which the thread was running. If thread related storage was used on the request the results are quite unpredictable.

Simultaneous asynchronous operations using the same *aiocbp*, attempting asynchronous operations using a non-valid *aiocbp*, or any system action, that changes the process memory space while asynchronous I/O is outstanding to that address range, will produce unpredictable results.

Simultaneous aio_write() operations on the same stream socket should not be done because the data may be transmitted on the network out of order. With UDP sockets each aio_write defines a single datagram and there is no implied order of arrival in UDP. Beware, though, of sending too many datagrams. If there is network congestion or the receiver is slow you can tie up a large amount of system storage with uncontrolled aio_writes, and eventually they may start to fail with ENOBUFS.

There are several sockets oriented extensions to asynchronous I/O available with the BPX1AIO callable service, such as asynchronous accept(), asynchronous accept_and_recv(), asynchronous forms of all five pairs of read and write type operations, and receiving I/O completion notifications via an ECB, exit program, or through a message queue. The <aio.h> header contains all the structure fields, constants, and prototypes necessary to use BPX1AIO from a C program. These extensions are exposed when the _AIO_OS390 feature test macro is defined. The BPX1AIO stub resides in SYS1.CSSLIB and must be bound with your program. For a more detailed description of asynchronous I/O services, see BPX1AIO in *z/OS UNIX System Services Programming: Assembler Callable Services Reference*.

The *aio_lio_opcode* field is set to LIO_WRITE by the function aio_write().

_POSIX_PRIORITIZED_IO is not supported. The *aio_reqprio* field may be set but it will be ignored.

_POSIX_SYNCHRONIZED_IO is not supported.

Returned value

If the I/O operation is successfully queued, aio_write() returns 0 to the calling process.

If the I/O operation is not queued, aio_write() returns -1 and sets errno to one of the following values:

Error Code
  Description

EAGAIN
  The requested asynchronous I/O operation was not queued due to system resource limitations.

ENOSYS
  The file associated with *aio_fildes* does not support the aio_write() function.

Each of the following conditions may be detected synchronously at the time of the call to aio_write(), or asynchronously. If any of the conditions below are detected synchronously, aio_write() returns -1 and sets the errno to the corresponding value. If any of the conditions below are detected asynchronously, the return status of the asynchronous operation is set to -1, and the error status of the asynchronous operation will be set to the corresponding value.

Error Code
  Description

EBADF
  The *aio_fildes* argument is not a valid file descriptor open for writing.

EINVAL
  The *aio_nbytes* is not a valid value or *aio_sigevent* contains a value that is not valid.

EWOULDBLOCK
  The file associated with *aio_fildes* is in nonblocking state and there is no data available.

In the case where aio_write() successfully queues the I/O operation but the operation is subsequently canceled or encounters an error, the return status of the asynchronous operation is set to -1, and the error status of the asynchronous operation is set to the error status normally set by the write() function call, or to the following value:

Error Code
Description

ECANCELED
The requested I/O was canceled before the I/O completed due to an explicit call to aio_cancel().

Related information

- “aio.h” on page 17
- “aio_cancel() — Cancel an asynchronous I/O request” on page 144
- “aio_error() — Retrieve error status for an asynchronous I/O operation” on page 146
- “aio_read() — Asynchronous read from a socket” on page 147
- “aio_return() — Retrieve status for an asynchronous I/O operation” on page 150
- “aio_suspend() — Wait for an asynchronous I/O request” on page 151

alarm() — Set an alarm

Standards

Standards / Extensions	C or C++	Dependencies
POSIX.1 XPG4 XPG4.2 Single UNIX Specification, Version 3	both	POSIX(ON)

Format

```
#define _POSIX_SOURCE
#include <unistd.h>

unsigned int alarm(unsigned int seconds);
```

General description

Generates a SIGALRM signal after the number of seconds specified by the *seconds* parameter has elapsed. The SIGALRM signal delivery is directed at the calling thread.

seconds is the number of real seconds to wait before the SIGALRM signal is generated. Because of processor delays, the SIGALRM signal may be generated slightly later than this specified time. If *seconds* is zero, any previously set alarm request is canceled.

Only one such alarm can be active at a time. If you set a new alarm time, any previous alarm is canceled.

This function is supported only in a POSIX program.

Special behavior for XPG4: The fork() function clears pending alarms in the child thread. However, a new thread image created by one of the *exec* functions inherits the time left to an alarm in the old thread's image.

Special behavior for XPG4.2: alarm() will interact with the setitimer() function when the setitimer() function is used to set the ‘real’ interval timer (`ITIMER_REAL`).

alarm() does not interact with the usleep() function.

Returned value

If a prior alarm request has not yet completed, alarm() returns the number of seconds remaining until that request would have generated a SIGALRM signal.

If there are no prior alarm requests with time remaining, alarm() returns 0. Because alarm() is always successful, there is no failure return. If any failures are encountered that prevent alarm() from completing successfully, an abend is generated.

Example

CELEBA05

```
/* CELEBA05

   The following example generates a SIGALRM signal.

 */
#define _POSIX_SOURCE
#include <stdio.h>
#include <signal.h>
#include <time.h>
#include <unistd.h>

volatile int footprint=0;

void catcher(int signum) {
  puts("inside signal catcher!");
  footprint = 1;
}

main() {
  struct sigaction sact;
  volatile double count;
  time_t t;

  sigemptyset(&sact.sa_mask);
  sact.sa_flags = 0;
  sact.sa_handler = catcher;
  sigaction(SIGALRM, &sact, NULL);

  alarm(5); /* timer will pop in five seconds */

  time(&t);
  printf("before loop, time is %s", ctime(&t));
  for (count=0; (count<1e10) && (footprint == 0); count++);
  time(&t);
  printf("after loop, time is %s", ctime(&t));

  printf("the sum so far is %.0f\n", count);

  if (footprint == 0)
    puts("the signal catcher never gained control");
  else
    puts("the signal catcher gained control");
}
```

Output

```
before loop, time is Fri Jun 16 08:37:03 2006
inside signal catcher!
after loop, time is Fri Jun 16 08:37:08 2006
the sum so far is 17417558
```

Related information

- "signal.h" on page 63
- "unistd.h" on page 82
- "exec functions" on page 436
- "fork() — Create a new process" on page 571
- "pause() — Suspend a process pending a signal" on page 1173
- "setitimer() — Set value of an interval timer" on page 1544
- "sigaction() — Examine or change a signal action" on page 1611
- "signal() — Handle interrupts" on page 1643
- "sigprocmask() — Examine or change a thread" on page 1651
- "sleep() — Suspend execution of a thread" on page 1678
- "usleep() — Suspend execution for an interval" on page 1966

alloca() — Allocate storage from the stack

Standards

Standards / Extensions	C or C++	Dependencies
	both	OS/390 V2R6

Format

```
#include <stdlib.h>

void *alloca(unsigned int size);
```

General description

The built-in alloca() function obtains memory from the stack. This eliminates the need for an explicit free() as the memory is freed when the stack is collapsed.

If the alloca() function is unable to obtain the requested storage, control will not return to the caller. Instead the application will terminate due to an out of memory condition (if the reserve stack is available and the caller is not XPLINK), or it will terminate with an abend indicating that storage could not be obtained.

To avoid infringing on the user's name space, this nonstandard function is exposed only when you use the compiler option, LANGLVL(EXTENDED). When you use LANGLVL(EXTENDED) any relevant information in the header is also exposed.

Note: Storage from an alloca() is done after a setjmp() (or any variation thereof) is freed on a longjmp() (or any variation thereof) to an XPLINK-compiled function, and not freed on a longjmp() to a NOXPLINK-compiled function. See the longjmp() family of functions for more details.

Returned value

If successful, alloca() returns the address of the requested storage.

Related information

- "stdlib.h" on page 70
- "Built-in functions" on page 93
- "getcontext() — Get user context" on page 694
- "longjmp() — Restore stack environment" on page 1005
- "_longjmp() — Nonlocal goto" on page 1008
- "setcontext() — Restore user context" on page 1524

- “setjmp() — Preserve stack environment” on page 1546
- “_setjmp() — Set jump point for a nonlocal goto” on page 1549
- “siglongjmp() — Restore the stack environment and signal mask” on page 1640
- “sigsetjmp() — Save stack environment and signal mask” on page 1659
- “swapcontext() — Save and restore user context” on page 1792

arm_bind_thread() — Bind the current thread to a given transaction

Standards

Standards / Extensions	C or C++	Dependencies
	both	

Format

```
#include <sys/_Elmarm4.h>

arm_error_t arm_bind_thread(
/* [in]     */  arm_start_handle_t     start_handle,
/* [in]     */  arm_int32_t            flags,
/* [in]     */  arm_buffer4_t          *buffer4,
);
```

General description

Use arm_bind_thread() to indicate the current thread is performing processing on behalf of a given transaction, and no other transaction. This enables eWLM to collect resource usage and delay information for threads serving each class of work, and to adjust the resources given to threads to help them meet the goals assigned to those classes.

Any number of threads can bind to the same transaction at the same time. However a single thread cannot bind to more than one transaction at the same time. If a thread calls arm_bind_thread() when it is already bound to a transaction, the call is not honored and it returns a negative value to indicate an error.

A thread that calls arm_bind_thread() must call arm_unbind_thread() when it completes its processing on behalf of the transaction. (If arm_unbind_thread() is not called, then arm_stop_transaction() unbinds any threads that remain bound to the transaction. This exists for recovery purposes. Applications are expected to use arm_unbind_thread() as part of normal processing.)

start_handle
The handle returned by arm_start_transaction() for the transaction.

flags
Reserved for future use. The argument must be set to 0.

buffer4
A pointer to a buffer that identifies one or more sub-buffers containing additional data. Currently no sub-buffers are defined for this function so a null pointer should be passed. If a buffer is passed eWLM ignores it.

Returned value

On success, arm_bind_thread returns ARM_RC_SUCCESS. On failure, the errno and return code are set to indicate the error. See “Return code” on page 160 for the list of all possible return codes.

Error Code
Description

EFAULT
A parameter of this service contained an address that was not accessible to the caller.

EINVAL
A parameter of this service contained a value that was not valid.

EMVSARMERROR
An ARM error occurred. Refer to the return code for the specific error.

EPERM
The caller does not have the appropriate privileges. The return code is set to ARM_RC_AUTH_ERROR.

EMVSSAF2ERR
An error occurred in the security product.

Return code

The following list contains all possible return codes for the ARM function calls.

Return Code
Description

ARM_RC_APP_INPUT
User provided application id is invalid.

ARM_RC_APPL_INST_MAX
Maximum number of application instances per process is exceeded.

ARM_RC_APPL_INST_NAME
Application instance name is too long.

ARM_RC_APPL_MAX
Maximum number of registered applications is exceeded.

ARM_RC_APPL_NAME
Application name is missing or too long.

ARM_RC_AUTH_ERROR
User is not authorized to perform ARM calls.

ARM_RC_BLOCKED_MAX
Maximum number of blocked threads per transaction is exceeded.

ARM_RC_CORR_BAD_SIZE
Correlator size is not valid.

ARM_RC_CORR_EFAULT
Correlator return address is not valid.

ARM_RC_ENOMEM
Process or system is out of memory.

ARM_RC_FLAGS_EINVAL
Flags value is not valid.

ARM_RC_GRP_MAX
Maximum number of registered application groups is exceeded.

ARM_RC_GRP_NAME
Application group name is too long.

ARM_RC_HANDLE_EFAULT
Handle return address is not valid.

ARM_RC_MAXMEM
Maximum ARM services memory limit is exceeded.

ARM_RC_PARENT_CORR_INVAL
Parent correlator is not valid.

ARM_RC_PARENT_CORR_SZ
Parent correlator is too small.

ARM_RC_PARENT_CORR_VERSION
Parent correlator version is not correct.

ARM_RC_PROC_VAL_MSMTCH
Property values do not match property names.

ARM_RC_PROP_NAME
Transaction property name is too long.

ARM_RC_PROP_NAME_MAX
Too many property names are passed to ARM call.

ARM_RC_PROP_NAME_SUBBUF_MAX
Too many property name sub-buffers are passed to ARM call.

ARM_RC_PROP_VAL
Transaction property value is too long.

ARM_RC_PROP_VAL_MAX
Too many property values are passed to ARM call.

ARM_RC_PROP_VAL_SUBBUF_MAX
Too many property value sub-buffers are passed to ARM call.

ARM_RC_TIME_SUBBUF_MAX
Too many arrival time sub-buffers are passed to ARM call.

ARM_RC_TRAN_INPUT
User provided transaction type id is not valid.

ARM_RC_TRAN_MAX
Maximum number of transactions is exceeded.

ARM_RC_TRAN_STATUS_EINVAL
Invalid transaction status.

ARM_RC_TRAN_TYPE_INST_MAX
Maximum number of registered transaction type instances per process is exceeded.

ARM_RC_TRAN_TYPE_MAX
Maximum number of registered transaction types is exceeded.

ARM_RC_TRAN_TYPE_NAME
Transaction type name is missing or too long.

ARM_RC_UNKN_APPL_INST
Application instance is not found.

ARM_RC_UNKN_BLOCKED
Blocked thread is not valid for the specified transaction.

ARM_RC_UNKN_PARENT
Parent transaction is not a valid transaction.

ARM_RC_UNKN_PROC
The calling processing did not register an application instance.

ARM_RC_UNKN_TRAN
Transaction was not found for the calling process.

ARM_RC_UNKN_TRANCLASS_INST
Transaction class instance is not found.

ARM_RC_URI_NAME
URI value is too long.

ARM_RC_URI_SUBBUF_MAX
Too many URI value sub-buffers are passed to ARM call.

ARM_RC_VERSION_NAME
Application version name is too long.

Related information

- "arm_blocked() — Indicate the processing of a transaction is blocked"
- "arm_correlator_get_length() — Get the actual size of the transaction correlator" on page 164
- "arm_end_application() — Undefines an ARM application" on page 165
- "arm_get_correlator_max_length() — Get the max length of the transaction correlator" on page 167
- "arm_get_timestamp() — Get the current timestamp" on page 168
- "arm_init_application() — Defines an ARM application" on page 169
- "arm_init_transaction_type() — Defines and initializes an ARM transaction type" on page 171
- "arm_start_transaction() — Mark the start of an ARM transaction" on page 173
- "arm_stop_transaction() — Mark the end of an ARM transaction" on page 175
- "arm_unbind_thread() — Unbind the current thread to a given transaction" on page 177
- "arm_unblocked() — Indicate the processing of a transaction is no longer blocked" on page 179
- "arm_update_transaction() — Update a given transaction" on page 180

arm_blocked() — Indicate the processing of a transaction is blocked

Standards

Standards / Extensions	C or C++	Dependencies
	both	

Format

```
#include <sys/_Elmarm4.h>

arm_error_t arm_blocked(
/* [in]     */  arm_start_handle_t     start_handle,
/* [in]     */  arm_int32_t       flags,
/* [in]     */  arm_buffer4_t   *buffer4,
/* [out]    */  arm_block_handle_t   *block_handle
);
```

General description

Use arm_blocked() to indicate that processing of a transaction is blocked waiting for a transaction in another application to complete. This helps to identify what portion of a transaction's response time is spent in its own application and what portion is spent in downstream applications.

`start_handle`
The handle returned by arm_start_transaction() for the transaction.

`flags`
Reserved for future use. The argument must be set to 0.

`buffer4`
A pointer to a buffer that identifies one or more sub-buffers containing additional data. Currently no sub-buffers are defined for this function so a null pointer should be passed. If a buffer is passed eWLM ignores it.

`block_handle`
A pointer to a 64-bit area where a handle that identifies the arm_blocked() call is returned. The handle must be passed to arm_unblocked(). The handle is valid only within the caller's process.

Returned value

On success, arm_blocked returns ARM_RC_SUCCESS. On failure, the errno and return code are set to indicate the error. See “Return code” on page 160 for the list of all possible return codes. On failure, the block_handle is set to a dummy value which can be used for later calls to other interfaces. Those interfaces will recognize that the block_handle contains a dummy value and return without performing any action.

Error Code
Description

EFAULT
A parameter of this service contained an address that was not accessible to the caller.

EINVAL
A parameter of this service contained a value that was not valid.

EMVSARMERROR
An ARM error occurred. Refer to the return code for the specific error.

EPERM
The caller does not have the appropriate privileges. The return code is set to ARM_RC_AUTH_ERROR.

EMVSSAF2ERR
An error occurred in the security product.

Related information

- “arm_bind_thread() — Bind the current thread to a given transaction” on page 159
- “arm_correlator_get_length() — Get the actual size of the transaction correlator” on page 164
- “arm_end_application() — Undefines an ARM application” on page 165
- “arm_get_correlator_max_length() — Get the max length of the transaction correlator” on page 167

- "arm_get_timestamp() — Get the current timestamp" on page 168
- "arm_init_application() — Defines an ARM application" on page 169
- "arm_init_transaction_type() — Defines and initializes an ARM transaction type" on page 171
- "arm_start_transaction() — Mark the start of an ARM transaction" on page 173
- "arm_stop_transaction() — Mark the end of an ARM transaction" on page 175
- "arm_unbind_thread() — Unbind the current thread to a given transaction" on page 177
- "arm_unblocked() — Indicate the processing of a transaction is no longer blocked" on page 179
- "arm_update_transaction() — Update a given transaction" on page 180

arm_correlator_get_length() — Get the actual size of the transaction correlator

Standards

Standards / Extensions	C or C++	Dependencies
	both	

Format

```
#include <sys/_Elmarm4.h>

arm_error_t arm_correlator_get_length(
/* [in]     */  arm_correlator_t   *corr_ptr,
/* [out] */  arm_correlator_length_t   **result_ptr
);
```

General description

Returns the actual size of a given correlator.

corr_ptr
Address of correlator.

result_ptr
Set to the address of the length of the correlator.

Returned value

On success, arm_correlator_get_length returns ARM_RC_SUCCESS. On failure, the errno and return code are set to indicate the error. See "Return code" on page 160 for the list of all possible return codes.

Error Code
Description

EFAULT
A parameter of this service contained an address that was not accessible to the caller.

EINVAL
A parameter of this service contained a value that was not valid.

EMVSARMERROR
An ARM error occurred. Refer to the return code for the specific error.

EPERM
The caller does not have the appropriate privileges. The return code is set to ARM_RC_AUTH_ERROR.

EMVSSAF2ERR
An error occurred in the security product.

Related information

- "arm_bind_thread() — Bind the current thread to a given transaction" on page 159
- "arm_blocked() — Indicate the processing of a transaction is blocked" on page 162
- "arm_end_application() — Undefines an ARM application"
- "arm_get_correlator_max_length() — Get the max length of the transaction correlator" on page 167
- "arm_get_timestamp() — Get the current timestamp" on page 168
- "arm_init_application() — Defines an ARM application" on page 169
- "arm_init_transaction_type() — Defines and initializes an ARM transaction type" on page 171
- "arm_start_transaction() — Mark the start of an ARM transaction" on page 173
- "arm_stop_transaction() — Mark the end of an ARM transaction" on page 175
- "arm_unbind_thread() — Unbind the current thread to a given transaction" on page 177
- "arm_unblocked() — Indicate the processing of a transaction is no longer blocked" on page 179
- "arm_update_transaction() — Update a given transaction" on page 180

arm_end_application() — Undefines an ARM application

Standards

Standards / Extensions	C or C++	Dependencies
	both	

Format

```
#include <sys/_Elmarm4.h>

arm_error_t arm_end_application(
/* [in/out] */  arm_appl_id_t   *application_id
/* [in]     */  arm_int32_t      flags,
/* [in]     */  arm_buffer4_t   *buffer4,
);
```

General description

Use arm_end_application() to close ARM API activity for a particular application. arm_end_application() invalidates the application handle returned by arm_init_application(), all transaction type handles associated with the application handle, and all transaction handles associated with those transaction types. Any further API calls using these handles will be rejected with a negative status return value. Information about transactions that were started and not yet stopped is discarded.

`application_id`
The handle passed to or returned by arm_init_application() for the application.

`flags`
Reserved for future use. The argument must be set to 0.

`buffer4`
A pointer to a buffer that identifies one or more sub-buffers containing additional data. Currently no sub-buffers are defined for this function so a null pointer should be passed. If a buffer is passed eWLM ignores it.

Returned value

On success, arm_stop_transaction returns ARM_RC_SUCCESS. On failure, the errno and return code are set to indicate the error. See "Return code" on page 160 for the list of all possible return codes.

Error Code
Description

EFAULT
A parameter of this service contained an address that was not accessible to the caller.

EINVAL
A parameter of this service contained a value that was not valid.

EMVSARMERROR
An ARM error occurred. Refer to the return code for the specific error.

EPERM
The caller does not have the appropriate privileges. The return code is set to ARM_RC_AUTH_ERROR.

EMVSSAF2ERR
An error occurred in the security product.

Related information

- "arm_bind_thread() — Bind the current thread to a given transaction" on page 159
- "arm_blocked() — Indicate the processing of a transaction is blocked" on page 162
- "arm_correlator_get_length() — Get the actual size of the transaction correlator" on page 164
- "arm_get_correlator_max_length() — Get the max length of the transaction correlator" on page 167
- "arm_get_timestamp() — Get the current timestamp" on page 168
- "arm_init_application() — Defines an ARM application" on page 169
- "arm_init_transaction_type() — Defines and initializes an ARM transaction type" on page 171
- "arm_start_transaction() — Mark the start of an ARM transaction" on page 173
- "arm_stop_transaction() — Mark the end of an ARM transaction" on page 175
- "arm_unbind_thread() — Unbind the current thread to a given transaction" on page 177
- "arm_unblocked() — Indicate the processing of a transaction is no longer blocked" on page 179
- "arm_update_transaction() — Update a given transaction" on page 180

arm_get_correlator_max_length() — Get the max length of the transaction correlator

Standards

Standards / Extensions	C or C++	Dependencies
	both	

Format

```
#include <sys/_Elmarm4.h>

arm_int32_t arm_get_correlator_max_length();
```

General description

Returns the size of the buffer required for the correlator returned by arm_start_transaction().

Note: The size applies only to correlators returned by arm_start_transaction(). It does not necessarily apply to parent correlators received from other applications, which may have been created by another ARM implementation and may have a different maximum size. The ARM 2.0 and ARM 3.0 standards defined the maximum size of the correlator for any implementation as 168 bytes. However the proposed ARM 4.0 standard most likely will drop this restriction and implementations will be allowed to create a correlator of any length.

Returned value

On success, arm_iget_correlator_max_length returns a positive number representing the maximum correlator length. On failure, the errno and return code are set to indicate the error. See “Return code” on page 160 for the list of all possible return codes.

Error Code
Description

EFAULT
A parameter of this service contained an address that was not accessible to the caller.

EINVAL
A parameter of this service contained a value that was not valid.

EMVSARMERROR
An ARM error occurred. Refer to the return code for the specific error.

EPERM
The caller does not have the appropriate privileges. The return code is set to ARM_RC_AUTH_ERROR.

EMVSSAF2ERR
An error occurred in the security product.

Related information

- “arm_bind_thread() — Bind the current thread to a given transaction” on page 159

- "arm_blocked() — Indicate the processing of a transaction is blocked" on page 162
- "arm_correlator_get_length() — Get the actual size of the transaction correlator" on page 164
- "arm_end_application() — Undefines an ARM application" on page 165
- "arm_get_timestamp() — Get the current timestamp"
- "arm_init_application() — Defines an ARM application" on page 169
- "arm_init_transaction_type() — Defines and initializes an ARM transaction type" on page 171
- "arm_start_transaction() — Mark the start of an ARM transaction" on page 173
- "arm_stop_transaction() — Mark the end of an ARM transaction" on page 175
- "arm_unbind_thread() — Unbind the current thread to a given transaction" on page 177
- "arm_unblocked() — Indicate the processing of a transaction is no longer blocked" on page 179
- "arm_update_transaction() — Update a given transaction" on page 180

arm_get_timestamp() — Get the current timestamp

Standards

Standards / Extensions	C or C++	Dependencies
	both	

Format

```
#include <sys/_Elmarm4.h>

arm_error_t arm_get_timestamp(
/* [out] */  arm_timestamp_t   *timestamp
);
```

General description

Use arm_get_timestamp() to obtain the current time. If the application wants to pass the arrival time sub-buffer to arm_start_transaction(), it must call arm_get_timestamp() to capture the arrival time in a format that is compatible with the arrival time sub-buffer.

timestamp
Area where the current time is returned. The format of the timestamp is implementation-dependent.

Returned value

On success, arm_get_timestamp returns ARM_RC_SUCCESS. On failure, the errno and return code are set to indicate the error. See "Return code" on page 160 for the list of all possible return codes.

Error Code
Description

EFAULT
A parameter of this service contained an address that was not accessible to the caller.

EINVAL
A parameter of this service contained a value that was not valid.

EMVSARMERROR
An ARM error occurred. Refer to the return code for the specific error.

EPERM
The caller does not have the appropriate privileges. The return code is set to ARM_RC_AUTH_ERROR.

EMVSSAF2ERR
An error occurred in the security product.

Related information

- “arm_bind_thread() — Bind the current thread to a given transaction” on page 159
- “arm_blocked() — Indicate the processing of a transaction is blocked” on page 162
- “arm_correlator_get_length() — Get the actual size of the transaction correlator” on page 164
- “arm_end_application() — Undefines an ARM application” on page 165
- “arm_get_correlator_max_length() — Get the max length of the transaction correlator” on page 167
- “arm_init_application() — Defines an ARM application”
- “arm_init_transaction_type() — Defines and initializes an ARM transaction type” on page 171
- “arm_start_transaction() — Mark the start of an ARM transaction” on page 173
- “arm_stop_transaction() — Mark the end of an ARM transaction” on page 175
- “arm_unbind_thread() — Unbind the current thread to a given transaction” on page 177
- “arm_unblocked() — Indicate the processing of a transaction is no longer blocked” on page 179
- “arm_update_transaction() — Update a given transaction” on page 180

arm_init_application() — Defines an ARM application

Standards

Standards / Extensions	C or C++	Dependencies
	both	

Format

```
#include <sys/_Elmarm4.h>

arm_error_t arm_init_application(
/* [in]     */  arm_string_t     application_name,
/* [in]     */  arm_string_t     application_group_name,
/* [in]     */  arm_string_t     application_instance_name,
/* [in]     */  arm_int32_t      flags,
/* [in]     */  arm_buffer4_t   *buffer4,
/* [in/out] */  arm_appl_id_t   *application_id
);
```

General description

Use arm_init_application() to define an application. This function must be called before any other ARM API calls. Typically it should be called during the application's initialization.

If an application exists as multiple processes, it must call arm_init_application() in each process. This is necessary for eWLM to understand how each process contributes to the application. Multiple applications can register from within one process. This is useful if external functions are packaged in the same process.

arm_init_application() is the ARM 4.0 equivalent of ARM 2.0's arm_init function.

application_name
The name used generically to identify the application. The maximum length is 128 characters including the null string terminator.

application_group_name
The name of a group of application instances to which this instance belongs. The maximum length is 256 characters including the null string terminator. If no value is desired, a null pointer should be passed.

application_instance_name
The name of this instance of the application. The maximum length is 256 characters including the null string terminator. If no value is desired, a null pointer should be passed.

flags
One flag is defined. ARM_INIT_FLAG_ID_INPUT indicates whether the application id is an input (flag is one) or an output (flag is zero). See the application id parameter below. All other flag bits are reserved and must be zero.

buffer4
A pointer to a buffer that identifies one or more sub-buffers containing additional data. Currently no sub-buffers are defined for this function so a null pointer should be passed. If a buffer is passed, eWLM ignores it.

application_id
A pointer to a 64-bit area handle that identifies the application. The application can define its own handle and pass it as input or it can have eWLM generate a handle and return it. The choice is indicated by the flag ARM_INIT_FLAG_ID_INPUT. The handle must be passed to arm_init_transaction_type() and arm_end_application(). The handle is defined only within the caller's process.

Returned value

On success, arm_init_application returns ARM_RC_SUCCESS. On failure, the errno and return code are set to indicate the error. See “Return code” on page 160 for the list of all possible return codes. On failure, the application_id is set to a dummy value which can be used for later calls to other interfaces. Those interfaces will recognize that the application_id contains a dummy value and return without performing any action.

Error Code
Description

EFAULT
A parameter of this service contained an address that was not accessible to the caller.

EINVAL
A parameter of this service contained a value that was not valid.

EMVSARMERROR
An ARM error occurred. Refer to the return code for the specific error.

EPERM
The caller does not have the appropriate privileges. The return code is set to ARM_RC_AUTH_ERROR.

EMVSSAF2ERR
An error occurred in the security product.

Related information

- "arm_bind_thread() — Bind the current thread to a given transaction" on page 159
- "arm_blocked() — Indicate the processing of a transaction is blocked" on page 162
- "arm_correlator_get_length() — Get the actual size of the transaction correlator" on page 164
- "arm_end_application() — Undefines an ARM application" on page 165
- "arm_get_correlator_max_length() — Get the max length of the transaction correlator" on page 167
- "arm_get_timestamp() — Get the current timestamp" on page 168
- "arm_init_transaction_type() — Defines and initializes an ARM transaction type"
- "arm_start_transaction() — Mark the start of an ARM transaction" on page 173
- "arm_stop_transaction() — Mark the end of an ARM transaction" on page 175
- "arm_unbind_thread() — Unbind the current thread to a given transaction" on page 177
- "arm_unblocked() — Indicate the processing of a transaction is no longer blocked" on page 179
- "arm_update_transaction() — Update a given transaction" on page 180

arm_init_transaction_type() — Defines and initializes an ARM transaction type

Standards

Standards / Extensions	C or C++	Dependencies
	both	

Format

```
#include <sys/_Elmarm4.h>

arm_error_t arm_init_transaction_type(
/* [in]     */  arm_appl_id_t   *application_id,
/* [in]     */  arm_string_t     transaction_name,
/* [in]     */  arm_int32_t      flags,
/* [in]     */  arm_buffer4_t   *buffer4,
/* [in/out] */  arm_appl_id_t   *transaction_type_id
);
```

General description

Use arm_init_transaction_type() to define a transaction classtype. A transaction type represents a type of transaction that is executed by an application. A transaction type must be defined before transactions of that type can be measured using arm_start_transaction() and arm_stop_transaction().

A transaction type consists of a transaction type name and a list of up to 20 transaction property names. Multiple transaction types with the same transaction type name and different transaction property names are allowed. It is the complete set of names that determines the uniqueness of a transaction type.

It is recommended that transaction types be defined during the application's initialization. eWLM expects transaction types to represent broad categories of work so that ordinarily there will be few of them. Detailed transaction identification should be done using the transaction-level properties.

Transaction types also can be defined as needed, during the application's processing of transactions. The ARM implementation must check whether there is an existing definition of the same type to prevent duplicates. Since this adds further overhead into mainline processing, this approach is not recommended.

Transaction classtype definitions remain valid for the life of the application (i.e. until it calls arm_end_application() or its process terminates).

arm_init_application_type() is the ARM 4.0 equivalent of ARM 2.0's arm_getid function.

application_id
The handle passed to or returned by arm_init_application() for the application associated with this transaction type.

transaction_name
The name used to identify the transaction type. The maximum length is 128 characters including the null string terminator. Classification rules in the eWLM policy can use the transaction type name to categorize transactions into eWLM service classes and report classes.

flags
One flag is defined. ARM_INIT_FLAG_ID_INPUT indicates whether the transaction type id is an input (flag is one) or an output (flag is zero). See the transaction type id parameter below. All other flag bits are reserved and must be zero.

buffer4
A pointer to a buffer that identifies one or more sub-buffers containing additional data. A null value can be specified if no additional data is required.

transaction_type_id
A pointer to a 64-bit handle that identifies the transaction type. The application can define its own handle and pass it as input or it can have eWLM generate a handle and return it. The choice is indicated by the flag ARM_INIT_FLAG_ID_INPUT. The handle must be passed to arm_start_transaction(). The handle is defined only within the caller's process. If eWLM is asked to generate a handle and the transaction type name and the transaction property names (in the user data buffer) match the names passed on a previous arm_init_transaction_type() call, the handle value that is returned will be the same as the previous call.

Returned value

On success, arm_init_application returns ARM_RC_SUCCESS. On failure, the errno and return code are set to indicate the error. See “Return code” on page 160 for the list of all possible return codes. On failure, the application_id is set to a dummy value which can be used for later calls to other interfaces. Those interfaces will recognize that the application_id contains a dummy value and return without performing any action.

Error Code
Description

EFAULT
A parameter of this service contained an address that was not accessible to the caller.

EINVAL
A parameter of this service contained a value that was not valid.

EMVSARMERROR
An ARM error occurred. Refer to the return code for the specific error.

EPERM
The caller does not have the appropriate privileges. The return code is set to ARM_RC_AUTH_ERROR.

EMVSSAF2ERR
An error occurred in the security product.

Related information

- “arm_bind_thread() — Bind the current thread to a given transaction” on page 159
- “arm_blocked() — Indicate the processing of a transaction is blocked” on page 162
- “arm_correlator_get_length() — Get the actual size of the transaction correlator” on page 164
- “arm_end_application() — Undefines an ARM application” on page 165
- “arm_get_correlator_max_length() — Get the max length of the transaction correlator” on page 167
- “arm_get_timestamp() — Get the current timestamp” on page 168
- “arm_init_application() — Defines an ARM application” on page 169
- “arm_start_transaction() — Mark the start of an ARM transaction”
- “arm_stop_transaction() — Mark the end of an ARM transaction” on page 175
- “arm_unbind_thread() — Unbind the current thread to a given transaction” on page 177
- “arm_unblocked() — Indicate the processing of a transaction is no longer blocked” on page 179
- “arm_update_transaction() — Update a given transaction” on page 180

arm_start_transaction() — Mark the start of an ARM transaction

Standards

Standards / Extensions	C or C++	Dependencies
	both	

Format

```
#include <sys/_Elmarm4.h>

arm_error_t arm_start_transaction(
/* [in]     */  arm_appl_id_t     *transaction_id,
/* [in]     */  arm_correlator_t     *parent_correlator_ptr,
/* [in]     */  arm_int32_t      flags,
/* [in]     */  arm_buffer4_t   *buffer4,
/* [out]    */  arm_start_handle_t     *start_handle,
/* [out]    */  arm_correlator_t   *current_correlator_ptr
);
```

General description

Use arm_start_transaction() to mark the beginning of execution of a transaction. The transaction must be identified as a member of a transaction type that was previously defined by arm_init_transaction_type(). There can be any number of transactions executing simultaneously.

transaction_id
The handle passed to or returned by arm_init_transaction_type() for the transaction type.

parent_correlator_ptr
A pointer to the parent correlator for this transaction. If there is no parent correlator, a null value should be passed.

flags
One flag bit is defined:. ARM_START_FLAG_TRACE_REQUEST requests tracing of the transaction. Currently eWLM has no support to trace transactions and ignores the flag. All other flag bits are reserved and must be zero.

buffer4
A pointer to a buffer that identifies one or more sub-buffers with additional data. A null value can be specified if no additional data is required.

start_handle
A pointer to a 64-bit area where a handle that identifies the transaction is returned. The handle must be passed to other transaction-level interfaces such as arm_stop_transaction(). The handle is valid only within the caller's process.

current_correlator_ptr
A pointer to a buffer into which eWLM will store a correlator for the transaction. The length of the buffer must be at least the length returned from the arm_get_correlator_max_length() function.

Returned value

On success, arm_init_application returns ARM_RC_SUCCESS. On failure, the errno and return code are set to indicate the error. See “Return code” on page 160 for the list of all possible return codes. On failure, the application_id is set to a dummy value which can be used for later calls to other interfaces. Those interfaces will recognize that the application_id contains a dummy value and return without performing any action.

Error Code
Description

EFAULT
A parameter of this service contained an address that was not accessible to the caller.

EINVAL
A parameter of this service contained a value that was not valid.

EMVSARMERROR
An ARM error occurred. Refer to the return code for the specific error.

EPERM
The caller does not have the appropriate privileges. The return code is set to ARM_RC_AUTH_ERROR.

EMVSSAF2ERR
An error occurred in the security product.

Related information

- "arm_bind_thread() — Bind the current thread to a given transaction" on page 159
- "arm_blocked() — Indicate the processing of a transaction is blocked" on page 162
- "arm_correlator_get_length() — Get the actual size of the transaction correlator" on page 164
- "arm_end_application() — Undefines an ARM application" on page 165
- "arm_get_correlator_max_length() — Get the max length of the transaction correlator" on page 167
- "arm_get_timestamp() — Get the current timestamp" on page 168
- "arm_init_application() — Defines an ARM application" on page 169
- "arm_init_transaction_type() — Defines and initializes an ARM transaction type" on page 171
- "arm_stop_transaction() — Mark the end of an ARM transaction"
- "arm_unbind_thread() — Unbind the current thread to a given transaction" on page 177
- "arm_unblocked() — Indicate the processing of a transaction is no longer blocked" on page 179
- "arm_update_transaction() — Update a given transaction" on page 180

arm_stop_transaction() — Mark the end of an ARM transaction

Standards

Standards / Extensions	C or C++	Dependencies
	both	

Format

```
#include <sys/_Elmarm4.h>

arm_error_t arm_stop_transaction(
/* [in]     */  arm_start_handle_t     start_handle,
/* [in]     */  arm_int32_t     transaction_status,
/* [in]     */  arm_int32_t      flags,
/* [in]     */  arm_buffer4_t   *buffer4,
);
```

General description

Use arm_stop_transaction() to mark the end of execution of a transaction.

If any threads are bound to the transaction, arm_stop_transaction() unbinds them. If the transaction is in a blocked state due to outstanding arm_blocked() calls, then arm_stop_transaction() considers the transaction to be unblocked at the time it is stopped. This cleanup processing exists for recovery purposes. Applications are expected to use arm_unbind_thread() and arm_unblocked() as part of normal processing.

start_handle
The handle returned by arm_start_transaction() for the transaction.

transaction_status
The completion code of the transaction.

ARM_STATUS_GOOD
Transaction successful.

ARM_STATUS_ABORT
Transaction aborted This value indicates there was a fundamental failure in the system; for example, a communications timeout or a database operation error.

ARM_STATUS_FAILED
Transaction failed. This value indicates the system worked properly but the transaction was not successful; for example, when making an airline reservation, no seats are available on the requested flight.

ARM_STATUS_UNKNOWN
Transaction status is unknown.

.

flags
Reserved for future use. The argument must be set to 0.

buffer4
A pointer to a buffer that identifies one or more sub-buffers containing additional data. A null pointer can be passed if no additional data is required. The proposed ARM4 standard defines a metric values sub-buffer for this function but eWLM does not process metrics. If a buffer is passed eWLM ignores it.

Returned value

On success, arm_end_application returns ARM_RC_SUCCESS. On failure, the errno and return code are set to indicate the error. See “Return code” on page 160 for the list of all possible return codes.

Error Code
Description

EFAULT
A parameter of this service contained an address that was not accessible to the caller.

EINVAL
A parameter of this service contained a value that was not valid.

EMVSARMERROR
An ARM error occurred. Refer to the return code for the specific error.

EPERM
The caller does not have the appropriate privileges. The return code is set to ARM_RC_AUTH_ERROR.

EMVSSAF2ERR
An error occurred in the security product.

Related information

- "arm_bind_thread() — Bind the current thread to a given transaction" on page 159
- "arm_blocked() — Indicate the processing of a transaction is blocked" on page 162
- "arm_correlator_get_length() — Get the actual size of the transaction correlator" on page 164
- "arm_end_application() — Undefines an ARM application" on page 165
- "arm_get_correlator_max_length() — Get the max length of the transaction correlator" on page 167
- "arm_get_timestamp() — Get the current timestamp" on page 168
- "arm_init_application() — Defines an ARM application" on page 169
- "arm_init_transaction_type() — Defines and initializes an ARM transaction type" on page 171
- "arm_start_transaction() — Mark the start of an ARM transaction" on page 173
- "arm_unbind_thread() — Unbind the current thread to a given transaction"
- "arm_unblocked() — Indicate the processing of a transaction is no longer blocked" on page 179
- "arm_update_transaction() — Update a given transaction" on page 180

arm_unbind_thread() — Unbind the current thread to a given transaction

Standards

Standards / Extensions	C or C++	Dependencies
	both	

Format

```
#include <sys/_Elmarm4.h>

arm_error_t arm_unbind_thread(
/* [in]     */  arm_start_handle_t     start_handle,
/* [in]     */  arm_int32_t      flags,
/* [in]     */  arm_buffer4_t   *buffer4,
);
```

General description

Use arm_unbind_thread() to indicate the current thread is no longer performing processing on behalf of a given transaction.

If arm_unbind_thread() is not called, then arm_stop_transaction() unbinds any threads that remain bound to the transaction. This exists for recovery purposes. Applications are expected to use arm_unbind_thread() as part of normal processing.

start_handle
The handle returned by arm_start_transaction() for the transaction.

flags
Reserved for future use. The argument must be set to 0.

buffer4
A pointer to a buffer that identifies one or more sub-buffers containing additional data. Currently no sub-buffers are defined for this function so a null pointer should be passed. If a buffer is passed eWLM ignores it.

Returned value

On success, arm_unbind_thread returns ARM_RC_SUCCESS. On failure, the errno and return code are set to indicate the error. See “Return code” on page 160 for the list of all possible return codes.

Error Code
Description

EFAULT
A parameter of this service contained an address that was not accessible to the caller.

EINVAL
A parameter of this service contained a value that was not valid.

EMVSARMERROR
An ARM error occurred. Refer to the return code for the specific error.

EPERM
The caller does not have the appropriate privileges. The return code is set to ARM_RC_AUTH_ERROR.

EMVSSAF2ERR
An error occurred in the security product.

Related information

- “arm_bind_thread() — Bind the current thread to a given transaction” on page 159
- “arm_blocked() — Indicate the processing of a transaction is blocked” on page 162
- “arm_correlator_get_length() — Get the actual size of the transaction correlator” on page 164
- “arm_end_application() — Undefines an ARM application” on page 165
- “arm_get_correlator_max_length() — Get the max length of the transaction correlator” on page 167
- “arm_get_timestamp() — Get the current timestamp” on page 168
- “arm_init_application() — Defines an ARM application” on page 169
- “arm_init_transaction_type() — Defines and initializes an ARM transaction type” on page 171
- “arm_start_transaction() — Mark the start of an ARM transaction” on page 173
- “arm_stop_transaction() — Mark the end of an ARM transaction” on page 175

- "arm_unblocked() — Indicate the processing of a transaction is no longer blocked"
- "arm_update_transaction() — Update a given transaction" on page 180

arm_unblocked() — Indicate the processing of a transaction is no longer blocked

Standards

Standards / Extensions	C or C++	Dependencies
	both	

Format

```
#include <sys/_Elmarm4.h>

arm_error_t arm_unblocked(
/* [in]     */  arm_start_handle_t     start_handle,
/* [in]     */  arm_block_handle_t     block_handle,
/* [in]     */  arm_int32_t      flags,
/* [in]     */  arm_buffer4_t   *buffer4,
);
```

General description

Use arm_unblocked() to indicate that processing of a transaction is no longer blocked.

If arm_unblocked() is not called, then arm_stop_transaction() considers the transaction to be unblocked at the time it is stopped. This exists for recovery purposes. Applications are expected to use arm_unblocked() as part of normal processing.

start_handle
The handle returned by arm_startarm_start_transaction() for the transaction.

block_handle
The handle returned by arm_blocked().

flags
Reserved for future use. The argument must be set to 0.

buffer4
A pointer to a buffer that identifies one or more sub-buffers containing additional data. Currently no sub-buffers are defined for this function so a null pointer should be passed. If a buffer is passed eWLM ignores it.

Returned value

On success, arm_unblocked returns ARM_RC_SUCCESS. On failure, the errno and return code are set to indicate the error. See "Return code" on page 160 for the list of all possible return codes.

Error Code
Description

EFAULT
A parameter of this service contained an address that was not accessible to the caller.

EINVAL
A parameter of this service contained a value that was not valid.

EMVSARMERROR
An ARM error occurred. Refer to the return code for the specific error.

EPERM
The caller does not have the appropriate privileges. The return code is set to ARM_RC_AUTH_ERROR.

EMVSSAF2ERR
An error occurred in the security product.

Related information

- “arm_bind_thread() — Bind the current thread to a given transaction” on page 159
- “arm_blocked() — Indicate the processing of a transaction is blocked” on page 162
- “arm_correlator_get_length() — Get the actual size of the transaction correlator” on page 164
- “arm_end_application() — Undefines an ARM application” on page 165
- “arm_get_correlator_max_length() — Get the max length of the transaction correlator” on page 167
- “arm_get_timestamp() — Get the current timestamp” on page 168
- “arm_init_application() — Defines an ARM application” on page 169
- “arm_init_transaction_type() — Defines and initializes an ARM transaction type” on page 171
- “arm_start_transaction() — Mark the start of an ARM transaction” on page 173
- “arm_stop_transaction() — Mark the end of an ARM transaction” on page 175
- “arm_unbind_thread() — Unbind the current thread to a given transaction” on page 177
- “arm_update_transaction() — Update a given transaction”

arm_update_transaction() — Update a given transaction

Standards

Standards / Extensions	C or C++	Dependencies
	both	

Format

```
#include <sys/_Elmarm4.h>

arm_error_t arm_update_transaction(
/* [in]     */  arm_start_handle_t     start_handle,
/* [in]     */  arm_int32_t      flags,
/* [in]     */  arm_buffer4_t   *buffer4,
);
```

General description

arm_update_transaction() does not perform any function in the eWLM developer's edition. A future edition of eWLM may support a version of arm_update_transaction() that can be used to log problem determination data associated with the transaction.

`start_handle`
The handle returned by arm_start_transaction() for the transaction.

`flags`
Reserved for future use. The argument must be set to 0.

`buffer4`
A pointer to a buffer that identifies one or more sub-buffers containing additional data. A null pointer can be passed if no additional data is required. The proposed ARM4 standard defines a metric values sub-buffer for this function but eWLM does not process metrics. If a buffer is passed eWLM ignores it.

Returned value

On success, arm_update_transaction returns ARM_RC_SUCCESS. On failure, the errno and return code are set to indicate the error. See “Return code” on page 160 for the list of all possible return codes.

Error Code
Description

EFAULT
A parameter of this service contained an address that was not accessible to the caller.

EINVAL
A parameter of this service contained a value that was not valid.

EMVSARMERROR
An ARM error occurred. Refer to the return code for the specific error.

EPERM
The caller does not have the appropriate privileges. The return code is set to ARM_RC_AUTH_ERROR.

EMVSSAF2ERR
An error occurred in the security product.

Related information

- “arm_bind_thread() — Bind the current thread to a given transaction” on page 159
- “arm_blocked() — Indicate the processing of a transaction is blocked” on page 162
- “arm_correlator_get_length() — Get the actual size of the transaction correlator” on page 164
- “arm_end_application() — Undefines an ARM application” on page 165
- “arm_get_correlator_max_length() — Get the max length of the transaction correlator” on page 167
- “arm_get_timestamp() — Get the current timestamp” on page 168
- “arm_init_application() — Defines an ARM application” on page 169

- “arm_init_transaction_type() — Defines and initializes an ARM transaction type” on page 171
- “arm_start_transaction() — Mark the start of an ARM transaction” on page 173
- “arm_stop_transaction() — Mark the end of an ARM transaction” on page 175
- “arm_unbind_thread() — Unbind the current thread to a given transaction” on page 177
- “arm_unblocked() — Indicate the processing of a transaction is no longer blocked” on page 179

asctime(), asctime64() — Convert time to character string

Standards

Standards / Extensions	C or C++	Dependencies
ISO C POSIX.1 XPG4 XPG4.2 C99 Single UNIX Specification, Version 3 Language Environment	both	

Format

```
#include <time.h>

char *asctime(const struct tm *timeptr);
```

```
#define _LARGE_TIME_API
#include <time.h>

char asctime64(const struct tm *timeptr);
```

General description

Converts time stored as a structure, pointed to by *timeptr*, to a character string. The *timeptr* value can be obtained from a call to gmtime() or localtime(). Both functions return a pointer to a tm structure defined in “time.h” on page 79.

The string result that asctime() produces contains exactly 26 characters and has the format:

```
"%.3s %.3s%3d %.2d:%.2d:%.2d %d\n"
```

The following is an example of the string returned:

```
Fri Jun 16 02:03:55 2006\n\0
```

Notes:

1. The calendar time returned by a call to the time() function begins at epoch, which was at 00:00:00 Coordinated Universal Time (UTC), January 1, 1970.
2. The asctime() function uses a 24-hour clock format.
3. The days are abbreviated to: Sun, Mon, Tue, Wed, Thu, Fri, and Sat.
4. The months are abbreviated to: Jan, Feb, Mar, Apr, May, Jun, Jul, Aug, Sep, Oct, Nov, and Dec.
5. All fields have a constant width.
6. Dates with only one digit are preceded either with a zero or a blank space.

7. The newline character (\n) and the NULL character (\0) occupy the last two positions of the string.
8. The asctime(), ctime(), and other time functions can use a common, statically allocated buffer for holding the return string. Each call to one of these functions may possibly destroy the result of the previous call.

The function asctime64() will behave exactly like asctime() except it will support a structured date beyond 03:14:07 UTC on January 19, 2038 with a limit of 23:59:59 UTC on December 31, 9999.

Returned value

If successful, asctime() returns a pointer to the resulting character string.

If the function is unsuccessful, it returns NULL.

Example

CELEBA06

```
/* CELEBA06

   This example polls the system clock and prints a message
   giving the current time.

 */
#include <time.h>
#include <stdio.h>

int main(void)
{
    struct tm *newtime;
    time_t ltime;

        /* Get the time in seconds */
    time(&ltime);
        /* Break it down & store it in the structure tm */
    newtime = localtime(&ltime);

        /* Print the local time as a string */
    printf("The current date and time are %s",
             asctime(newtime));
}
```

Output

```
The current date and time are Fri Jun 16 13:29:51 2006
```

Related information

- "time.h" on page 79
- "asctime_r(), asctime64_r() — Convert date and time to a character string" on page 184
- "ctime(), ctime64() — Convert time to character string" on page 359
- "ctime_r(), ctime64_r() — Convert time value to date and time character string" on page 362
- "gmtime(), gmtime64() — Convert time to broken-down UTC time" on page 810
- "gmtime_r(), gmtime64_r() — Convert a time value to broken-down UTC time" on page 812
- "localdtconv() — Date and time formatting convention inquiry" on page 981
- "localtime(), localtime64() — Convert time and correct for local time" on page 984

- "localtime_r(), localtime64_r() — Convert time value to broken-down local time" on page 986
- "mktime(), mktime64() — Convert local time" on page 1078
- "strftime() — Convert to formatted time" on page 1743
- "time(),time64() — Determine current UTC time" on page 1878
- "tzset() — Set the time zone" on page 1936

asctime_r(), asctime64_r() — Convert date and time to a character string

Standards

Standards / Extensions	C or C++	Dependencies
Single UNIX Specification, Version 2 Single UNIX Specification, Version 3 Language Environment	both	OS/390 V2R8

Format

```
#define _XOPEN_SOURCE 500
#include <time.h>

char *asctime_r(const struct tm *__restrict__ tm, char *__restrict__ buf);
#define _LARGE_TIME_API
#include <time.h>

char *asctime64_r(const struct tm *__restrict__ tm, char *__restrict__ buf);
```

General description

The asctime_r() function converts the broken-down time in the structure pointed to by *tm* into a character string that is placed in the user-supplied buffer pointed to by *buf* (which contains at least 26 bytes) and then returns *buf*.

The function asctime64_r() will behave exactly like asctime_r() except it will support a structured date beyond 03:14:07 UTC on January 19, 2038 with a limit of 23:59:59 UTC on December 31, 9999.

Returned value

If successful, asctime_r() returns a pointer to a character string containing the date and time. This string is pointed to by the argument *buf*.

If unsuccessful, asctime_r() returns NULL.

There are no documented errno values.

Related information

- "time.h" on page 79
- "asctime(), asctime64() — Convert time to character string" on page 182
- "ctime(), ctime64() — Convert time to character string" on page 359
- "ctime_r(), ctime64_r() — Convert time value to date and time character string" on page 362
- "gmtime(), gmtime64() — Convert time to broken-down UTC time" on page 810
- "gmtime_r(), gmtime64_r() — Convert a time value to broken-down UTC time" on page 812

- "localdtconv() — Date and time formatting convention inquiry" on page 981
- "localtime(), localtime64() — Convert time and correct for local time" on page 984
- "localtime_r(), localtime64_r() — Convert time value to broken-down local time" on page 986
- "mktime(), mktime64() — Convert local time" on page 1078
- "strftime() — Convert to formatted time" on page 1743
- "time(),time64() — Determine current UTC time" on page 1878
- "tzset() — Set the time zone" on page 1936

asin(), asinf(), asinl() — Calculate arcsine

Standards

Standards / Extensions	C or C++	Dependencies
ISO C POSIX.1 XPG4 XPG4.2 ISO/ANSI C++ C99 Single UNIX Specification, Version 3 C++ TR1 C99	both	

Format

```
#include <math.h>

double asin(double x);
float asin(float x);                  /* C++ only */
long double asin(long double x);      /* C++ only */
float asinf(float x);
long double asinl(long double x);
```

General description

Calculates the arcsine of *x*, in the range -pi/2 to pi/2 radians.

The value of *x* must be between -1 and 1.

Note: These functions work in both IEEE Binary Floating-Point and hexadecimal floating-point formats. See "IEEE binary floating-point" on page 94 for more information about IEEE Binary Floating-Point.

Returned value

If *x* is less than -1 or greater than 1, the function sets errno to EDOM, and returns 0. Otherwise, it returns a nonzero value.

If the correct value would cause an underflow, 0 is returned and the value ERANGE is stored in errno.

Special behavior for IEEE: If successful, the function returns the arcsine of its argument *x*.

If *x* is less than -1 or greater than 1, the function sets errno to EDOM and returns NaNQ. No other errors will occur.

Example

CELEBA07

```
/* CELEBA07

   This example prompts for a value for x.
   It prints an error message if x is greater than 1 or
   less than -1; otherwise, it assigns the arcsine of
   x to y.

 */
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#define MAX  1.0
#define MIN -1.0

int main(void)
{
  double x, y;

  printf( "Enter x\n" );
  scanf( "%lf", &x );

  /* Output error if not in range */
  if ( x > MAX )
    printf( "Error: %f too large for asin\n", x );
  else if ( x < MIN )
    printf( "Error: %f too small for asin\n", x );
  else {
    y = asin( x );
    printf( "asin( %f ) = %f\n", x, y );
  }
}
```

Output

```
Enter x
 0.2 is entered
asin( 0.200000 ) = 0.201358
```

Related information

- "math.h" on page 44
- "acos(), acosf(), acosl() — Calculate arccosine" on page 135
- "acosh(), acoshf(), acoshl() — Calculate hyperbolic arccosine" on page 139
- "asinh(), asinhf(), asinhl() — Calculate hyperbolic arcsine" on page 188
- "atan(), atanf(), atanl(), atan2(), atan2f(), atan2l() — Calculate arctangent" on page 192
- "atanh(), atanhf(), atanhl() — Calculate hyperbolic arctangent" on page 195
- "cos(), cosf(), cosl() — Calculate cosine" on page 328
- "cosh(), coshf(), coshl() — Calculate hyperbolic cosine" on page 331
- "sin(), sinf(), sinl() — Calculate sine" on page 1672
- "sinh(), sinhf(), sinhl() — Calculate hyperbolic sine" on page 1674
- "tan(), tanf(), tanl() — Calculate tangent" on page 1815
- "tanh(), tanhf(), tanhl() — Calculate hyperbolic tangent" on page 1818

asind32(), asind64(), asind128() - Calculate arcsine

Standards

Standards / Extensions	C or C++	Dependencies
C/C++ DFP	both	z/OS V1.10

Format

```
#define __STDC_WANT_DEC_FP__
#include <math.h>

_Decimal32  asind32(_Decimal32 x);
_Decimal64  asind64(_Decimal64 x);
_Decimal128 asind128(_Decimal128 x);
_Decimal32  asin(_Decimal32 x);      /* C++ only */
_Decimal64  asin(_Decimal64 x);      /* C++ only */
_Decimal128 asin(_Decimal128 x);     /* C++ only */
```

General description

Calculates the arcsine of *x*, in the range -pi/2 to pi/2 radians.

The value of *x* must be between -1 and 1.

These functions work in IEEE decimal floating-point format. See “IEEE decimal floating-point” on page 95 for more information.

Note: To use IEEE decimal floating-point, the hardware must have the Decimal Floating-Point Facility installed.

Returned value

If successful, the function returns the arcsine of its argument x.

If *x* is less than -1 or greater than 1, the function sets errno to EDOM and returns NaNQ. No other errors will occur.

Example

CELEBA13

```
/* CELEBA13

   This example illustrates the asind128() function.

   This example prompts for a value for x.
   It prints an error message if x is greater than 1 or
   less than -1; otherwise, it assigns the arcsine of
   x to y.

*/

#define  __STDC_WANT_DEC_FP__
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#define MAX  1.0DL
#define MIN -1.0DL

int main(void)
```

```
{
   _Decimal128 x, y;

   printf( "Enter x\n" );
   scanf( "%DDf", &x );

   /* Output error if not in range */
   if ( x > MAX )
      printf( "Error: %f too large for asind128\n", x );
   else if ( x < MIN )
      printf( "Error: %f too small for asind128\n", x );
   else {
      y = asind128( x );
      printf( "asind128( %DDf ) = %DDf\n", x, y );
   }
}
```

Related information

- “math.h” on page 44
- “acosd32(), acosd64(), acosd128() - Calculate arccosine” on page 137
- “acoshd32(), acoshd64(), acoshd128() - Calculate hyperbolic arccosine” on page 140
- “asinhd32(), asinhd64(), asinhd128() - Calculate hyperbolic arcsine” on page 189
- “atand32(), atand64(), atand128(), atan2d32(), atan2d64(), atan2d128() - Calculate arctangent” on page 194
- “atanhd32(), atanhd64(), atanhd128() - Calculate hyperbolic arctangent” on page 197
- “cosd32(), cosd64(), cosd128() — Calculate cosine” on page 329
- “coshd32(), coshd64(), coshd128() - Calculate hyperbolic cosine” on page 332
- “sind32(), sind64(), sind128() — Calculate sine” on page 1673
- “sinhd32(), sinhd64(), sinhd128() - Calculate hyperbolic sine” on page 1676
- “tand32(), tand64(), tand128() - Calculate tangent” on page 1817
- “tanhd32(), tanhd64(), tanhd128() - Calculate hyperbolic tangent” on page 1820

asinh(), asinhf(), asinhl() — Calculate hyperbolic arcsine

Standards

Standards / Extensions	C or C++	Dependencies
XPG4.2 C99 Single UNIX Specification, Version 3 C++ TR1 C99	both	

Format

```
#define _XOPEN_SOURCE_EXTENDED 1
#include <math.h>

double asinh(double x);
```

C99:

```
#define _ISOC99_SOURCE
#include <math.h>

float asinhf(float x);
long double asinhl(long double x);
```

C++ TR1 C99:

```
#define _TR1_C99
#include <math.h>

float asinh(float x);
long double asinh(long double x);
```

General description

The asinh() functions return the hyperbolic arcsine of its argument *x*.

Note: The following table shows the viable formats for these functions. See "IEEE binary floating-point" on page 94 for more information about IEEE Binary Floating-Point.

Function	Hex	IEEE
asinh	X	X
asinhf	X	X
asinhl	X	X

Returned value

asinh() returns the hyperbolic arcsine of its argument *x*. The function is always successful.

Related information

- "math.h" on page 44
- "acos(), acosf(), acosl() — Calculate arccosine" on page 135
- "acosh(), acoshf(), acoshl() — Calculate hyperbolic arccosine" on page 139
- "asin(), asinf(), asinl() — Calculate arcsine" on page 185
- "atan(), atanf(), atanl(), atan2(), atan2f(), atan2l() — Calculate arctangent" on page 192
- "atanh(), atanhf(), atanhl() — Calculate hyperbolic arctangent" on page 195
- "cos(), cosf(), cosl() — Calculate cosine" on page 328
- "cosh(), coshf(), coshl() — Calculate hyperbolic cosine" on page 331
- "sin(), sinf(), sinl() — Calculate sine" on page 1672
- "sinh(), sinhf(), sinhl() — Calculate hyperbolic sine" on page 1674
- "tan(), tanf(), tanl() — Calculate tangent" on page 1815
- "tanh(), tanhf(), tanhl() — Calculate hyperbolic tangent" on page 1818

asinhd32(), asinhd64(), asinhd128() - Calculate hyperbolic arcsine

Standards

Standards / Extensions	C or C++	Dependencies
C/C++ DFP	both	z/OS V1.10

Format

```
#define __STDC_WANT_DEC_FP__
#include <math.h>

_Decimal32  asinhd32(_Decimal32 x);
_Decimal64  asinhd64(_Decimal64 x);
_Decimal128 asinhd128(_Decimal128 x);
```

```
_Decimal32  asinh(_Decimal32 x);     /* C++ only */
_Decimal64  asinh(_Decimal64 x);     /* C++ only */
_Decimal128 asinh(_Decimal128 x);    /* C++ only */
```

General description

The asinhd() functions return the hyperbolic arcsine of its argument *x*.

These functions work in IEEE decimal floating-point format. See "IEEE decimal floating-point" on page 95 for more information.

Note: To use IEEE decimal floating-point, the hardware must have the Decimal Floating-Point Facility installed.

Returned value

asinhd() returns the hyperbolic arcsine of its argument *x*. The function is always successful.

Example

CELEBA14

```
/* CELEBA14

   This example illustrates the asinhd32() function.

*/

#define  __STDC_WANT_DEC_FP__
#include <math.h>
#include <stdio.h>

int main(void)
{
   _Decimal32 x, y;

   x = 1.0DF;
   y = asinhd32(x);

   printf("asinhd32(%Hf) = %Hf\n", x, y);
}
```

Related information

- "math.h" on page 44
- "acosd32(), acosd64(), acosd128() - Calculate arccosine" on page 137
- "acoshd32(), acoshd64(), acoshd128() - Calculate hyperbolic arccosine" on page 140
- "asind32(), asind64(), asind128() - Calculate arcsine" on page 187
- "atand32(), atand64(), atand128(), atan2d32(), atan2d64(), atan2d128() - Calculate arctangent" on page 194
- "atanhd32(), atanhd64(), atanhd128() - Calculate hyperbolic arctangent" on page 197
- "cosd32(), cosd64(), cosd128() — Calculate cosine" on page 329
- "coshd32(), coshd64(), coshd128() - Calculate hyperbolic cosine" on page 332
- "sind32(), sind64(), sind128() — Calculate sine" on page 1673
- "sinhd32(), sinhd64(), sinhd128() - Calculate hyperbolic sine" on page 1676
- "tand32(), tand64(), tand128() - Calculate tangent" on page 1817
- "tanhd32(), tanhd64(), tanhd128() - Calculate hyperbolic tangent" on page 1820

assert() — Verify condition

Standards

Standards / Extensions	C or C++	Dependencies
ISO C POSIX.1 XPG4 XPG4.2 C99 Single UNIX Specification, Version 3	both	

Format

```
#include <assert.h>

void assert(int expression);
```

General description

The assert() macro inserts diagnostics into a program. If the expression (which will have a scalar type) is false (that is, compares equal to 0), a diagnostic message of the form shown below is printed to stderr, and abort() is called to abnormally end the program. The assert() macro takes no action if the expression is true (nonzero).

Without a compiler that is designed to support C99, or when compiling C++ code, the diagnostic message has the following format:

```
Assertion failed: expression, file filename, line line-number.
```

With a compiler that is designed to support C99, or when compiling C++ code, the diagnostic message has the following format:

```
Assertion failed: expression, file filename, line line-number, function function-name.
```

If you define NDEBUG to any value with a `#define` directive or with the DEFINE compiler option, the C/C++ preprocessor expands all assert() invocations to `void` expressions.

Note: The assert() function is a macro. Using the `#undef` directive with the assert() macro results in undefined behavior. The assert() macro uses `__FILE__`, `__LINE__` and, with a compiler that is designed to support C99, __func__...

Returned value

assert() returns no values.

Example

CELEBA08

```
/* CELEBA08

   In this example, the assert() macro tests the string argument for a
   null string and an empty string, and verifies that the length argument
   is positive before proceeding.

 */
#include <stdio.h>
#include <assert.h>
```

```
void analyze(char *, int);
int main(void)
{
   char *string1 = "ABC";
   char *string2 = "";
   int length = 3;

   analyze(string1, length);
   printf("string1 %s is not null or empty, "
          "and has length %d \n", string1, length);
   analyze(string2, length);
   printf("string2 %s is not null or empty,"
          "and has length %d\n", string2, length);
}

void analyze(char *string, int length)
{
   assert(string != NULL);     /* cannot be NULL */
   assert(*string != '\0');    /* cannot be empty */
   assert(length > 0);         /* must be positive */
}
```

Output without a compiler that is designed to support C99

```
String1 ABC is not NULL or empty, and has length 3
Assertion failed: *string != '\0', file: CELEBA08 C       A1, line: 26
```

Output with a compiler that is designed to support C99

```
String1 ABC is not NULL or empty, and has length 3
Assertion failed: *string != '\0', file: CELEBA08 C       A1, line: 26 in function analyze
```

Related information

- "assert.h" on page 17
- "abort() — Stop a program" on page 101

atan(), atanf(), atanl(), atan2(), atan2f(), atan2l() — Calculate arctangent

Standards

Standards / Extensions	C or C++	Dependencies
ISO C POSIX.1 XPG4 XPG4.2 ISO/ANSI C++ C99 Single UNIX Specification, Version 3 C++ TR1 C99	both	

Format

```
#include <math.h>

double atan(double x);
float atan(float x);                  /* C++ only */
long double atan(long double x);      /* C++ only */
float atanf(float x);
long double atanl(long double x);

double atan2(double y, double x);
float atan2(float y, float x);                    /* C++ only */
```

```
long double atan2(long double y, long double x);  /* C++ only */
float atan2f(float y, float x);
long double atan2l(long double y, long double x);
```

General description

The atan() and atan2() functions calculate the arctangent of *x* and *y/x*, respectively.

Note: These functions work in both IEEE Binary Floating-Point and hexadecimal floating-point formats. See “IEEE binary floating-point” on page 94 for more information about IEEE Binary Floating-Point.

Restriction: The atan2f() function does not support the _FP_MODE_VARIABLE feature test macro.

Returned value

Returns a value in the range -pi/2 to pi/2 radians.

The atan2() functions return a value in the range -pi to pi radians. If both arguments of atan2() are zero, the function sets errno to EDOM, and returns 0. If the correct value would cause underflow, zero is returned and the value ERANGE is stored in errno.

Special behavior for IEEE: If successful, atan2() returns the arctangent of *y/x*.

If both arguments of atan2() are zero, the function sets errno to EDOM and returns 0. No other errors will occur.

Example

CELEBA09

```
/* CELEBA09 */
#include <math.h>
#include <stdio.h>

int main(void)
{
    double a,b,c,d;

    c = 0.45;
    d = 0.23;

    a = atan(c);
    b = atan2(c,d);

    printf("atan( %f ) = %f\n", c, a);
    printf("atan2( %f, %f ) = %f\n", c, d, b);
}
```

Output

```
atan( 0.450000 ) = 0.422854
atan2( 0.450000, 0.230000 ) = 1.098299
```

Related information

- “math.h” on page 44
- “acos(), acosf(), acosl() — Calculate arccosine” on page 135
- “acosh(), acoshf(), acoshl() — Calculate hyperbolic arccosine” on page 139
- “asin(), asinf(), asinl() — Calculate arcsine” on page 185

- “asinh(), asinhf(), asinhl() — Calculate hyperbolic arcsine” on page 188
- “atanh(), atanhf(), atanhl() — Calculate hyperbolic arctangent” on page 195
- “cos(), cosf(), cosl() — Calculate cosine” on page 328
- “cosh(), coshf(), coshl() — Calculate hyperbolic cosine” on page 331
- “sin(), sinf(), sinl() — Calculate sine” on page 1672
- “sinh(), sinhf(), sinhl() — Calculate hyperbolic sine” on page 1674
- “tan(), tanf(), tanl() — Calculate tangent” on page 1815
- “tanh(), tanhf(), tanhl() — Calculate hyperbolic tangent” on page 1818

atand32(), atand64(), atand128(), atan2d32(), atan2d64(), atan2d128() - Calculate arctangent

Standards

Standards / Extensions	C or C++	Dependencies
C/C++ DFP	both	z/OS V1.10

Format

```
#define __STDC_WANT_DEC_FP__
#include <math.h>

_Decimal32  atand32(_Decimal32 x);
_Decimal64  atand64(_Decimal64 x);
_Decimal128 atand128(_Decimal128 x);
_Decimal32  atan(_Decimal32 x);     /* C++ only */
_Decimal64  atan(_Decimal64 x);     /* C++ only */
_Decimal128 atan(_Decimal128 x);    /* C++ only */

_Decimal32  atan2d32(_Decimal32 y, _Decimal32 x);
_Decimal64  atan2d64(_Decimal64 y, _Decimal64 x);
_Decimal128 atan2d128(_Decimal128 y, _Decimal128 x);
_Decimal32  atan2(_Decimal32 y, _Decimal32 x);   /* C++ only */
_Decimal64  atan2(_Decimal64 y, _Decimal64 x);   /* C++ only */
_Decimal128 atan2(_Decimal128 y, _Decimal128 x); /* C++ only */
```

General description

The atan() and atan2() functions calculate the arctangent of x and *y/x*, respectively.

These functions work in IEEE decimal floating-point format. See “IEEE decimal floating-point” on page 95 for more information.

Note: To use IEEE decimal floating-point, the hardware must have the Decimal Floating-Point Facility installed.

Returned value

Returns a value in the range -pi/2 to pi/2 radians.

If both arguments of atan2() are zero, the function sets errno to EDOM and returns 0. No other errors will occur.

Example

CELEBA15

```
/* CELEBA15

   This example illustrates the atand64() and atan2d64() functions.

*/

#define __STDC_WANT_DEC_FP__
#include <math.h>
#include <stdio.h>

int main(void)
{
    _Decimal64 a,b,c,d;

    c = 0.45DD;
    d = 0.23DD;

    a = atand64(c);
    b = atan2d64(c,d);

    printf("atand64( %Df ) = %Df\n", c, a);
    printf("atan2d64( %Df, %Df ) = %Df\n", c, d, b);
}
```

Related information

- "math.h" on page 44
- "acosd32(), acosd64(), acosd128() - Calculate arccosine" on page 137
- "acoshd32(), acoshd64(), acoshd128() - Calculate hyperbolic arccosine" on page 140
- "asind32(), asind64(), asind128() - Calculate arcsine" on page 187
- "asinhd32(), asinhd64(), asinhd128() - Calculate hyperbolic arcsine" on page 189
- "atanhd32(), atanhd64(), atanhd128() - Calculate hyperbolic arctangent" on page 197
- "cosd32(), cosd64(), cosd128() — Calculate cosine" on page 329
- "coshd32(), coshd64(), coshd128() - Calculate hyperbolic cosine" on page 332
- "sind32(), sind64(), sind128() — Calculate sine" on page 1673
- "sinhd32(), sinhd64(), sinhd128() - Calculate hyperbolic sine" on page 1676
- "tand32(), tand64(), tand128() - Calculate tangent" on page 1817
- "tanhd32(), tanhd64(), tanhd128() - Calculate hyperbolic tangent" on page 1820

atanh(), atanhf(), atanhl() — Calculate hyperbolic arctangent

Standards

Standards / Extensions	C or C++	Dependencies
XPG4.2 C99 Single UNIX Specification, Version 3 C++ TR1 C99	both	

Format

```
#define _XOPEN_SOURCE_EXTENDED 1
#include <math.h>

double atanh(double x);
```

C99:

```
#define _ISOC99_SOURCE
#include <math.h>

float atanhf(float x);
long double atanhl(long double x);
```

C++ TR1 C99

```
#define _TR1_C99
#include <math.h>
float atanh(float x);
long double atanh(long double x);
```

General description

The atanh() function returns the hyperbolic arctangent of its argument *x*.

Note: The following table shows the viable formats for these functions. See "IEEE binary floating-point" on page 94 for more information about IEEE Binary Floating-Point.

Function	Hex	IEEE
atanh	X	X
atanhf	X	X
atanhl	X	X

Returned value

If successful, atanh() returns the hyperbolic arctangent of its argument *x*.

atanh() fails, returns 0.0 and sets errno to one of the following values:

Error Code
  Description

EDOM
  The *x* argument has a value greater than 1.0.

ERANGE
  The *x* argument has a value equal to 1.0.

Special behavior for IEEE: If successful, the function returns the hyperbolic arctangent of its argument *x*.

If the absolute value of *x* is greater than 1.0, atanh() sets errno to EDOM and returns NaNQ. If the value of *x* is equal to 1.0, the function sets errno to ERANGE and returns +HUGE_VAL.

Related information

- "math.h" on page 44
- "acos(), acosf(), acosl() — Calculate arccosine" on page 135
- "acosh(), acoshf(), acoshl() — Calculate hyperbolic arccosine" on page 139
- "asin(), asinf(), asinl() — Calculate arcsine" on page 185
- "asinh(), asinhf(), asinhl() — Calculate hyperbolic arcsine" on page 188
- "atan(), atanf(), atanl(), atan2(), atan2f(), atan2l() — Calculate arctangent" on page 192
- "cos(), cosf(), cosl() — Calculate cosine" on page 328
- "cosh(), coshf(), coshl() — Calculate hyperbolic cosine" on page 331

- “sin(), sinf(), sinl() — Calculate sine” on page 1672
- “sinh(), sinhf(), sinhl() — Calculate hyperbolic sine” on page 1674
- “tan(), tanf(), tanl() — Calculate tangent” on page 1815
- “tanh(), tanhf(), tanhl() — Calculate hyperbolic tangent” on page 1818

atanhd32(), atanhd64(), atanhd128() - Calculate hyperbolic arctangent

Standards

Standards / Extensions	C or C++	Dependencies
C/C++ DFP	both	z/OS V1.10

Format

```
#define __STDC_WANT_DEC_FP__
#include <math.h>

_Decimal32  atanhd32(_Decimal32 x);
_Decimal64  atanhd64(_Decimal64 x);
_Decimal128 atanhd128(_Decimal128 x);
_Decimal32  atanh(_Decimal32 x);      /* C++ only */
_Decimal64  atanh(_Decimal64 x);      /* C++ only */
_Decimal128 atanh(_Decimal128 x);     /* C++ only */
```

General description

The atanh() function returns the hyperbolic arctangent of its argument *x*.

These functions work in IEEE decimal floating-point format. See “IEEE decimal floating-point” on page 95 for more information.

Note: To use IEEE decimal floating-point, the hardware must have the Decimal Floating-Point Facility installed.

Returned value

If successful, the function returns the hyperbolic arctangent of its argument x.

If the absolute value of *x* is greater than 1.0, atanh() sets errno to EDOM and returns NaNQ. If the value of *x* is equal to 1.0, the function sets errno to ERANGE and returns +HUGE_VAL_D32, +HUGE_VAL_D64 or +HUGE_VAL_D128.

Example

CELEBA16

```
/* CELEBA16

   This example illustrates the atanhd32() function.

*/

#define  __STDC_WANT_DEC_FP__
#include <math.h>
#include <stdio.h>

int main(void)
{
   _Decimal32 x, y;
   x = 0.5DF;
   y = atanhd32(x);

   printf("atanhd32(%Hf) = %Hf\n", x, y);
}
```

Related information

- "math.h" on page 44
- "acosd32(), acosd64(), acosd128() - Calculate arccosine" on page 137
- "acoshd32(), acoshd64(), acoshd128() - Calculate hyperbolic arccosine" on page 140
- "asind32(), asind64(), asind128() - Calculate arcsine" on page 187
- "asinhd32(), asinhd64(), asinhd128() - Calculate hyperbolic arcsine" on page 189
- "atand32(), atand64(), atand128(), atan2d32(), atan2d64(), atan2d128() - Calculate arctangent" on page 194
- "cosd32(), cosd64(), cosd128() — Calculate cosine" on page 329
- "coshd32(), coshd64(), coshd128() - Calculate hyperbolic cosine" on page 332
- "sind32(), sind64(), sind128() — Calculate sine" on page 1673
- "sinhd32(), sinhd64(), sinhd128() - Calculate hyperbolic sine" on page 1676
- "tand32(), tand64(), tand128() - Calculate tangent" on page 1817
- "tanhd32(), tanhd64(), tanhd128() - Calculate hyperbolic tangent" on page 1820

__atanpid32(), __atanpid64(), __atanpid128() - Calculate arctangent(x)/pi

Standards

Standards / Extensions	C or C++	Dependencies
Language Environment	both	z/OS V1.10

Format

```
#define __STDC_WANT_DEC_FP__
#include <math.h>

_Decimal32  __atanpid32(_Decimal32 x);
_Decimal64  __atanpid64(_Decimal64 x);
_Decimal128 __atanpid128(_Decimal128 x);
```

General description

Calculates the value of arctangent(*x*)/pi.

These functions work in IEEE decimal floating-point format. See "IEEE decimal floating-point" on page 95 for more information.

Note: To use IEEE decimal floating-point, the hardware must have the Decimal Floating-Point Facility installed.

Returned value

Returns the calculated value expressed in radians.

Example

CELEBA17

```
/* CELEBA17

   This example illustrates the __atanpid64() function.

*/

#define  __STDC_WANT_DEC_FP__
#include <math.h>
#include <stdio.h>

int main(void)
{
   _Decimal64 x, y;

   x = 5.0DD;
   y = __atanpid64(x);

   printf("__atanpid64(%Df) = %Df\n", x, y);
}
```

Related information

- "math.h" on page 44
- "__cospid32(), __cospid64(), __cospid128() — Calculate cosine of pi *x" on page 333
- "__sinpid32(), __sinpid64(), __sinpid128() — Calculate sine of pi * x" on page 1677

atexit() — Register program termination function

Standards

Standards / Extensions	C or C++	Dependencies
ISO C XPG4 XPG4.2 C99 Single UNIX Specification, Version 3	both	

Format

```
#include <stdlib.h>

int atexit(void (*func)(void));
```

General description

Records a function, pointed to by *func*, that the system calls at normal program termination. Termination is a result of exit() or returning from main(), regardless of the language of the main() routine. Process termination started by _exit() or by a terminating signal under Language Environment is not included.

The functions are executed in the reverse order that they were registered. The registered function must return to ensure that all registered functions are called. The functions registered with atexit() are started before streams and files are closed. You may specify a number of functions to the limit set by the ATEXIT_MAX constant, which is defined in <limits.h>.

Under z/OS UNIX services only, when a process ends, the address space is ended; otherwise, the address space persists.

Special behavior for z/OS XL C: The C Library atexit() function has the following restrictions:

- Any function registered by a fetched module that has been released is removed from the list at the time of release(). See fetch(), fetchep(), and release() for details about fetching and releasing modules.
- Any function registered in an explicitly loaded DLL (using dllload()) that has been freed (using dllfree()) is removed from the list. But, if the DLL in question has also been implicitly loaded, then the function is NOT removed from the atexit list.
- All C Library library functions can be used in a registered routine except exit().
- When a program is running under CICS control, if an `EXEC CICS RETURN` command or an `EXEC CICS XCTL` command is issued, the atexit() list that has been previously registered is not run.
- Use of the system() library function within atexit() may result in undefined behavior.
- Use of non-C subroutines or functions in the atexit() list will result in undefined behavior.
- The atexit() list will not be run when abort() is called.

Special behavior for C++:

- All of the behaviors listed under "Special Behavior for z/OS XL C".
- Because C and C++ linkage conventions are incompatible, atexit() cannot receive C++ function pointers. If you attempt to pass a C++ function pointer to atexit(), the compiler will flag it as an error. To use the C++ atexit() function, you must ensure that all functions registered for atexit() have C linkage by declaring them as `extern "C"`.
- You can use `try`, `throw`, and `catch` in a function registered for atexit(). However, by the time an atexit() function is driven, all stack frames will have collapsed. As a result, the only `catch` clauses available for `throw` will be the ones coded in the atexit() function. If those `catch` clauses cannot handle the thrown object, terminate() will be called.

Special behavior for XPG4.2: The maximum number of functions that can be registered is specified by the symbol ATEXIT_MAX which is defined in the limits.h header.

Returned value

If successful, atexit() returns 0.

If unsuccessful, atexit() returns nonzero.

Example

CELEBA10

```
/* CELEBA10

   This example uses the atexit() function to call the function goodbye()
   at program termination.

 */
#include <stdlib.h>
#include <stdio.h>

#ifdef __cplusplus             /* the __cplusplus macro is     */
```

```
extern "C" void goodbye(void); /* automatically defined by the  */
#else                          /* C++/MVS compiler              */
void goodbye(void);
#endif

int main(void)
{
   int rc;

   rc = atexit(goodbye);
   if (rc != 0)
      printf("Error in atexit");
   exit(0);
}

void goodbye(void)
   /* This function is called at normal program termination */
{
   printf("The function goodbye was called \
at program termination\n");
}
```

Output

```
The function goodbye was called at program termination
```

Related information

- “Condition Handling” in *IBM Language Environment Programming Guide*
- “stdlib.h” on page 70
- “abort() — Stop a program” on page 101
- “exit() — End program” on page 443
- “fetch() — Get a load module” on page 507
- “fetchep() — Share writable static” on page 519
- “release() — Delete a load module” on page 1424
- “signal() — Handle interrupts” on page 1643

__atoe() — ISO8859-1 to EBCDIC string conversion

Standards

Standards / Extensions	C or C++	Dependencies
z/OS UNIX	both	

Format

```
#include <unistd.h>

int __atoe(char *string);
```

General description

The __atoe() function converts an ISO8859-1 character string *string* to its EBCDIC equivalent. The conversion is performed using the codeset page associated with the current locale. The input character string up to, but not including, the NULL is changed from an ISO8859-1 representation to that of the current locale.

The argument *string* points to the ISO8859-1 character string to be converted to its EBCDIC equivalent.

Returned value

If successful, __atoe() converts the input ISO8859-1 character string to its equivalent EBCDIC value, and returns the length of the converted string.

If unsuccessful, __atoe() returns -1 and sets errno to one of the following values:

Error Code
Description

EINVAL
The current locale does not describe a single-byte character set.

ENOMEM
There is insufficient storage to complete the conversion process.

Note: This function may internally call iconv_open() and iconv(). The errnos returned by these functions are propagated without modification.

Related information

- "unistd.h" on page 82
- "iconv() — Code conversion" on page 824
- "iconv_open() — Allocate code conversion descriptor" on page 829

__atoe_l() — ISO8859-1 to EBCDIC conversion operation

Standards

Standards / Extensions	C or C++	Dependencies
z/OS UNIX	both	

Format

```
#include <unistd.h>

int __atoe_l(char *bufferptr, int leng);
```

General description

The __atoe_l() function converts *leng* ISO8859-1 bytes in the buffer pointed to by *bufferptr* to their EBCDIC equivalent. The conversion is performed using the codeset page associated with the current locale.

The argument *bufferptr* points to a buffer containing the ISO8859-1 bytes to be converted to their EBCDIC equivalent. The input buffer is treated as sequence of bytes, and all bytes in the input buffer are converted, including any imbedded NULLs.

Returned value

If successful, __atoe_l() converts the input IOS8859-1 bytes to their equivalent EBCDIC value, and returns the number of bytes that were converted.

If unsuccessful, __atoe_l() returns -1 and sets errno to one of the following values:

Error Code
Description

EINVAL
The current locale does not describe a single-byte character set.

ENOMEM
There is insufficient storage to complete the conversion process.

Note: This function may internally call iconv_open() and iconv(). The errnos returned by these functions are propagated without modification.

Related information

- “unistd.h” on page 82
- “iconv() — Code conversion” on page 824
- “iconv_open() — Allocate code conversion descriptor” on page 829

atof() — Convert character string to double

Standards

Standards / Extensions	C or C++	Dependencies
ISO C POSIX.1 XPG4 XPG4.2 C99 Single UNIX Specification, Version 3	both	

Format

```
#include <stdlib.h>

double atof(const char *nptr);
```

General description

The atof() function converts the initial portion of the string pointed to by *nptr* to a 'double'. This is equivalent to

```
strtod(nptr, (char**)NULL)
```

The double value is either hexadecimal floating point or binary floating point, depending on the floating point mode of the thread invoking the atof() function. This function uses _isBF() to determine the floating point mode of the invoking thread.

See the “*fscanf* Family of Formatted Input Functions” on fscanf(), scanf(), sscanf() — Read and format data for a description of special infinity and NaN sequences recognized by z/OS formatted input functions, including atof() and strtod() in IEEE Binary Floating-Point mode.

Returned value

The atof() function returns the converted value if the value can be represented, otherwise the return value is undefined.

Related information

- “stdlib.h” on page 70
- “atoi() — Convert character string to integer” on page 204
- “atol() — Convert character string to long” on page 204

- “fscanf(), scanf(), sscanf() — Read and format data” on page 625
- “__isBFP() — Determine application floating-point format” on page 904
- “strtod() — Convert character string to double” on page 1765
- “strtol() — Convert character string to long” on page 1775
- “strtoul() — Convert string to unsigned integer” on page 1780

atoi() — Convert character string to integer

Standards

Standards / Extensions	C or C++	Dependencies
ISO C POSIX.1 XPG4 XPG4.2 C99 Single UNIX Specification, Version 3	both	

Format

```
#include <stdlib.h>

int atoi(const char *nptr);
```

General description

The atoi() function converts the initial portion of the string pointed to by *nptr* to a 'int'. This is equivalent to

```
(int)strtol(nptr, NULL, 10)
```

Returned value

There are no documented errno values.

Related information

- “stdlib.h” on page 70
- “atof() — Convert character string to double” on page 203
- “atol() — Convert character string to long”
- “fscanf(), scanf(), sscanf() — Read and format data” on page 625
- “strtod() — Convert character string to double” on page 1765
- “strtol() — Convert character string to long” on page 1775
- “strtoul() — Convert string to unsigned integer” on page 1780

atol() — Convert character string to long

Standards

Standards / Extensions	C or C++	Dependencies
ISO C POSIX.1 XPG4 XPG4.2 C99 Single UNIX Specification, Version 3	both	

Format

```
#include <stdlib.h>

long int atol(const char *nptr);
```

General description

The atol() function converts the initial portion of the string pointed to by *nptr* to a 'long int'. This is equivalent to

```
strtol(nptr, (char**)NULL, 10)
```

Returned value

The atol() function returns the converted value if the value can be represented, otherwise the return value is undefined.

Related information

- "stdlib.h" on page 70
- "atof() — Convert character string to double" on page 203
- "atoi() — Convert character string to integer" on page 204
- "fscanf(), scanf(), sscanf() — Read and format data" on page 625
- "strtod() — Convert character string to double" on page 1765
- "strtol() — Convert character string to long" on page 1775
- "strtoul() — Convert string to unsigned integer" on page 1780

atoll() — Convert character string to signed long long

Standards

Standards / Extensions	C or C++	Dependencies
C99 Single UNIX Specification, Version 3 C++ TR1 C99	both	z/OS V1R7

Format

```
#define _ISOC99_SOURCE
#include <stdlib.h>
long long atoll(const char *nptr);
```

Compile requirement: Use of this function requires the long long data type. See *z/OS XL C/C++ Language Reference* for information on how to make long long available.

General description

atoll() converts the initial portion of the string pointed to by *nptr* to a '**long long** int'. This is equivalent to strtoll(*nptr*, (char **)NULL, 10).

Returned value

If successful, atoll() returns the converted signed **long long** value, represented in the string. If unsuccessful, it returns an undefined value.

Related information

- “stdlib.h” on page 70
- “atof() — Convert character string to double” on page 203
- “atoi() — Convert character string to integer” on page 204
- “atol() — Convert character string to long” on page 204
- “fscanf(), scanf(), sscanf() — Read and format data” on page 625
- “strtod() — Convert character string to double” on page 1765
- “strtol() — Convert character string to long” on page 1775
- “strtoul() — Convert string to unsigned integer” on page 1780

__a2e_l() — Convert characters from ASCII to EBCDIC

Standards

Standards / Extensions	C or C++	Dependencies
	both	z/OS V1R2

Format

```
#include <unistd.h>

size_t  __a2e_l(char  *bufptr, size_t  szLen)
```

General description

The __a2e_l() function converts *szLen* characters in *bufptr* from ASCII to EBCDIC, returning the number of characters converted if successful or -1 if not. Conversion occurs in place in the buffer. __a2e_l() is not sensitive to the locale, and only converts between ISO8859-1 and IBM-1047.

Note: This function is valid for applications compiled XPLINK only.

Returned value

If successful, __a2e_l() returns the number of characters converted.

If unsuccessful, __a2e_l() returns -1 and sets errno to the following value:

Error Code
Description

EINVAL
The pointer to *bufptr* is NULL or *szLen* is a negative value.

Related information

- “unistd.h” on page 82
- “__a2e_s() — Convert string from ASCII to EBCDIC”
- “__e2a_l() — Convert characters from EBCDIC to ASCII” on page 458
- “__e2a_s() — Convert string from EBCDIC to ASCII” on page 459

__a2e_s() — Convert string from ASCII to EBCDIC

Standards

Standards / Extensions	C or C++	Dependencies
	both	z/OS V1R2

Format

```
#include <unistd.h>

size_t  __a2e_s(char  *string)
```

General description

The __a2e_s() function converts a string from ASCII to EBCIDIC, returning the string length if successful or -1 if not. Conversion occurs in place in the string. __a2e_s() is not sensitive to the locale, and only converts between ISO8859-1 and IBM-1047.

Note: This function is valid for applications compiled XPLINK only.

Returned value

If successful, __a2e_s() returns the string length.

If unsuccessful, __a2e_s() returns -1 and sets errno to the following value:

Error Code
 Description

EINVAL
 The pointer to string is NULL.

Related information

- "unistd.h" on page 82
- "__a2e_l() — Convert characters from ASCII to EBCDIC" on page 206
- "__e2a_l() — Convert characters from EBCDIC to ASCII" on page 458
- "__e2a_s() — Convert string from EBCDIC to ASCII" on page 459

a64l() — Convert base 64 string representation to long integer

Standards

Standards / Extensions	C or C++	Dependencies
XPG4.2 Single UNIX Specification, Version 3	both	

Format

```
#define  _XOPEN_SOURCE_EXTENDED 1
#include <stdlib.h>

long a64l(const char *string);
```

General description

The a64l() function converts a string representation of a base 64 number into its corresponding long value. It scans the string from left to right with the least significant character on the left, decoding each character as a 6-bit base 64 number. If the string pointed to by *string* contains more than six characters, a64l() uses only the first six. If the first six characters of the string contain a NULL character, a64l() uses only the characters preceding the first NULL. The following characters are used to represent digits:

Character	Digit Represented
.	0
/	1
0-9	2-11
A-Z	12-37
a-z	38-63

Returned value

If successful, a64l() returns the long value resulting from conversion of the input string.

If the string pointed to by *string* is NULL, a64l() returns 0.

There are no errno values defined.

Related information

- “stdlib.h” on page 70
- “l64a() — Convert long to base 64 string representation” on page 1026
- “strtoul() — Convert string to unsigned integer” on page 1780

basename() — Return the last component of a path name

Standards

Standards / Extensions	C or C++	Dependencies
XPG4.2 Single UNIX Specification, Version 3	both	

Format

```
#define _XOPEN_SOURCE_EXTENDED 1
#include <libgen.h>

char *basename(char *path);
```

General description

The basename() function takes the pathname pointed to by *path* and returns a pointer to the final component of the pathname, deleting any trailing '/' characters.

If the string consists entirely of the '/' character, basename() returns a pointer to the string “/”.

If *path* is a NULL pointer or points to an empty string, basename() returns a pointer to the string “.”. The basename() function may modify the string pointed to by path.

Examples:

```
Input String     Output String
"/usr/lib"       "lib"
"/usr/"          "usr"
"/"              "/"
```

Returned value

If successful, basename() returns a pointer to the final component of *path*.

There are no errno values defined.

Related information

- “libgen.h” on page 38
- “dirname() — Report the parent directory of a path name” on page 381

bcmp() — Compare bytes in memory

Standards

Standards / Extensions	C or C++	Dependencies
XPG4.2 Single UNIX Specification, Version 3	both	

Format

```
#define  _XOPEN_SOURCE_EXTENDED 1
#include <strings.h>

int bcmp(const void *s1, const void *s2, size_t n);
```

General description

The bcmp() function compares the first *n* bytes of the area pointed to by *s1* with the area pointed to by *s2*.

Note: The bcmp() function has been moved to the Legacy Option group in Single UNIX Specification, Version 3 and may be withdrawn in a future version. The memcmp() function is preferred for portability.

Returned value

If *s1* and *s2* are identical, bcmp() returns 0. Otherwise, bcmp() returns nonzero. Both areas are assumed to be at least *n* bytes long.

If the value of *n* is zero, bcmp() returns 0.

There are no errno values defined.

Related information

- “strings.h” on page 72
- “bcopy() — Copy bytes in memory”
- “bzero() — Zero bytes in memory” on page 223
- “memcmp() — Compare bytes” on page 1062

bcopy() — Copy bytes in memory

Standards

Standards / Extensions	C or C++	Dependencies
XPG4.2 Single UNIX Specification, Version 3	both	

Format

```
#define  _XOPEN_SOURCE_EXTENDED 1
#include <strings.h>

void bcopy(const void *s1, void *s2, size_t n);
```

General description

The bcopy() function copies *n* bytes from the area pointed to by *s1* to the area pointed to by *s2* using the memcpy() function.

Note: The bcopy() function has been moved to the Legacy Option group in Single UNIX Specification, Version 3 and may be withdrawn in a future version. The memmove() function is preferred for portability.

Returned value

bcopy() returns no values.

There are no errno values defined.

Related information

- "strings.h" on page 72
- "bcmp() — Compare bytes in memory" on page 209
- "bzero() — Zero bytes in memory" on page 223
- "memccpy() — Copy bytes in memory" on page 1060
- "memcpy() — Copy buffer" on page 1063
- "memmove() — Move buffer" on page 1065

bind() — Bind a name to a socket

Standards

Standards / Extensions	C or C++	Dependencies
XPG4.2 Single UNIX Specification, Version 3	both	

Format

X/Open

```
#define _XOPEN_SOURCE_EXTENDED 1
#include <sys/socket.h>

int bind(int socket, const struct sockaddr *address, socklen_t address_len);
```

Berkeley Sockets

```
#define _OE_SOCKETS
#include <sys/types.h>
#include <sys/socket.h>

int bind(int socket, struct sockaddr *address, int address_len);
```

General description

The bind() function binds a unique local name to the socket with descriptor *socket*. After calling socket(), a descriptor does not have a name associated with it. However, it does belong to a particular address family as specified when socket() is called. The exact format of a name depends on the address family.

Parameter
Description

socket The socket descriptor returned by a previous socket() call.

address The pointer to a **sockaddr** structure containing the name that is to be bound to *socket*.

address_len
The size of *address* in bytes.

The *socket* parameter is a socket descriptor of any type created by calling socket().

The *address* parameter is a pointer to a buffer containing the name to be bound to *socket*. The *address_len* parameter is the size, in bytes, of the buffer pointed to by *address*. For AF_UNIX, this function creates a file that you later need to unlink besides closing the socket.

Socket Descriptor Created in the AF_INET Domain

If the socket descriptor *socket* was created in the AF_INET domain, the format of the name buffer is expected to be **sockaddr_in**, as defined in the include file **netinet/in.h**:

```
struct in_addr
{
        ip_addr_t s_addr;
};

 struct sockaddr_in {
     unsigned char  sin_len;
     unsigned char  sin_family;
     unsigned short sin_port;
     struct in_addr sin_addr;
     unsigned char  sin_zero[8];

};
```

The *sin_family* field must be set to AF_INET.

The *sin_port* field is set to the port to which the application must bind. It must be specified in network byte order. If *sin_port* is set to 0, the caller leaves it to the system to assign an available port. The application can call getsockname() to discover the port number assigned.

The *sin_addr.s_addr* field is set to the Internet address and must be specified in network byte order. On hosts with more than one network interface (called multihomed hosts), a caller can select the interface to which it is to bind. Subsequently, only UDP packets and TCP connection requests from this interface (which match the bound name) are routed to the application. If this field is set to the constant INADDR_ANY, as defined in **netinet/in.h**, the caller is requesting that the socket be bound to all network interfaces on the host. Subsequently, UDP packets and TCP connections from all interfaces (which match the bound name) are routed to the application. This becomes important when a server offers a service to multiple networks. By leaving the address unspecified, the server can accept all UDP packets and TCP connection requests made for its port, regardless of the network interface on which the requests arrived.

The *sin_zero* field is not used and must be set to all zeros.

Socket Descriptor Created in the AF_INET6 Domain If the socket descriptor *socket* was created in the AF_INET6 domain, the format of the name buffer is expected to be **sockaddr_in6**, as defined in the include file **netinet/in.h**. The structure is defined as follows:

```
struct sockaddr_in6 {
   uint8_t         sin6_len;
   sa_family_t     sin6_family;
   in_port_t       sin6_port;
   uint32_t        sin6_flowinfo;
   struct in6_addr  sin6_addr;
   uint32_t        sin6_scope_id;
};
```

The sin6_len field is set to the size of this structure. The SIN6_LEN macro is defined to indicate the version of the sockaddr_in6 structure being used.

The sin6_family field identifies this as a sockaddr_in6 structure. This field overlays the sa_family field when the buffer is cast to a sockaddr structure. The value of this field must be AF_INET6.

The sin6_port field contains the 16-bit UDP or TCP port number. This field is used in the same way as the sin_port field of the sockaddr_in structure. The port number is stored in network byte order.

The sin6_flowinfo field is a 32-bit field that contains the traffic class and the flow label.

The sin6_addr field is a single in6_addr structure. This field holds one 128-bit IPv6 address. The address is stored in network byte order.

The sin6_scope_id field is a 32-bit integer that identifies a set of interfaces as appropriate for the scope of the address carried in the sin6_addr field. For a link scope sin6_addr, sin6_scope_id, this would be an interface index. For a site scope sin6_addr, sin6_scope_id, this would be a site identifier.

Socket Descriptor Created in the AF_UNIX Domain

If the socket descriptor *socket* is created in the AF_UNIX domain, the format of the name buffer is expected to be **sockaddr_un**, as defined in the include file **un.h**.

```
 struct sockaddr_un {
    unsigned char  sun_len;
    unsigned char  sun_family;
             char  sun_path[108];       /* pathname */
};
```

The *sun_family* field is set to AF_UNIX.

The *sun_path* field contains the NULL-terminated pathname, and *sun_len* contains the length of the pathname.

Notes:

1. For AF_UNIX, when a bind is issued, a file is created with a mode of 660. In order to allow other users to access this file, a chmod() should be issued to modify this mode if desired.
2. For AF_UNIX, when closing sockets that were bound, you should also use unlink() to delete the file created at bind() time.
3. The pathname the client uses on the bind() must be unique.
4. The sendto() call must specify the pathname associated with the server.
5. For AF_INET or AF_INET6, the user must have appropriate privileges to bind to a port in the range from 1 to 1023.

Special Behavior for C++

To use this function with C++, you must use the _XOPEN_SOURCE_EXTENDED 1 feature test macro.

Note: The bind() function has a dependency on the level of the Enhanced ASCII Extensions. See “Enhanced ASCII support” on page 2113 for details.

Returned value

If successful, bind() returns 0.

If unsuccessful, bind() returns -1 and sets errno to one of the following values:

Error Code
Description

EACCES
Permission denied.

EADDRINUSE
The address is already in use. See the SO_REUSEADDR option described under “getsockopt() — Get the options associated with a socket” on page 777 and the SO_REUSEADDR described under the “setsockopt() — Set options associated with a socket” on page 1578 for more information.

EADDRNOTAVAIL
The address specified is not valid on this host. For example, the Internet address does not specify a valid network interface.

EAFNOSUPPORT
The address family is not supported (it is not AF_UNIX, AF_INET, or AF_INET6).

EBADF
The *socket* parameter is not a valid socket descriptor.

EINVAL
One of three conditions may apply:

- The socket is already bound to an address—for example, trying to bind a name to a socket that is already connected.
- The socket was shut down.
- An incorrect parameter was passed on the bind() invocation.

Check the parameter values passed and ensure they are specified as described above.

EIO There has been a network or transport failure.

ENOBUFS
bind() is unable to obtain a buffer due to insufficient storage.

ENOTSOCK
The descriptor is for a file, not for a socket.

EOPNOTSUPP
The socket type of the specified socket does not support binding to an address.

EPERM
The user is not authorized to bind to the port specified.

The following are for AF_UNIX only:

Error Code
Description

EACCES
A component of the path prefix denies search permission, or the requested name requires writing in a directory with a mode that denies write permission.

EDESTADDRREQ
The *address* argument is a NULL pointer.

EIO An I/O error occurred.

ELOOP
Too many symbolic links were encountered in translating the pathname in *address*.

ENAMETOOLONG
A component of a pathname exceeded **NAME_MAX** characters, or an entire pathname exceeded **PATH_MAX** characters.

ENOENT
A component of the pathname does not name an existing file or the pathname is an empty string.

ENOTDIR
A component of the path prefix of the pathname in *address* is not a directory.

EROFS
The name would reside on a read-only file system.

Example

The following are examples of the bind() call. It is a good idea to zero the structure before using it to ensure that the name requested does not set any reserved fields.

AF_INET Domain Example

The following example illustrates the bind() call binding to interfaces in the AF_INET domain. The Internet address and port must be in network byte order. To put the port into network byte order, the htons() utility routine is called to convert a short integer from host byte order to network byte order. The *address* field is set using another utility routine, inet_addr(), which takes a character string representing the dotted-decimal address of an interface and returns the binary Internet address representation in network byte order.

```
int rc;
int s;
struct sockaddr_in myname;
/* Bind to a specific interface in the Internet domain */
/* make sure the sin_zero field is cleared */
memset(&myname, 0, sizeof(myname));
myname.sin_family = AF_INET;
myname.sin_addr.s_addr = inet_addr("129.5.24.1");
/* specific interface */
myname.sin_port = htons(1024);
⋮
rc = bind(s, (struct sockaddr *) &myname,
sizeof(myname));
/* Bind to all network interfaces in the Internet domain */
/* make sure the sin_zero field is cleared */
memset(&myname, 0, sizeof(myname));
myname.sin_family = AF_INET;
myname.sin_addr.s_addr = INADDR_ANY; /* specific interface */
myname.sin_port = htons(1024);
⋮
rc = bind(s, (struct sockaddr *) &myname,
sizeof(myname));
/* Bind to a specific interface in the Internet domain.
   Let the system choose a port                         */
/* make sure the sin_zero field is cleared */
memset(&myname, 0, sizeof(myname));
myname.sin_family = AF_INET;
myname.sin_addr.s_addr = inet_addr("129.5.24.1");
/* specific interface */
myname.sin_port = 0;
⋮
rc = bind(s, (struct sockaddr *) &myname,
sizeof(myname));
```

AF_UNIX Domain Example

The following example illustrates the bind() call binding to interfaces in the AF_UNIX domain.

```
/* Bind to a name in the UNIX domain */
struct sockaddr_un myname;
char socket_name[]="/tmp/socket.for._";
⋮
memset(&myname, 0, sizeof(myname));
myname.sun_family = AF_UNIX;
strcpy(myname.sun_path,socket_name);
myname.sun_len = sizeof(myname.sun_path);
⋮
rc = bind(s, (struct sockaddr *) &myname, SUN_LEN(&myname));
```

Related information

- “sys/socket.h” on page 74
- “connect() — Connect a socket” on page 308
- “getnetbyname() — Get a network entry by name” on page 742
- “getsockname() — Get the name of a socket” on page 775
- “htons() — Translate an unsigned short integer into network byte order” on page 820
- “inet_addr() — Translate an Internet address into network byte order” on page 856
- “listen() — Prepare the server for incoming client requests” on page 972
- “socket() — Create a socket” on page 1688

bind2addrsel() - Bind with source address selection

Standards

Standards / Extensions	C or C++	Dependencies
RFC 5014	both	

Format

```
#define _OPEN_SYS_SOCK_IPV6
#include <netinet/in.h>

int bind2addrsel(int sd, const struct sockaddr *dstaddr, socklen_t dstaddrlen);
```

General description

The bind2addrsel() function binds a socket to a source address and port that is appropriate to communicate with a given destination address. The source address is selected by the TCP stack according to either the default preference flags or the flags previously expressed by the application by use of the setsockopt() IPV6_ADDR_PREFERENCES call on this socket. If the bind is successful, the application can call getsockname() to determine the address and port selected. The application can then call inet6_is_srcaddr() to determine if the bound local address meets its preferences. For more information on source address selection, see *z/OS Communications Server: IPv6 Network and Appl Design Guide*.

Argument
Description

sd The socket to bind to a stack-selected source address and port.

dstaddr
A non-NULL pointer to a sockaddr_in6 structure initialized as follows:

- Clear the entire structure for sizeof(struct sockaddr_in6).
- *sin6_family* must be set to AF_INET6.
- Set *sin6_len* to the correct length for AF_INET6.
- Set *sin6_addr* to a 128-bit IPv6 destination address with which the local node wants to communicate.
- The *sin6_scope_id* must be set if the address is link-local.

dstaddrlen
The size of the sockaddr structure passed as argument.

Returned value

If successful, bind2addrsel() returns 0.

If unsuccessful, bind2addrsel() returns -1 and sets errno to one of the following values:

Error Code
Description

EADDRNOTAVAIL
The destination address is a multicast address but the socket type is SOCK_STREAM or there are no ephemeral ports available to satisfy the bind request.

EAFNOSUPPORT
The address family specified in the address structure is not supported.

EBADF
The socket descriptor is incorrect.

EHOSTUNREACH
The destination address is not reachable because there is no route.

EINVAL
One of the input parameters was not valid.

EIO There has been a network or transport failure.

ENOBUFS
A buffer could not be obtained.

ENOTSOCK
The *sd* parameter does not refer to a valid socket descriptor.

EOPNOTSUPP
The socket domain type is not supported.

EPROTOTYPE
The socket protocol is not TCP or UDP.

Related information

- “netinet/in.h” on page 53
- “bind() — Bind a name to a socket” on page 210
- “inet6_is_srcaddr() - Socket address verification” on page 842
- “getaddrinfo() — Get address information” on page 684
- “getsockopt() — Get the options associated with a socket” on page 777
- “setsockopt() — Set options associated with a socket” on page 1578

brk() — Change space allocation

Standards

Standards / Extensions	C or C++	Dependencies
XPG4.2	both	

Format

```
#define _XOPEN_SOURCE_EXTENDED 1
#include <unistd.h>

int brk(void *addr);
```

General description

Restriction: This function is not supported in AMODE 64.

The brk() function is used to change the space allocated for the calling process. The change is made by setting the process's break value to addr and allocating the appropriate amount of space. The amount of allocated space increases as the break value increases. The newly-allocated space is set to 0. However, if the application first decrements and then increments the break value, the contents of the reallocated space are not zeroed.

The storage space from which the brk() and sbrk() functions allocate storage is separate from the storage space that is used by the other memory allocation functions (malloc(), calloc(), etc.). Because this storage space must be a contiguous segment of storage, it is allocated from the initial heap segment only and thus is limited to the initial heap size specified for the calling program or the largest contiguous segment of storage available in the initial heap at the time of the first brk() or sbrk() call. Since this is a separate segment of storage, the brk() and sbrk() functions can be used by an application that is using the other memory allocation functions. However, it is possible that the user's region may not be large enough to support extensive usage of both types of memory allocation.

Prior usage of the brk() function has been limited to specialized cases where no other memory allocation function performed the same function. Because the brk() function may be unable to sufficiently increase the space allocation of the process when the calling application is using other memory functions, the use of other memory allocation functions, such as mmap(), is now preferred because it can be used portably with all other memory allocation functions and with any function that uses other allocation functions. Applications that require the use of brk() and/or sbrk() should refrain from using the other memory allocation functions and should be run with an initial heap size that will satisfy the maximum storage requirements of the program. The brk() function is not supported from a multithreaded environment, it will return in error if it is invoked in this environment.

Note:

This function is kept for historical reasons. It was part of the Legacy Feature in Single UNIX Specification, Version 2, but has been withdrawn and is not supported as part of Single UNIX Specification, Version 3. New applications should use malloc() instead of brk() or sbrk().

If it is necessary to continue using this function in an application written for Single UNIX Specification, Version 3, define the feature test macro _UNIX03_WITHDRAWN before including any standard system headers. The macro exposes all interfaces and symbols removed in Single UNIX Specification, Version 3.

Returned value

If successful, brk() returns 0.

If unsuccessful, brk() returns -1 and sets errno to one of the following values:

Error Code
    Description

ENOMEM
    The requested change would allocate more space than allowed for the calling process.

Related information

- “unistd.h” on page 82
- “sbrk() — Change space allocation” on page 1461

bsd_signal() — BSD version of signal()

Standards

Standards / Extensions	C or C++	Dependencies
XPG4.2 Single UNIX Specification, Version 3	both	POSIX(ON)

Format

```
#define _XOPEN_SOURCE_EXTENDED 1
#include <signal.h>

void (*bsd_signal(int sig, void (*func)(int)))(int);
```

General description

The bsd_signal() function provides a partially compatible interface for programs written to use the BSD form of the signal() function.

BSD signal() differs from ANSI signal() in that the **SA_RESTART** flag is set and the **SA_RESETHAND** is cleared when bsd_signal() is used. Whereas for signal() both of these flags are cleared and **_SA_OLD_STYLE** is set.

There are three functions available for establishing a signal's action, signal(), bsd_signal(), and sigaction(). The sigaction() function is the strategic way to establish a signal's action. The bsd_signal() and signal() functions are provided for compatibility with BSD and ANSI, respectively.

The argument *sig* is the signal type. See Table 48 on page 1612 for a list of the supported signal types or refer to the <signal.h> header. The argument *func* is the signal action. It may be set to **SIG_DFL**, **SIG_IGN**, or the address of a signal catching function that takes one input argument.

Special Behavior for C++

Because C and C++ linkage conventions are incompatible, bsd_signal() cannot receive a C++ function pointer as the start routine function pointer If you attempt to pass a C++ function pointer to bsd_signal(), the compiler will flag it as an error. You can pass a C or C++ function to bsd_signal() by declaring it as extern "C".

Usage notes

1. The use of the SIGTHSTOP and SIGTHCONT signal is not supported with this function.
2. The bsd_signal() function has been marked obsolescent in Single UNIX Specification, Version 3 and may be withdrawn in a future version. The sigaction() function is preferred for portability.

Returned value

If successful, bsd_signal() returns the previous action established for this signal type.

If unsuccessful, bsd_signal() returns **SIG_ERR** and sets errno to one of the following values:

Error Code
Description

EINVAL
The value of the argument *sig* was not a valid signal type, or an attempt was made to catch a signal that cannot be caught, or ignore a signal that cannot be ignored.

Related information

- "signal.h" on page 63
- "sigaction() — Examine or change a signal action" on page 1611
- "signal() — Handle interrupts" on page 1643
- "sigprocmask() — Examine or change a thread" on page 1651

bsearch() — Search arrays

Standards

Standards / Extensions	C or C++	Dependencies
ISO C POSIX.1 XPG4 XPG4.2 C99 Single UNIX Specification, Version 3	both	

Format

```
#include <stdlib.h>

void *bsearch(const void *key, const void *base, size_t num, size_t size,
              int (*compare)(const void *element1, const void *element2));
```

General description

Performs a binary search of an array of *num* elements, each of *size* bytes.

The pointer *base* points to the initial element of the array to be searched. *key* points to the object containing the value being sought. The array must be sorted in ascending key sequence, according to the comparison function. Otherwise, undefined behavior occurs.

The *compare* parameter is a pointer to a function you must supply. It compares two array elements and returns a value specifying their relationship. The bsearch() function calls this function one or more times during the search, passing the key and the pointer to one array element on each call. The function compares the elements and then returns one of the following values:

Value	**Meaning**
< 0	Object pointed to by `key` is less than the array element.
= 0	Object pointed to by `key` is equal to the array element.
> 0	Object pointed to by `key` is greater than the array element.

Special Behavior for C++

Because C++ and C linkage conventions are incompatible, bsearch() cannot receive C++ function pointers. If you attempt to pass a C++ function pointer to bsearch(), the compiler will flag it as an error. To use the C++ bsearch() function, you must ensure that the *compare* function has C linkage by declaring it as `extern "C"`.

Returned value

If successful, bsearch() returns a pointer to a matching element of the array. If two or more elements are equal, the element pointed to is not specified.

If unsuccessful finding the *key*, bsearch() returns NULL.

Example

CELEBB01

```
/* CELEBB01

   This example performs a binary search on the argv array of pointers  o
   to the program arguments and finds the position of the argument PATH.
   It first removes the program name from argv, and then sorts the array
   alphabetically before calling bsearch().

   The functions compare1 and compare2 compare the values pointed to by
   arg1 and arg2, and they return the result to bsearch().

 */
#include <stdlib.h>
#include <stdio.h>
#include <string.h>

#ifdef __cplusplus  /* the __cplusplus macro is   */
extern "C" {        /* automatically defined by the   */
#endif              /* C++/MVS compiler               */
   int compare1(const void *arg1, const void *arg2);
   int compare2(const void *arg1, const void *arg2);
#ifdef __cplusplus
}
#endif


int main(int argc, char *argv[])
{
   char **result;
   char *key = "PATH";
   int i;
   argv++;
   argc--;

              /* sort to ensure that the input is ordered */
   qsort((char *)argv, argc, sizeof(char *), compare1);

   result = (char**)bsearch(key, (void *)argv, argc, sizeof(char *),
                            compare2);
   if (result != NULL) {
      printf("The key <%s> was found.\n",*result);
   }
   else printf("Match not found\n");
}

int compare1(const void *arg1, const void *arg2)
{
   return (strcmp(*(char **)arg1, *(char **)arg2));
}

int compare2(const void *arg1, const void *arg2) {
   return (strcmp((char *)arg1, *(char **)arg2));
}
```

Input

```
progname Is there PATH in this sentence?
```

Output

```
The key <PATH> was found.
```

Related information

- "stdlib.h" on page 70
- "qsort() — Sort array" on page 1363

btowc() — Convert single-byte character to wide-character

Standards

Standards / Extensions	C or C++	Dependencies
ISO C C99 Single UNIX Specification, Version 3	both	z/OS V1R2

Format

```
#include <wchar.h>

wint_t btowc(int c);
```

General description

The btowc() function determines whether *c* constitutes a valid (one-byte) character in the initial shift state.

The behavior of this wide-character function is affected by the LC_CTYPE category of the current locale. If you change the category, undefined results can occur.

Returned value

If successful, btowc() returns the wide-character representation of the character *c*.

If *c* has the value EOF or if (unsigned char)*c* does not constitute a valid (one-byte) character in the initial shift state, btowc() returns WEOF.

There are no documented errno values.

Related information

- "Internationalization: Locales and Character Sets" in *z/OS XL C/C++ Programming Guide*
- "locale.h" on page 40
- "wchar.h" on page 85
- "mbtowc() — Convert multibyte character to wide character" on page 1056
- "setlocale() — Set locale" on page 1552
- "wctob() — Convert wide character to byte" on page 2052

bzero() — Zero bytes in memory

Standards

Standards / Extensions	C or C++	Dependencies
XPG4.2 Single UNIX Specification, Version 3	both	

Format

```
#define  _XOPEN_SOURCE_EXTENDED 1
#include <strings.h>

void bzero(void *s, size_t n);
```

General description

The bzero() function places *n* zero-valued bytes in the area pointed to by *s*.

Note: The bzero() function has been moved to the Legacy Option group in Single UNIX Specification, Version 3 and may be withdrawn in a future version. The memset() function is preferred for portability.

Returned value

bzero() returns no values.

There are no errno values defined for bzero().

Related information

- “strings.h” on page 72
- “bcmp() — Compare bytes in memory” on page 209
- “bcopy() — Copy bytes in memory” on page 209
- “memset() — Set buffer to value” on page 1066

c16rtomb() — Convert a char16_t character to a multibyte character

Standards

Standards / Extensions	C or C++	Dependencies
ISO C Amendment C11	both	z/OS V2R1

Format

```
#include <uchar.h>

size_t c16rtomb(char * restrict s, char16_t c16, mbstate_t * restrict ps);
```

General description

The c16rtomb() function converts a wide character of type char16_t to a multibyte character, and returns the number of bytes stored in *s* (including any shift sequences).

If *s* is not a null pointer, the c16rtomb() function determines the number of bytes needed to represent the multibyte character that corresponds to the wide character given by *c16* (including any shift sequences), and stores the multibyte character representation in the array whose first element is pointed to by *s*. At most MB_CUR_MAX bytes are stored. If the *c16* is a null wide character, a null byte is stored, preceded by any shift sequence needed to restore the initial shift state; the resulting state described is the initial conversion state.

If *s* is a null pointer, the c16rtomb() function is equivalent to the call `c16rtomb(buf, L'\0', ps)`, where *buf* is an internal buffer.

If *ps* is a null pointer, c16rtomb() uses its own internal object to track the shift state. Otherwise **ps* must be a valid mbstate_t object. An mbstate_t object **ps* can be initialized to the initial state by assigning 0 to it, or by calling `c16rtomb(NULL, L'\0', ps)`.

Usage notes

1. To use the c16rtomb() function, compile the source code with the LANGLVL(EXTC1X) option.
2. The result *s* for stateful multibyte encodings, such as EBCDIC MBCS, might leave out shift bytes according to the conversion state. The first DBCS character in the output sequence has only shift-out character, and the following characters have neither shift-out nor shift-in. The ending shift-in will not be produced until an SBCS character or a null wide character is encountered.
3. The c16rtomb() function only supports the CCSIDs that are provided by Unicode Services.
4. The Unicode combining characters are not supported, and will be converted to substitute character of target CCSID.
5. The result of converting multiple string alternately in one thread by using multiple mbstate_t objects (including the internal one) is undefined.

Returned value

The c16rtomb() function returns the number of bytes stored in the array object (including any shift sequences). When *c16* is not a valid wide character, an encoding error occurs. The function stores the vlaue of the macro EILSEQ in errno and returns (size_t)(-1); the conversion state is unspecified.

Example

```
#include <stdio.h>
#include <stdlib.h>
#include <uchar.h>

int main(void)
{
   char16_t in = u'a';
   mbstate_t st = 0;
   char out[MB_CUR_MAX];
   int rc, i;
   rc = c16rtomb(out, in, &st);
   if (rc < 0) {
      perror("c16rtomb() fails to convert");
      exit(-1);
   }
   printf(" c16: 0x%04x \n", in);
   printf(" return code: %d \n", rc);
   printf(" mb character: ");
   for (i=0; i<rc; i++)
```

```
      printf(" 0x%02x", out[i]);
   printf("\n");
   return 0;
}
```

Output:

```
c16: 0x0061
return code: 1
mb character: 0x81
```

Related information

- “Internationalization: Locales and Character Sets” in *z/OS XL C/C++ Programming Guide*
- “Appendix A. Description of CCSIDs” in *z/OS Unicode Services User's Guide and Reference*
- “uchar.h” on page 82
- “setlocale() — Set locale” on page 1552
- “c32rtomb() — Convert a char32_t character to a multibyte character”
- “mbrtoc16() — Convert a multibyte character to a char16_t character” on page 1045

c32rtomb() — Convert a char32_t character to a multibyte character

Standards

Standards / Extensions	C or C++	Dependencies
ISO C Amendment C11	both	z/OS V2R1

Format

```
#include <uchar.h>

size_t c32rtomb(char * restrict s, char32_t c32, mbstate_t * restrict ps);
```

General description

The c32rtomb() function converts a wide character of type char32_t to a multibyte character, and returns the number of bytes stored in the *s* (including any shift sequences).

If *s* is not a null pointer, the c32rtomb() function determines the number of bytes needed to represent the multibyte character that corresponds to the wide character given by *c32* (including any shift sequences), and stores the multibyte character representation in the array whose first element is pointed to by *s*. At most MB_CUR_MAX bytes are stored. If the *c32* is a null wide character, a null byte is stored, preceded by any shift sequence needed to restore the initial shift state; the resulting state described is the initial conversion state.

If *s* is a null pointer, the c32rtomb() function is equivalent to the call `c32rtomb(buf, L'\0', ps)`, where *buf* is an internal buffer.

If *ps* is a null pointer, c32rtomb() uses its own internal object to track the shift state. Otherwise **ps* must be a valid mbstate_t object. An mbstate_t object **ps* can be initialized to the initial state by assigning 0 to it, or by calling `c32rtomb(NULL, L'\0', ps)`.

Usage notes

1. To use the c32rtomb() function, compile the source code with the LANGLVL(EXTC1X) option.
2. The result *s* for stateful multibyte encodings, such as EBCDIC MBCS, might leave out shift bytes according to the conversion state. The first DBCS character in the output sequence has only shift-out character, and the following characters have neither shift-out nor shift-in. The ending shift-in will not be produced until an SBCS character or a null wide character is encountered.
3. The c32rtomb() function only supports the CCSIDs that are provided by Unicode Services.
4. The Unicode combining characters are not supported, and will be converted to substitute character of target CCSID.
5. The result of converting multiple string alternately in one thread by using multiple mbstate_t objects (including the internal one) is undefined.

Returned value

The c32rtomb() function returns the number of bytes stored in the array object (including any shift sequences). When *c32* is not a valid wide character, an encoding error occurs. The function stores the vlaue of the macro EILSEQ in errno and returns (size_t)(-1); the conversion state is unspecified.

Example

```
#include <stdio.h>
#include <stdlib.h>
#include <uchar.h>

int main(void)
{
   char32_t in = U'a';
   mbstate_t st = 0;
   char out[MB_CUR_MAX];
   int rc, i;
   rc = c32rtomb(out, in, &st);
   if (rc < 0) {
      perror("c32rtomb() fails to convert");
      exit(-1);
   }
   printf(" c32: 0x%04x \n", in);
   printf(" return code: %d \n", rc);
   printf(" mb character: ");
   for (i=0; i<rc; i++)
      printf(" 0x%02x", out[i]);
   printf("\n");
   return 0;
}
```

Output:

```
c32: 0x00000061
return code: 1
mb character:  0x81
```

Related information

- "Internationalization: Locales and Character Sets" in *z/OS XL C/C++ Programming Guide*
- "Appendix A. Description of CCSIDs" in *z/OS Unicode Services User's Guide and Reference*
- "uchar.h" on page 82

- “setlocale() — Set locale” on page 1552
- “c16rtomb() — Convert a char16_t character to a multibyte character” on page 223
- “mbrtoc32() — Convert a multibyte character to a char32_t character” on page 1047

__cabend() — Terminate the process with an abend

Standards

Standards / Extensions	C or C++	Dependencies
Language Environment	both	AMODE 64

Format

```
#include <ctest.h>

void  __cabend(int abendcode, int reasoncode,int clean_up);
```

General description

Causes an abnormal process termination and returns an abend code.

Note: When TRAP(OFF) is specified, __cabend() behaves as if clean_up was set to 0.

Parameter
Description

abendcode
Numeric value for the user abend code.

reasoncode
Numeric value of the reason code.

clean_up
Specifies whether normal process cleanup should be performed with the type of dump the user requires.

0 - Issue the abend without cleanup
1 - Issue the abend with cleanup honoring the TERMTHDACT
runtime option that the user has specified
2 - Issue the abend with cleanup honoring the TERMTHDACT
runtime option for system dump of the
user address space but always suppressing the CEEDUMP
3 - Issue the abend with cleanup honoring the TERMTHDACT
runtime option but always suppressing
both the system dump and the CEEDUMP
4 - Issue the abend with cleanup honoring the TERMTHDACT
runtime option for CEEDUMP but always
suppressing the system dump
5 - Issue the abend with cleanup forcing a system dump of
the user address space but not specifying the
TERMTHDACT runtime option

cabs(), cabsf(), cabsl() — Calculate the complex absolute value

Standards

Standards / Extensions	C or C++	Dependencies
C99 Single UNIX Specification, Version 3	both	z/OS V1R7 a compiler that is designed to support C99

Format

```
#include <complex.h>

double cabs(double complex z);
float cabsf(float complex z);
long double cabsl(long double complex z);
```

General description

The cabs() family of functions compute the complex absolute value (also called norm, modules, or magnitude) of *z*.

Note: The following table shows the viable formats for these functions. See “IEEE binary floating-point” on page 94 for more information about IEEE Binary Floating-Point.

Function	Hex	IEEE
cabs	X	X
cabsf	X	X
cabsl	X	X

Returned value

The cabs functions return the complex absolute value.

Example

```
/*
 * This example calculates the complex absolute
 * value of complex number 'z'
 */
#include <complex.h>
#include <stdio.h>

void main()
{
   double complex z=3.5 + I*2.21;
   double res;

   res = cabs(z);
   printf("cabs(%f + I*%f) = %f\n",creal(z), cimag(z),res);
}

/*
 * Output:
 * cabs(3.5 + I*2.21) = 4.139336
 */
```

Related information

- “complex.h” on page 19
- “cimag(), cimagf(), cimagl() — Calculate the complex imaginary part” on page 280
- “creal(), crealf(), creall() — Calculate the complex real part” on page 340

cacos(), cacosf(), cacosl() — Calculate the complex arc cosine

Standards

Standards / Extensions	C or C++	Dependencies
C99 Single UNIX Specification, Version 3	both	z/OS V1R7 a compiler that is designed to support C99

Format

```
#include <complex.h>

double complex cacos(double complex z);
float complex cacosf(float complex z);
long double complex cacosl(long double complex z);
```

General description

The cacos() family of functions compute the complex arc cosine of *z*, with branch cuts outside the interval [-1, +1] along the real axis.

Note: The following table shows the viable formats for these functions. See “IEEE binary floating-point” on page 94 for more information about IEEE Binary Floating-Point.

Function	Hex	IEEE
cacos	X	X
cacosf	X	X
cacosl	X	X

Returned value

The cacos() family of functions return the complex arc cosine value, in the range of a strip, mathematically unbounded along the imaginary axis and in the interval [0, *π*] along the real axis.

Example

```
/*
 *  This example calculates the complex arc-cosine of
 *  complex number 'z'
 */
#include <complex.h>
#include <stdio.h>

void main()
{

   long double complex zl=3.5 + I*2.21;
   double complex zd=(double complex)zl;
   float complex zf=(float complex)zl;
   long double resl;
   double resd;
   float resf;
   char *func = "cacos";

   printf("Example of the %s complex function\n",func);
   resd = cacos(zd);
   resf = cacosf(zf);
   resl = cacosl(zl);
   printf("\t%s(%f + I*%f) = %f\n",func, creal(zd), cimag(zd),resd);
   printf("\t%sf(%f + I*%f) = %f\n",func, crealf(zf), cimagf(zf),resf);
   printf("\t%sl(%Lf + I*%Lf) = %Lf\n",func, creall(zl), cimagl(zl),resl);
}
```

```
Output:

Example of the cacos complex function
        cacos(3.500000 + I*2.210000) = 0.576628
        cacosf(3.500000 + I*2.209999) = 0.576627
        cacosl(3.500000 + I*2.210000) = 0.576628
```

Related information

- "complex.h" on page 19
- "cacosh(), cacoshf(), cacoshl() — Calculate the complex arc hyperbolic cosine"
- "catan(), catanf(), catanl() — Calculate the complex arc tangent" on page 236
- "ccos(), ccosf(), ccosl() — Calculate the complex cosine" on page 244
- "ccosh(), ccoshf(), ccoshl() — Calculate the complex hyperbolic cosine" on page 245

cacosh(), cacoshf(), cacoshl() — Calculate the complex arc hyperbolic cosine

Standards

Standards / Extensions	C or C++	Dependencies
C99 Single UNIX Specification, Version 3	both	z/OS V1R7 a compiler that is designed to support C99

Format

```
#include <complex.h>

double complex cacosh(double complex z);
float complex cacoshf(float complex z);
long double complex cacoshl(long double complex z);
```

General description

The cacosh() family of functions compute the complex arc hyperbolic cosine of *z*, with a branch cut at values less than 1 along the real axis.

Note: The following table shows the viable formats for these functions. See "IEEE binary floating-point" on page 94 for more information about IEEE Binary Floating-Point.

Function	Hex	IEEE
cacosh	X	X
cacoshf	X	X
cacoshl	X	X

Returned value

The cacosh() family of functions return the complex arc hyperbolic cosine value, in the range of a half-strip, non-negative value along the real axis and in the interval [-i *π*, +i *π*] along the imaginary axis.

Example

For an example of a similar function see cacos(), cexp() or cpow().

Related information

- “complex.h” on page 19
- “cacos(), cacosf(), cacosl() — Calculate the complex arc cosine” on page 229
- “ccosh(), ccoshf(), ccoshl() — Calculate the complex hyperbolic cosine” on page 245
- “ccos(), ccosf(), ccosl() — Calculate the complex cosine” on page 244
- “csin(), csinf(), csinl() — Calculate the complex sine” on page 348
- “csinh(), csinhf(), csinhl() — Calculate the complex hyperbolic sine” on page 349

calloc() — Reserve and initialize storage

Standards

Standards / Extensions	C or C++	Dependencies
ISO C POSIX.1 XPG4 XPG4.2 C99 Single UNIX Specification, Version 3	both	

Format

```
#include <stdlib.h>

void *calloc(size_t num, size_t size);
```

General description

Reserves storage space for an array of *num* elements, each of length *size* bytes. The calloc() function then gives all the bits of each element an initial value of 0.

calloc() returns a pointer to the reserved space. The storage space to which the returned value points is aligned for storage of any type of object.

This function is also available to C applications in free-standing System Programming C (SPC) Facilities applications.

Special behavior for C++: The C++ keywords new and delete are not interoperable with calloc(), free(), malloc(), or realloc().

Returned value

If successful, calloc() returns the pointer to the area of memory reserved.

If there is not enough space to satisfy the request or if *num* or *size* is 0, calloc() returns NULL. If calloc() returns NULL because there is not enough storage, it sets errno to one of the following values:

Error Code
Description

ENOMEM
Insufficient memory is available

Example

CELEBC01

```
/* CELEBC01

   This example prompts for the number of array entries required
   and then reserves enough space in storage for the entries.

   If &calloc. is successful, the example prints out each entry;
   otherwise, it prints out an error message.

 */
#include <stdio.h>
#include <stdlib.h>

int main(void)
{
  long * array;    /* start of the array */
  long * index;    /* index variable     */
  int        i;    /* index variable     */
  int      num;    /* number of entries in the array */

  printf( "Enter the number of elements in the array\n" );
  scanf( "%i", &num );

  /* allocate num entries */
  if ( (index = array = (long *)calloc( num, sizeof( long ))) != NULL )
  {

    for ( i = 0; i < num; ++i )           /* put values in array    */
       *index++ = i;                      /* using pointer notation */

    for ( i = 0; i < num; ++i )           /* print the array out    */
      printf( "array[ %i ] = %i\n", i, array[i] );
  }
  else
  { /* out of storage */
    printf( "Out of storage\n" );
    abort();
  }
}
```

Output

```
Enter the size of the array
array[ 0 ] = 0
array[ 1 ] = 1
array[ 2 ] = 2
```

Related information

- See the topic about using the system programming C facilities in *z/OS XL C/C++ Programming Guide*.
- “stdlib.h” on page 70
- “free() — Free a block of storage” on page 618
- “malloc() — Reserve storage block” on page 1031
- “realloc() — Change reserved storage block size” on page 1391

carg(), cargf(), cargl() — Calculate the argument

Standards

Standards / Extensions	C or C++	Dependencies
C99 Single UNIX Specification, Version 3	both	z/OS V1R7 a compiler that is designed to support C99

Format

```
#include <complex.h>

double carg(double complex z);
float cargf(float complex z);
long double cargl(long double complex z);
```

General description

The carg() family of functions compute the argument (phase angle) of *z*, with a branch cut along the negative real axis.

Note: The following table shows the viable formats for these functions. See “IEEE binary floating-point” on page 94 for more information about IEEE Binary Floating-Point.

Function	Hex	IEEE
carg	X	X
cargf	X	X
cargl	X	X

Returned value

The carg() family of functions return the value of the argument in the interval [-*π*, +*π*].

Example

For an example of a similar function see cacos(), cexp() or cpow().

Related information

- "complex.h" on page 19
- "creal(), crealf(), creall() — Calculate the complex real part" on page 340
- "cimag(), cimagf(), cimagl() — Calculate the complex imaginary part" on page 280
- "conj(), conjf(), conjl() — Calculate the complex conjugate" on page 307
- "cproj(), cprojf(), cprojl() — Calculate the projection" on page 338

casin(), casinf(), casinl() — Calculate the complex arc sine

Standards

Standards / Extensions	C or C++	Dependencies
C99 Single UNIX Specification, Version 3	both	z/OS V1R7 a compiler that is designed to support C99

Format

```
#include <complex.h>

double complex casin(double complex z);
float complex casinf(float complex z);
long double complex casinl(long double complex z);
```

General description

The casin() family of functions compute the complex arc sine of *z*, with branch cuts outside the interval [-1, +1] along the real axis.

Note: The following table shows the viable formats for these functions. See "IEEE binary floating-point" on page 94 for more information about IEEE Binary Floating-Point.

Function	Hex	IEEE
casin	X	X
casinf	X	X
casinl	X	X

Returned value

The casin() family of functions return the complex arc sine value, in the range of a strip, mathematically unbounded along the imaginary axis and in the interval [-π/2, + π/2] along the real axis.

Related information

- "complex.h" on page 19
- "casinh(), casinhf(), casinhl() — Calculate the complex arc hyperbolic sine" on page 235
- "catan(), catanf(), catanl() — Calculate the complex arc tangent" on page 236
- "csin(), csinf(), csinl() — Calculate the complex sine" on page 348
- "csinh(), csinhf(), csinhl() — Calculate the complex hyperbolic sine" on page 349

casinh(), casinhf(), casinhl() — Calculate the complex arc hyperbolic sine

Standards

Standards / Extensions	C or C++	Dependencies
C99 Single UNIX Specification, Version 3	both	z/OS V1R7 a compiler that is designed to support C99

Format

```
#define _ISOC99_SOURCE
#include <complex.h>

double complex casinh(double complex z);
float complex casinhf(float complex z);
long double complex casinhl(long double complex z);
```

General description

The casinh() family of functions compute the complex arc hyperbolic sine of *z*, with branch cuts outside the interval [-i, +i] along the imaginary axis.

Note: The following table shows the viable formats for these functions. See “IEEE binary floating-point” on page 94 for more information about IEEE Binary Floating-Point.

Function	Hex	IEEE
casinh	X	X
casinhf	X	X
casinhl	X	X

Returned value

The casinh() family of functions return the complex arc hyperbolic sine value, in the range of a strip, mathematically unbounded along the real axis and in the interval [-i $\pi/2$, +i $\pi/2$] along the imaginary axis.

Example

For an example of a similar function see cacos(), cexp() or cpow().

Related information

- “complex.h” on page 19
- “casin(), casinf(), casinl() — Calculate the complex arc sine” on page 234
- “cacosh(), cacoshf(), cacoshl() — Calculate the complex arc hyperbolic cosine” on page 230
- “cacos(), cacosf(), cacosl() — Calculate the complex arc cosine” on page 229
- “csin(), csinf(), csinl() — Calculate the complex sine” on page 348
- “csinh(), csinhf(), csinhl() — Calculate the complex hyperbolic sine” on page 349

catan(), catanf(), catanl() — Calculate the complex arc tangent

Standards

Standards / Extensions	C or C++	Dependencies
C99 Single UNIX Specification, Version 3	both	z/OS V1R7 a compiler that is designed to support C99

Format

```
#include <complex.h>

double complex catan(double complex z);
float complex catanf(float complex z);
long double complex catanl(long double complex z);
```

General description

The catan() family of functions compute the complex arc tangent of *z*, with branch cuts outside the interval [-i, +i] along the imaginary axis.

Note: The following table shows the viable formats for these functions. See “IEEE binary floating-point” on page 94 for more information about IEEE Binary Floating-Point.

Function	Hex	IEEE
catan	X	X
catanf	X	X
catanl	X	X

Returned value

The catan() family of functions return the complex arc tangent value, in the range of a strip, mathematically unbounded along the imaginary axis and in the interval [-*π*/2, +*π*/2] along the real axis.

Example

For an example of a similar function see cacos(), cexp() or cpow().

Related information

- “complex.h” on page 19
- “casin(), casinf(), casinl() — Calculate the complex arc sine” on page 234
- “casinh(), casinhf(), casinhl() — Calculate the complex arc hyperbolic sine” on page 235
- “catanh(), catanhf(), catanhl() — Calculate the complex arc hyperbolic tangent” on page 237
- “csin(), csinf(), csinl() — Calculate the complex sine” on page 348
- “csinh(), csinhf(), csinhl() — Calculate the complex hyperbolic sine” on page 349

catanh(), catanhf(), catanhl() — Calculate the complex arc hyperbolic tangent

Standards

Standards / Extensions	C or C++	Dependencies
C99 Single UNIX Specification, Version 3	both	z/OS V1R7 a compiler that is designed to support C99

Format

```
#include <complex.h>

double complex catanh(double complex z);
float complex catanhf(float complex z);
long double complex catanhl(long double complex z);
```

General description

The catanh() family of functions compute the complex arc hyperbolic tangent of *z*, with branch cuts outside the interval [-1, +1] along the real axis.

Note: The following table shows the viable formats for these functions. See “IEEE binary floating-point” on page 94 for more information about IEEE Binary Floating-Point.

Function	Hex	IEEE
catanh	X	X
catanhf	X	X
catanhl	X	X

Returned value

The catanh() family of functions return the complex arc hyperbolic tangent value, in the range of a strip, mathematically unbounded along the real axis and in the interval [-i $\pi/2$, +i $\pi/2$] along the imaginary axis.

Example

For an example of a similar function see cacos(), cexp() or cpow().

Related information

- “complex.h” on page 19
- “catan(), catanf(), catanl() — Calculate the complex arc tangent” on page 236
- “catanh(), catanhf(), catanhl() — Calculate the complex arc hyperbolic tangent”
- “ctan(), ctanf(), ctanl()— Calculate the complex tangent” on page 353
- “csin(), csinf(), csinl() — Calculate the complex sine” on page 348
- “csinh(), csinhf(), csinhl() — Calculate the complex hyperbolic sine” on page 349

catclose() — Close a message catalog descriptor

Standards

Standards / Extensions	C or C++	Dependencies
XPG4 XPG4.2 Single UNIX Specification, Version 3	both	

Format

```
#define _XOPEN_SOURCE
#include <nl_types.h>

int catclose(nl_catd catd);
```

General description

The catclose() function closes the message catalog identified by catd. If a catalog is opened more than once in the same process, a use count is incremented. catclose() decrements this use count. When the use count reaches zero then the file descriptor for that catalog is closed.

Returned value

If successful, catclose() returns 0.

If unsuccessful, catclose() returns -1 and sets errno to one of the following values:

Error Code
Description

EBADF
The catalog descriptor is not valid.

EINTR
catclose() was interrupted by a signal.

Related information

- "nl_types.h" on page 57
- "catgets() — Read a program message"
- "catopen() — Open a message catalog" on page 240

catgets() — Read a program message

Standards

Standards / Extensions	C or C++	Dependencies
XPG4 XPG4.2 Single UNIX Specification, Version 3	both	

Format

```
#define _XOPEN_SOURCE
#include <nl_types.h>

char *catgets(nl_catd catd, int set_id, int msg_id, const char *s);
```

General description

The catgets() function attempts to read message msg_id, in set set_id, from the message catalog identified by catd. The catd argument is a message catalog descriptor returned from an earlier call to catopen(). The s argument points to a default message string which will be returned by catgets() if it cannot retrieve the identified message.

When message source files are processed by the gencat command, the CODESET used to create them is saved in the resulting message catalog. The catgets() function interrogates this CODESET value to see if it differs from the CODESET value of the current locale. If it does differ then catgets() uses the iconv() function to convert the message text coming from the message catalog into the codeset of the current locale. The default message string (s) is not affected by this conversion. If iconv() does not support the conversion specified by the two CODESETs then the default message string is returned.

Returned value

If the identified message is retrieved successfully, catgets() returns a pointer to an internal buffer area containing the NULL-terminated message string.

If unsuccessful, catgets() returns *s* and sets errno to one of the following values:

Error Code
    Description

EBADF
    The *catd* argument is not a valid message catalog descriptor open for reading.

EINTR
    The read operation was terminated due to the receipt of a signal, and no data was transferred.

Special behavior for z/OS UNIX Services:

Error Code
    Description

EINVAL
    May be returned for several reasons:

- The message catalog identified by catd is not a valid message catalog, or has been corrupted. Ensure that the message catalog was created using the z/OS UNIX gencat command.
- iconv() does not support the conversion between the codeset of the message catalog and that of the current locale. To check the codeset that the message catalog was created in, look for the codeset name at offset 28 into the message catalog.

ENOMSG
    The message identified by set_id and msg_id is not in the message catalog.

Related information

- “nl_types.h” on page 57
- “catclose() — Close a message catalog descriptor” on page 238
- “catopen() — Open a message catalog” on page 240

catopen() — Open a message catalog

Standards

Standards / Extensions	C or C++	Dependencies
XPG4 XPG4.2 Single UNIX Specification, Version 3	both	

Format

```
#define _XOPEN_SOURCE
#include <nl_types.h>

nl_catd catopen(const char *name, int oflag);
```

General description

The catopen() function opens a message catalog and returns a message catalog descriptor. The name argument specifies the name of the message catalog to be opened. If name contains a "/", then name specifies a complete name for the message catalog. Otherwise, the environment variable NLSPATH is used with name substituted for %N (see the XBD specification, Chapter 6, Environment Variables). If NLSPATH does not exist in the environment, or if a message catalog cannot be found in any of the components specified by NLSPATH, then the default path of "/usr/lib/nls/msg/%L/%N" is used. The "%L" component of this default path is replaced by the setting of LC_MESSAGES if the value of oflag is NL_CAT_LOCALE, or the LANG environment variable if oflag is 0. A change in the setting of the LANG or LC_MESSAGES will have no effect on existing open catalogs.

A message catalog descriptor remains valid in a process until that process closes it, or a successful call to one of the exec functions. When a message catalog is opened the FD_CLOEXEC flag will be set. See “fcntl() — Control open file descriptors” on page 474. Portable applications must assume that message catalog descriptors are not valid after a call to one of the exec functions.

If a catalog is opened more than once in the same process, a use count is incremented. When the use count reaches zero, by using catclose() to close the catalog, then the file descriptor for that catalog is closed.

Returned value

If successful, catopen() returns a message catalog descriptor for use on subsequent calls to catgets() and catclose().

If unsuccessful, catopen() returns (nl_catd)-1 and sets errno to one of the following values:

Error Code
Description

EACCES
Search permission is denied for the component of the path prefix of the message catalog or read permission is denied for the message catalog.

EMFILE
OPEN_MAX file descriptors are currently open in the calling process.

ENAMETOOLONG
The length of the path name of the message catalog exceeds **PATH_MAX**, or a path name component is longer than **NAME_MAX**.

ENFILE
Too many files are currently open in the system.

ENOENT
The message catalog does not exist or the name argument points to an empty string.

ENOMEM
Insufficient storage space is available.

ENOTDIR
A component of the path prefix of the message catalog is not a directory.

Related information

- “nl_types.h” on page 57
- “catclose() — Close a message catalog descriptor” on page 238
- “catgets() — Read a program message” on page 238

cbrt(), cbrtf(), cbrtl() — Calculate the cube root

Standards

Standards / Extensions	C or C++	Dependencies
XPG4.2 C99 Single UNIX Specification, Version 3 C++ TR1 C99	both	

Format

```
#define _XOPEN_SOURCE_EXTENDED 1
#include <math.h>

double cbrt(double x);
```

C99:

```
#define _ISOC99_SOURCE
#include <math.h>

float cbrtf(float x);
long double cbrtl(long double x);
```

C++ TR1 C99:

```
#define _TR1_C99
#include <math.h>

float cbrt(float x);
long double cbrt(long double x);
```

General description

The cbrt() function calculates the real cube root of its argument *x*.

Note: The following table shows which functions work in IEEE Binary Floating-Point format and which work in hexadecimal floating-point format. See "IEEE binary floating-point" on page 94 for more information about IEEE Binary Floating-Point.

Function	Hex	IEEE
cbrt	X	X
cbrtf	X	X
cbrtl	X	X

Returned value

The cbrt functions return x to the 1/3 power.

cbrt() does not fail.

Related information

- "math.h" on page 44

cbrtd32(), cbrtd64(), cbrtd128() — Calculate the cube root

Standards

Standards / Extensions	C or C++	Dependencies
C/C++ DFP	both	z/OS V1.11

Format

```
#define __STDC_WANT_DEC_FP__
#include <math.h>

_Decimal32  cbrtd32(_Decimal32 x);
_Decimal64  cbrtd64(_Decimal64 x);
_Decimal128 cbrtd128(_Decimal128 x);

_Decimal32  cbrt(_Decimal32 x);      /* C++ only */
_Decimal64  cbrt(_Decimal64 x);      /* C++ only */
_Decimal128 cbrt(_Decimal128 x);     /* C++ only */
```

General description

The cbrt() function calculates the real cube root of its argument *x*.

Notes:

1. These functions work in IEEE decimal floating-point format. See "IEEE decimal floating-point" on page 95IEEE Decimal Floating-Point for more information.
2. To use IEEE decimal floating-point, the hardware must have the Decimal Floating-Point Facility installed.

Example

```
/* CELEBC52

   This example illustrates the cbrtd64() function.

*/
```

```
#define  __STDC_WANT_DEC_FP__
#include <math.h>
#include <stdio.h>

void main(void)
{
   _Decimal64 x, y;

   x = 1000.0DD;
   y = cbrtd64(x);

   printf("cbrtd64( %Df ) = %Df\n", x, y);
}
```

Returned value

The cbrt functions return *x* to the 1/3 power.

cbrt() does not fail.

Related information

- “math.h” on page 44

cclass() — Return characters in a character class

Standards

Standards / Extensions	C or C++	Dependencies
C Library	both	

Format

```
#include <collate.h>

int cclass(char *class, collel_t **list);
```

General description

Finds all the collating elements of the class, *class*. The list is updated to point to the array of collating elements found. The list is valid until the next call to setlocale().

The function supports user-defined character classes. In C Library programs, the function also supports POSIX.2 character classes.

Returned value

If successful, cclass() returns the number of elements in the list pointed to by *list*.

If the first argument specifies a class that does not exist in the LC_CTYPE category of the current locale, cclass() returns -1.

Example

CELEBC02

```
/* CELEBC02  */
#include <stdio.h>
#include <collate.h>

int main(void)
```

```
{
   collel_t *list;      /* ptr to the digit class collation weights   */
   int       weights;   /* no. of class collation class weights found */
   int       i;

   weights = cclass("digit", &list);

   printf("weights=%d\n", weights);
   for (i=0; i<weights; i++)
      printf("*(list + %d) = %d\n", i, *(list  + i) );
}
```

Related information

- “collate.h” on page 19
- “locale.h” on page 40
- “Internationalization: Locales and Character Sets” in *z/OS XL C/C++ Programming Guide*
- “collequiv() — Return a list of equivalent collating elements” on page 295
- “collorder() — Return list of collating elements” on page 297
- “collrange() — Calculate the range list of collating elements” on page 298
- “colltostr() — Return a string for a collating element” on page 299
- “getmccoll() — Get next collating element from string” on page 734
- “getwmccoll() — Get next collating element from wide string” on page 803
- “ismccollel() — Identify a multicharacter collating element” on page 914
- “maxcoll() — Return maximum collating element” on page 1039
- “setlocale() — Set locale” on page 1552
- “strtocoll() — Return collating element for string” on page 1763
- “wctype() — Obtain handle for character property classification” on page 2056

ccos(), ccosf(), ccosl() — Calculate the complex cosine

Standards

Standards / Extensions	C or C++	Dependencies
C99 Single UNIX Specification, Version 3	both	z/OS V1R7 a compiler that is designed to support C99

Format

```
#include <complex.h>

double complex ccos(double complex z);
float complex ccosf(float complex z);
long double complex ccosl(long double complex z);
```

General description

The ccos() family of functions compute the complex cosine of *z*.

Note: The following table shows the viable formats for these functions. See “IEEE binary floating-point” on page 94 for more information about IEEE Binary Floating-Point.

Function	Hex	IEEE
ccos	X	X
ccosf	X	X

Function	Hex	IEEE
ccosl	X	X

Returned value

The ccos() family of functions return the complex cosine value.

Example

For an example of a similar function see cacos(), cexp() or cpow().

Related information

- "complex.h" on page 19
- "ccosh(), ccoshf(), ccoshl() — Calculate the complex hyperbolic cosine"
- "casinh(), casinhf(), casinhl() — Calculate the complex arc hyperbolic sine" on page 235
- "casin(), casinf(), casinl() — Calculate the complex arc sine" on page 234
- "csin(), csinf(), csinl() — Calculate the complex sine" on page 348
- "csinh(), csinhf(), csinhl() — Calculate the complex hyperbolic sine" on page 349

ccosh(), ccoshf(), ccoshl() — Calculate the complex hyperbolic cosine

Standards

Standards / Extensions	C or C++	Dependencies
C99 Single UNIX Specification, Version 3	both	z/OS V1R7 a compiler that is designed to support C99

Format

```
#define _ISOC99_SOURCE
#include <complex.h>

double complex ccosh(double complex z);
float complex ccoshf(float complex z);
long double complex ccoshl(long double complex z);
```

General description

The ccosh() family of functions compute the complex hyperbolic cosine of *z*.

Note: The following table shows the viable formats for these functions. See "IEEE binary floating-point" on page 94 for more information about IEEE Binary Floating-Point.

Function	Hex	IEEE
ccosh	X	X
ccoshf	X	X
ccoshl	X	X

Returned value

The ccosh() family of functions return the complex hyperbolic cosine value.

Example

For an example of a similar function see cacos(), cexp() or cpow().

Related information

- "complex.h" on page 19
- "ccos(), ccosf(), ccosl() — Calculate the complex cosine" on page 244
- "cacosh(), cacoshf(), cacoshl() — Calculate the complex arc hyperbolic cosine" on page 230
- "cacos(), cacosf(), cacosl() — Calculate the complex arc cosine" on page 229
- "csin(), csinf(), csinl() — Calculate the complex sine" on page 348
- "csinh(), csinhf(), csinhl() — Calculate the complex hyperbolic sine" on page 349

__CcsidType() — Return coded character set ID type

Standards

Standards / Extensions	C or C++	Dependencies
z/OS UNIX	both	z/OS V1R2

Format

```
#include <_Ccsid.h>

__csType __CcsidType(__ccsid_t Ccsid);
```

General description

__CcsidType() returns a __csType value which indicates the corresponding coded character set ID type.

Returned value

If *Ccsid* is valid, __CcsidType() returns one of the following __csType values, which are defined in <_Ccsid.h>:

- _CSTYPE_EBCDIC
- _CSTYPE_ASCII
- _CSTYPE_UCS2
- _CSTYPE_UTF8
- _CSTYPE_UTF16
- _CSTYPE_UTF32

If *Ccsid* is not valid, __CcsidType() returns _CSTYPE_INVALID.

Related information

- "_Ccsid.h" on page 17
- "__CSNameType() — Return codeset name type" on page 350
- "__toCcsid() — Convert codeset name to coded character set ID" on page 1896
- "__toCSName() — Convert coded character set ID to codeset name" on page 1897

cds() — Compare double and swap

Standards

Standards / Extensions	C or C++	Dependencies
C Library	both	

Format

```
#include <stdlib.h>

int cds(cds_t *oldptr, cds_t *curptr, cds_t newwords);
```

General description

The cds() built-in function compares the 8-byte value pointed to by *oldptr* to the 8-byte value pointed to by *curptr*. If they are equal, the 8-byte value *newwords* is copied into the location pointed to by *curptr*. If they are unequal, the value pointed to by *curptr* is copied into the location pointed to by *oldptr*.

To avoid infringing on the user's name space, this nonstandard function is exposed only when you use the compiler option, LANGLVL(EXTENDED). When you use LANGLVL(EXTENDED) any relevant information in the header is also exposed.

The function uses the COMPARE DOUBLE AND SWAP (CDS) instructions, which can be used in multiprogramming or multiprocessing environments to serialize access to counters, flags, control words, and other common storage areas. For a detailed description, see the appendixes in the *z/Architecture Principles of Operation* on number representation and instruction.

Returned value

cds() returns 0 if the 8-byte value pointed to by *oldptr* is equal to the 8-byte value pointed to by *curptr*.

Otherwise cds() returns 1.

Related information

- *z/Architecture Principles of Operation*
- “stdlib.h” on page 70
- “cs() — Compare and swap” on page 346

cdump() — Request a main storage dump

Standards

Standards / Extensions	C or C++	Dependencies
C Library	both	

Format

```
#include <ctest.h>

int cdump(char *dumptitle);
```

General description

1. Creates a display of the activation stack, by calling trace in the same way as the ctrace() function does.
2. Displays the Language Environment-formatted dump.
3. If the source file was compiled with `TEST(SYM)`, cdump() displays the contents of the user's variables. The output is identified with *dumptitle*. See the `CEE3DMP` Language Environment callable service in *z/OS Language Environment Programming Guide, SA38-0682*, to determine where the output is written to.

To avoid infringing on the user's name space, this nonstandard function has two names. One name is prefixed with two underscore characters, and one name is not. The name without the prefix underscore characters is exposed only when you use LANGLVL(EXTENDED).

To use this function, you must either invoke the function using its external entry point name (that is, the name that begins with two underscore characters), or compile with LANGLVL(EXTENDED). When you use LANGLVL(EXTENDED) any relevant information in the header is also exposed.

When cdump() is invoked from a user routine, the C/C++ library issues an OS SNAP macro to obtain a dump of virtual storage. The first invocation of cdump() results in a SNAP identifier of 0. For each successive invocation, the ID is increased by one to a maximum of 256, after which the ID is reset to 0.

The output of the dump is directed to the CEESNAP data set. The DD definition for CEESNAP is as follows:

```
//CEESNAP DD SYSOUT= *
```

If the data set is not defined, or is not usable for any reason, cdump() returns a failure code of 1. This occurs even if the call to CEE3DMP is successful. For more information see "Debugging C/C++ Routines " in *z/OS Language Environment Debugging Guide*.

Returned value

If successful, cdump() returns 0.

If unsuccessful, cdump() returns nonzero.

Related information

- “ctest.h” on page 21
- “csnap() — Request a condensed dump” on page 351
- “ctrace() — Request a traceback” on page 363

ceil(), ceilf(), ceill() — Round up to integral value

Standards

Standards / Extensions	C or C++	Dependencies
ISO C POSIX.1 XPG4 XPG4.2 ISO/ANSI C++ C99 Single UNIX Specification, Version 3 C++ TR1 C99	both	

Format

```
#include <math.h>

double ceil(double x);
float ceil(float x);                    /* C++ only */
long double ceil(long double x);        /* C++ only */
float ceilf(float x);
long double ceill(long double x);
```

General description

Computes the smallest integer that is greater than or equal to *x*.

Note: These functions work in both IEEE Binary Floating-Point and hexadecimal floating-point formats. See “IEEE binary floating-point” on page 94 for more information about IEEE Binary Floating-Point.

Returned value

Returns the calculated value as a double, float, or long double value.

If there is an overflow, the function sets errno to ERANGE and returns HUGE_VAL.

Special behavior for IEEE: The ceil() functions are always successful.

Example

CELEBC04

```
/* CELEBC04

   This example sets y to the smallest integer greater than
   1.05, and then to the smallest integer greater than -1.05.

   The results are 2.0 and -1.0, respectively.

 */
#include <math.h>
#include <stdio.h>
int main(void)
{
   double y, z;
   y = ceil(1.05);       /* y = 2.0 */
   z = ceil(-1.05);      /* z = -1.0 */
   printf("y = %f\n z = %f\n", y, z);
}
```

Related information

- “math.h” on page 44
- “floor(), floorf(), floorl() — Round down to integral value” on page 548

ceild32(), ceild64(), ceild128() — Round up to integral value

Standards

Standards / Extensions	C or C++	Dependencies
C/C++ DFP	both	z/OS V1.8

Format

```
#define __STDC_WANT_DEC_FP__
#include <math.h>

_Decimal32  ceild32(_Decimal32 x);
_Decimal64  ceild64(_Decimal64 x);
_Decimal128 ceild128(_Decimal128 x);
_Decimal32  ceil(_Decimal32 x); /* C++ only */
_Decimal64  ceil(_Decimal64 x); /* C++ only */
_Decimal128 ceil(_Decimal128 x); /* C++ only */
```

General description

Computes the smallest integer that is greater than or equal to *x*.

Notes:

1. To use IEEE decimal floating-point, the hardware must have the Decimal Floating-Point Facility installed.
2. These functions work in IEEE decimal floating-point format. See "IEEE Decimal Floating-Point" for more information.

Returned value

These functions are always successful.

Example

```
/* CELEBC50

   This example illustrates the ceild32() function.

   This example sets y to the smallest integer greater than
   1.05, and then to the smallest integer greater than -1.05.

   The results are 2.0 and -1.0, respectively.

*/

#define  __STDC_WANT_DEC_FP__
#include <math.h>
#include <stdio.h>

int main(void)
{
   _Decimal32 y, z;

   y = ceild32(+1.05DF);      /* y = +2.0 */
   z = ceild32(-1.05DF);      /* z = -1.0 */

   printf("ceild32(+1.05) = % Hf\n"
          "ceild32(-1.05) = % Hf\n", y, z);
}
```

Related information

- “math.h” on page 44
- “floord32(), floord64(), floord128() — Round down to integral value” on page 549

__certificate() — Register, deregister, or authenticate a digital certificate

Standards

Standards / Extensions	C or C++	Dependencies
z/OS UNIX	both	OS/390 V2R7

Format

```
#define _OPEN_SYS
#include <unistd.h>

int __certificate(int function_code,
                  int certificate_length,
                  char *certificate,...);
```

General description

The __certificate() function allows the user to register or deregister a digital certificate with or from the userid that is associated with the current security environment, or to authenticate a security environment using a digital certificate in lieu of a userid/password combination.

The function takes at least the following arguments:

function_code
Specifies one of the following functions:

__CERTIFICATE_REGISTER
Register the passed certificate to the user. No new security environment is created, and no authentication of the user is done.

__CERTIFICATE_DEREGISTER
Deregister the passed certificate from the user. Certificate must have been previously registered to the user.

__CERTIFICATE_AUTHENTICATE
As of z/OS V1R4, authenticate the passed certificate for this caller. The certificate must have already been registered.

certificate_length
The length of the digital certificate. A zero length will cause -1 return value with errno set to EINVAL.

certificate
The certificate must be a single BER encoded X.509 certificate. PKCS7, PEM, or Base64 encoded certificates are allowed.

Note: Only a single BER encoded X.509 certificate is supported for the authenticate function.

As of z/OS V1R4, the __CERTIFICATE_AUTHENTICATE function code requires the following additional parameters to be specified o nthe function call:

buflen (size_t)
Specifies the size of the buffer pointed to by buf. Up to buflen bytes of userid (including the NULL terminator) will be copied into the buffer. Note that truncation may occur if the buffer is too small. The buffer size should be large enough for any userid on the system. A value less than 1 will cause -1 return value with errno set to EINVAL.

buf (char *)
Pointer to character buffer where _certificate() will place the userid associated with the digital certificate. A value of NULL will cause -1 return value with errno set to EINVAL.

Usage notes

1. The _certificate function is intended for servers that support the automatic registration of certificates for clients they are supporting (on the World Wide Web for example).
2. The __CERTIFICATE_REGISTER function code will associate the passed certificate with whatever user identity is present. If the task level identity is present the certificate is associated with the task. Task level security can be created by pthread_security_np(), __login() or by any other means of creating a task level ACEE. If no task level identity (ACEE) is present, the certificate will be associated with the address space identity.
3. The __certificate() function calls the z/OS z/OS UNIX System Services BPX1SEC service. For a more detailed description of the BPX1SEC service, see *z/OS UNIX System Services Programming: Assembler Callable Services Reference*.

Restrictions

A security manager supporting digital certificate registration and deregistration must be installed and operational.

Returned value

If successful, __certificate() returns 0.

If unsuccessful, __certificate() returns -1 and sets errno to one of the following values:

Error Code
Description

EACCES
Permission is denied.

EINVAL
A parameter is not valid.

EMVSERR
An MVS environmental error or internal error occurred.

EMVSSAF2ERR
An error occurred in the security product. Certificate is already defined for another process or certificate is not valid or certificate does not meet required format. Also, realized when an internal error has occurred.

ENOSYS
The function is not implemented or installed.

EPERM
The operation was not permitted. Calling process may not be authorized in BPX.DAEMON facility class.

Use __errno2() to obtain a more detailed reason code (in most cases) when __certificate() fails.

Related information

- "unistd.h" on page 82
- "__login(), __login_applid() — Create a new security environment for process" on page 995
- "pthread_security_np(), pthread_security_applid_np() — Create or delete thread-level security" on page 1324

cexp(), cexpf(), cexpl() — Calculate the complex exponential

Standards

Standards / Extensions	C or C++	Dependencies
C99 Single UNIX Specification, Version 3	both	z/OS V1R7 a compiler that is designed to support C99

Format

```
#include <complex.h>

double complex cexp(double complex z);
float complex cexpf(float complex z);
long double complex cexpl(long double complex z);
```

General description

The cexp() family of functions compute the complex base-e exponential of *z*.

Note: The following table shows the viable formats for these functions. See "IEEE binary floating-point" on page 94 for more information about IEEE Binary Floating-Point.

Function	Hex	IEEE
cexp	X	X
cexpf	X	X
cexpl	X	X

Returned value

The cexp() family of functions return the complex base-e exponential value.

Example

```
/*
 * This example illustrates the complex exponential
 * function
 */
#include <complex.h>
#include <stdio.h>

void main()
{
   double complex z=6.0146 + I*(-2.41958),
                  res;

   res = cexp(z);
   printf("cexp(%f + (%f)*I) = %f + (%f)*I\n",creal(z), cimag(z),creal(res),cimag(res));
}

Output:

cexp(6.014600 + (-2.419580)*I) = -307.216850 + (-270.545937)*I
```

Related information

- “complex.h” on page 19
- “clog(), clogf(), clogl() — Calculate the complex natural logarithm” on page 287

cfgetispeed() — Determine the input baud rate

Standards

Standards / Extensions	C or C++	Dependencies
POSIX.1 XPG4 XPG4.2 Single UNIX Specification, Version 3	both	

Format

```
#define _POSIX_SOURCE
#include <termios.h>

speed_t cfgetispeed(const struct termios *termptr);
```

General description

Extracts the input baud rate from the termios structure indicated by **termptr*. The termios structure contains information about a terminal. A program should first use tcgetattr() to get the termios structure, and then use cfgetispeed() to extract the speed from the structure. The program can then use cfgetispeed() to set a new baud rate in the structure and tcsetattr() to pass the changed value to the system.

Although in a z/OS UNIX application valid speeds can be set with cfsetispeed() and passed to the system with tcsetattr(), the speed has no effect on the operation of a pseudoterminal. However, the operation will have an effect if issued for an OCS remote terminal.

Returned value

cfgetispeed() returns a code indicating the baud rate; see Table 19. These codes are defined in the termios.h header file and have an unsigned integer type.

There are no documented errno values.

Table 19. Baud Rate Codes

B0	Hang up
B50	50 baud
B75	75 baud
B110	110 baud
B134	134.5 baud
B150	150 baud
B200	200 baud
B300	300 baud
B600	600 baud
B1200	1200 baud
B1800	1800 baud
B2400	2400 baud
B4800	4800 baud
B9600	9600 baud
B19200	19,200 baud
B38400	38,400 baud

Example

CELEBC05

```
/* CELEBC05

   This example determines the speed of stdin.

 */
#define _POSIX_SOURCE
#include <termios.h>
#include <stdio.h>

char *see_speed(speed_t speed) {
  static char   SPEED[20];
  switch (speed) {
    case B0:       strcpy(SPEED, "B0");
                   break;
    case B50:      strcpy(SPEED, "B50");
                   break;
    case B75:      strcpy(SPEED, "B75");
                   break;
    case B110:     strcpy(SPEED, "B110");
                   break;
    case B134:     strcpy(SPEED, "B134");
                   break;
    case B150:     strcpy(SPEED, "B150");
                   break;
    case B200:     strcpy(SPEED, "B200");
```

```
                 break;
    case B300:     strcpy(SPEED, "B300");
                   break;
    case B600:     strcpy(SPEED, "B600");
                   break;
    case B1200:    strcpy(SPEED, "B1200");
                   break;
    case B1800:    strcpy(SPEED, "B1800");
                   break;
    case B2400:    strcpy(SPEED, "B2400");
                   break;
    case B4800:    strcpy(SPEED, "B4800");
                   break;
    case B9600:    strcpy(SPEED, "B9600");
                   break;
    case B19200:   strcpy(SPEED, "B19200");
                   break;
    case B38400:   strcpy(SPEED, "B38400");
                   break;
    default:       sprintf(SPEED, "unknown (%d)", (int) speed);
  }
  return SPEED;
}

main() {
  struct termios term;
  speed_t speed;

  if (tcgetattr(0, &term) != 0)
    perror("tcgetattr() error");
  else {
    speed = cfgetispeed(&term);
    printf("cfgetispeed() says the speed of stdin is %s\n",
           see_speed(speed));
  }
}
```

Output

```
cfgetispeed() says the speed of stdin is B0
```

Related information

- "termios.h" on page 78
- "cfgetospeed() — Determine the output baud rate"
- "cfsetispeed() — Set the input baud rate in the termios" on page 258
- "cfsetospeed() — Set the output baud rate in the termios" on page 260
- "tcgetattr() — Get the attributes for a terminal" on page 1830
- "tcsetattr() — Set the attributes for a terminal" on page 1843

cfgetospeed() — Determine the output baud rate

Standards

Standards / Extensions	C or C++	Dependencies
POSIX.1 XPG4 XPG4.2 Single UNIX Specification, Version 3	both	

Format

```
#define _POSIX_SOURCE
#include <termios.h>

speed_t cfgetospeed(const struct termios *termptr);
```

General description

Extracts the output baud rate from the `termios` structure indicated by **termptr*. The `termios` structure contains information about a terminal. A program should first use tcgetattr() to get the `termios` structure, and then use cfgetospeed() to extract the speed from the structure. The program can then use cfgetospeed() to set a new baud rate in the structure and tcsetattr() to pass the changed value to the system.

Although in a z/OS UNIX application valid speeds can be set with cfsetospeed() and passed to the system with tcsetattr(), the speed has no effect on the operation a pseudoterminal. However, the operation will have an effect if issued for an OCS remote terminal.

Returned value

cfgetospeed() returns a code indicating the baud rate. The codes are defined in the termios.h header file and have an unsigned integer type. Table 19 on page 255 shows the codes to set the baud rate.

There are no documented errno values.

Example

CELEBC06

```
/* CELEBC06

   This example determines the speed of stdout.

 */
#define _POSIX_SOURCE
#include <termios.h>
#include <stdio.h>

char *see_speed(speed_t speed) {
  static char   SPEED[20];
  switch (speed) {
    case B0:       strcpy(SPEED, "B0");
                   break;
    case B50:      strcpy(SPEED, "B50");
                   break;
    case B75:      strcpy(SPEED, "B75");
                   break;
    case B110:     strcpy(SPEED, "B110");
                   break;
    case B134:     strcpy(SPEED, "B134");
                   break;
    case B150:     strcpy(SPEED, "B150");
                   break;
    case B200:     strcpy(SPEED, "B200");
                   break;
    case B300:     strcpy(SPEED, "B300");
                   break;
    case B600:     strcpy(SPEED, "B600");
                   break;
    case B1200:    strcpy(SPEED, "B1200");
```

```
                   break;
    case B1800:    strcpy(SPEED, "B1800");
                   break;
    case B2400:    strcpy(SPEED, "B2400");
                   break;
    case B4800:    strcpy(SPEED, "B4800");
                   break;
    case B9600:    strcpy(SPEED, "B9600");
                   break;
    case B19200:   strcpy(SPEED, "B19200");
                   break;
    case B38400:   strcpy(SPEED, "B38400");
                   break;
    default:       sprintf(SPEED, "unknown (%d)", (int) speed);
  }
  return SPEED;
}

main() {
  struct termios term;
  speed_t speed;

  if (tcgetattr(1, &term) != 0)
    perror("tcgetattr() error");
  else {
    speed = cfgetospeed(&term);
    printf("cfgetospeed() says the speed of stdout is %s\n",
           see_speed(speed));
  }
}
```

Output

```
cfgetospeed() says the speed of stdout is B0
```

Related information

- "termios.h" on page 78
- "cfgetispeed() — Determine the input baud rate" on page 254
- "cfsetispeed() — Set the input baud rate in the termios"
- "cfsetospeed() — Set the output baud rate in the termios" on page 260
- "tcgetattr() — Get the attributes for a terminal" on page 1830
- "tcsetattr() — Set the attributes for a terminal" on page 1843

cfsetispeed() — Set the input baud rate in the termios

Standards

Standards / Extensions	C or C++	Dependencies
POSIX.1 XPG4 XPG4.2 Single UNIX Specification, Version 3	both	

Format

```
#define _POSIX_SOURCE
#include <termios.h>

int cfsetispeed(struct termios *termptr, speed_t speed);
```

General description

Specifies a new input baud rate for the termios control structure, **termptr*. cfsetispeed() records this new baud rate in the control structure but does not actually change the terminal device file. The program must call tcsetattr() to modify the terminal device file to reflect the settings in the termios structure.

A program should first use tcgetattr() to get the termios structure. Then it should use cfsetispeed() to set the speed in termios and tcsetattr() to pass the modified termios structure to the system.

Although in a z/OS UNIX application valid speeds can be set with cfsetispeed() and passed to the system with tcsetattr(), the speed has no effect on the operation of a pseudoterminal. However, the operation will have an effect if issued for an OCS remote terminal.

The *speed* argument indicates the new baud rate with one of the following codes, defined in the termios.h header file. The codes have an unsigned integer type. Table 19 on page 255 shows the codes to set the baud rate.

Returned value

If successful, cfsetispeed() sets the baud rate in the control structure and returns 0.

If unsuccessful, cfsetispeed() returns -1 and sets errno to one of the following values:

Error Code
Description

EINVAL
An unsupported value for *speed*

Example

CELEBC07

```
/* CELEBC07

   This example specifies a new input baud rate.

 */
#define _POSIX_SOURCE
#include <termios.h>
#include <stdio.h>

main() {
  struct termios term;

  if (tcgetattr(0, &term) != 0)
    perror("tcgetattr() error");
  else if (cfsetispeed(&term, B0) != 0)
    perror("cfsetispeed() error");
  else if (tcsetattr(0, TCSANOW, &term) != 0)
    perror("tcsetattr() error");
}
```

Related information

- “termios.h” on page 78
- “cfgetispeed() — Determine the input baud rate” on page 254
- “cfgetospeed() — Determine the output baud rate” on page 256

- “cfsetospeed() — Set the output baud rate in the termios”
- “tcgetattr() — Get the attributes for a terminal” on page 1830
- “tcsetattr() — Set the attributes for a terminal” on page 1843

cfsetospeed() — Set the output baud rate in the termios

Standards

Standards / Extensions	C or C++	Dependencies
POSIX.1 XPG4 XPG4.2 Single UNIX Specification, Version 3	both	

Format

```
#define _POSIX_SOURCE
#include <termios.h>

int cfsetospeed(struct termios *termptr, speed_t speed);
```

General description

Specifies a new output baud rate for the `termios` control structure, **termptr*. cfsetospeed() records this new baud rate in the control structure, but does not actually change the terminal device file. The program must call tcsetattr() to modify the terminal device file to reflect the settings in the `termios` structure.

A program should first use tcgetattr() to get the `termios` structure. It should then use cfsetospeed() to set the speed in `termios` and tcsetattr() to pass the modified `termios` structure to the system.

Although in a z/OS UNIX application valid speeds can be set with cfsetospeed() and passed to the system with tcsetattr(), the speed has no effect on the operation of a pseudoterminal. However, the operation will have an effect if issued for an OCS remote terminal.

The *speed* argument should be a code indicating the new baud rate. These codes are defined in the termios.h header file and have an unsigned integer type. Table 19 on page 255 shows the codes to set the baud rate.

Returned value

If successful, cfsetospeed() sets the baud rate for the structure and returns 0.

If unsuccessful, cfsetospeed() returns -1 and sets errno to one of the following values:

Error Code
    Description

EINVAL
    The value *speed* is not supported by the hardware or software.

Example

CELEBC08

```
/* CELEBC08

   This example specifies a new output baud rate.

 */
#define _POSIX_SOURCE
#include <termios.h>
#include <stdio.h>

main() {
  struct termios term;

  if (tcgetattr(1, &term) != 0)
    perror("tcgetattr() error");
  else if (cfsetospeed(&term, B38400) != 0)
    perror("cfsetospeed() error");
  else if (tcsetattr(1, TCSANOW, &term) != 0)
    perror("tcsetattr() error");
}
```

Related information

- “termios.h” on page 78
- “cfgetispeed() — Determine the input baud rate” on page 254
- “cfgetospeed() — Determine the output baud rate” on page 256
- “cfsetispeed() — Set the input baud rate in the termios” on page 258
- “tcgetattr() — Get the attributes for a terminal” on page 1830
- “tcsetattr() — Set the attributes for a terminal” on page 1843

__chattr() — Change the attributes of a file or directory

Standards

Standards / Extensions	C or C++	Dependencies
z/OS UNIX	both	z/OS V1R2

Format

```
#define _OPEN_SYS_FILE_EXT 1
#include <sys/stat.h>

int __chattr(char* pathname, attrib_t *attributes, int attributes_len);
```

General description

The __chattr() function modifies the attributes that are associated with a file. It can be used to change the mode, owner, access time, modification time, change time, reference time, audit flags, general attribute flags, file tag, and file format and size. The file to be impacted is defined by the *pathname* argument.

The *attributes* argument is the address of an attrib_t structure which is used to identify the attributes to be modified and the new values desired. The attrib_t type is an f_attributes structure as defined in <sys/stat.h> for use with the __chattr() function. For proper behavior the user should ensure that this structure has been initialized to zeros before it is populated. Available elements of the f_attributes structure are defined in Table 20:

Table 20. Struct f_attributes Element Descriptions

Element	Data Type	General Description
Bit Flags Indicating Which Attributes to Change		

Table 20. Struct f_attributes Element Descriptions (continued)

Element	Data Type	General Description
att_modechg:1	int	1=Change to mode indicated
att_ownerchg:1	int	1=Change to Owner indicated
att_setgen:1	int	1=Set General Attributes
att_trunc:1	int	1=Truncate Size
att_atimechg:1	int	1=Change the Atime
att_atimetod:1	int	1=Change Atime to Cur.Time
att_mtimechg:1	int	1=Change the Mtime
att_mtimetod:1	int	1=Change Mtime to Cur.Time
att_maaudit:1	int	1=Modify auditor audit info
att_muaudit:1	int	1=Modify user audit info
att_ctimechg:1	int	1=Change the Ctime
att_ctimetod:1	int	1=Change Ctime to Cur.Time
att_reftimechg:1	int	1=Change the RefTime
att_reftimetod:1	int	1=Change RefTime to Cur.Time
att_filefmtchg:1	int	1=Change File Format
att_filetagchg:1	int	1=Change File Tag
att_seclabelchg:1	int	1=Change Seclabel
Modified Values for Indicated Attributes to Change		
att_mode	mode_t	File Mode
att_uid	int	User ID of the owner of the file
att_gid	int	Group ID of the Group of the file
att_sharelibmask:1	int	1=Shared Library Mask
att_noshareasmask:1	int	1=No Shareas Flag Mask
att_apfauthmask:1	int	1=APF Authorized Flag Mask
att_progctlmask:1	int	1=Prog. Controlled Flag Mask
att_sharelib:1	int	1=Shared Library Flag
att_noshareas:1	int	1=No Shareas Flag
att_apfauth:1	int	1=APF Authorized Flag
att_progctl:1	int	1=Program Controlled Flag
att_size	off_t	File size
att_atime	time_t	Time of last access
att_mtime	time_t	Time of last data modification
att_auditoraudit	int	Area for auditor audit info
att_useraudit	int	Area for user audit info
att_ctime	time_t	Time of last file status change
att_reftime	time_t	Reference Time
att_filefmt	char	File Format
att_filetag	struct file_tag	File Tag
att_seclabel	char	Security Label

Note: If you set att_nodelfilesmask, att_sharelibmask, att_nodelfiles, att_sharelib, att_noshareasmask, att_apfauthmask, att_progctlmask, att_noshareas, att_apfauth or att_progctl, then att_setgen must also be set. The att_setgen flag is a required indicator when setting "general" attributes.

Returned value

If successful, __chattr() returns 0.

If unsuccessful, __chattr() returns -1 and sets errno to one of the following values:

Error Code
Description

EACCES
The calling process did not have appropriate permissions. Possible reasons include:

- The calling process was attempting to set access time or modification time to current time, and the effective UID of the calling process does not match the owner of the file; the process does not have write permission for the file; or the process does not have appropriate privileges.
- The calling process was attempting to truncate the file, and it does not have write permission for the file.

ECICS
An attempt was made to change file tag attributes under non-OTE CICS and file tagging is not supported in that environment.

EFBIG
The calling process was attempting to change the size of a file, but the specified length is greater than the maximum file size limit for the process.

EINVAL
The attributes structure containing the requested changes is not valid.

ELOOP
A loop exists in symbolic links that were encountered during resolution of the *pathname* argument. This error is issued if more than 24 symbolic links are detected in the resolution of *pathname*.

ENAMETOOLONG
pathname is longer than 1023 characters, or a component of the *pathname* is longer than 255 characters. (File name truncation is not supported.)

ENOENT
No file named *pathname* was found.

ENOTDIR
Some component of *pathname* is not a directory.

EPERM
The operation is not permitted for one of the following reasons:

- The calling process was attempting to change the mode or the file format but the effective UID of the calling process does not match the owner of the file, and the calling process does not have appropriate privileges.
- The calling process was attempting to change the owner but it does not have appropriate privileges.

- The calling process was attempting to change the general attribute bits but it does not have write permission for the file.
- The calling process was attempting to set a time value (not current time) but the effective UID does not match the owner of the file, and it does not have appropriate privileges.
- The calling process was attempting to set the change time or reference time to current time but it does not have write permission for the file.
- The calling process was attempting to change auditing flags but the effective UID of the calling process does not match the owner of the file and the calling process does not have appropriate privileges.
- The calling process was attempting to change the Security Auditor's auditing flags but the user does not have auditor authority.

EROFS
pathname specifies a file that is on a read-only file system.

Example

```
#define _POSIX_SOURCE 1
#define _OPEN_SYS_FILE_EXT 1
#include <stdio.h>
#include <fcntl.h>
#include <sys/stat.h>

int main(int argc, char *argv[]) {

   int fd;
   attrib_t myAtt;
   struct stat attr;
   char filename[] = "chattr.testfile";

   /* Create an empty file */
   if ( (fd = creat(filename, S_IRWXU | S_IRGRP | S_IROTH)) < 0) {
      perror("Failed to create testfile");
      exit(1);
   }
   close(fd);

   /* Clear myAtt structure */
   memset(&myAtt, 0, sizeof(myAtt));

   /* Update myAtt to request file tag change and set file tag values */
   myAtt.att_filetagchg = 1;
   myAtt.att_filetag.ft_ccsid = 12345;
   myAtt.att_filetag.ft_txtflag = 1;

   /* Change Attributes */
   if ( __chattr(filename, &myAtt, sizeof(myAtt)) != 0 ) {
      perror("Failed to change attributes for testfile");
      exit(2);
   }

   /* Verify Change */
   if ( stat(filename,&attr) !=0 ) {
      perror("Failed to acquire statistics for testfile");
      exit(3);
   }

   if ((attr.st_tag.ft_ccsid == 12345)
       && (attr.st_tag.ft_txtflag == 1)) {
      printf("File attributes changed successfully\n");
   }

}
```

Related information

- "sys/stat.h" on page 75
- "__fchattr() — Change the attributes of a file or directory by file descriptor" on page 465
- "__lchattr() — Change the attributes of a file or directory when they point to a symbolic or external link" on page 936

chaudit() — Change audit flags for a file by path

Standards

Standards / Extensions	C or C++	Dependencies
z/OS UNIX	both	

Format

```
#define _OPEN_SYS 1
#include <sys/stat.h>

int chaudit(const char *pathname, unsigned int flags,
            unsigned int option);
```

General description

Changes the audit flags for a file to indicate the type of requests the security product should audit. chaudit() can change user audit flags or security auditor audit flags, depending on the *option* specified.

pathname is the name of the file for which the audit flags are to be changed.

flags is the setting for the audit flags:

AUDTREADFAIL
Audit the failing read requests.

AUDTREADSUCC
Audit the successful read requests.

AUDTWRITEFAIL
Audit the failing write requests.

AUDTWRITESUCC
Audit the successful write requests.

AUDTEXECFAIL
Audit the failing execute or search requests.

AUDTEXECSUCC
Audit the successful execute or search requests. The bitwise inclusive-OR of any or all of these can be used to set more than one type of auditing.

option indicates whether the user audit flags or the security-auditor audit flags are to be changed:

AUDT_USER (0)
Change user flags. The user must be the file owner or have appropriate authority to change the user audit flags for a file.

AUDT_AUDITOR (1)
Change security auditor audit flags. The user must have security-auditor authority to modify the security auditor audit flags for a file.

Returned value

If successful, chaudit() returns 0.

If unsuccessful, chaudit() returns -1 and sets errno to one of the following values:

Error Code
Description

EACCES
The calling process does not have permission to search some component of *pathname*.

EINVAL
option is not AUDT_USER or AUDT_AUDITOR.

ELOOP
A loop exists in symbolic links. This error is issued if the number of symbolic links detected in the resolution of *pathname* is greater than POSIX_SYMLOOP (a value defined in the limits.h header file).

ENAMETOOLONG
pathname is longer than **PATH_MAX** characters or a component of *pathname* is longer than **NAME_MAX** characters while _POSIX_NO_TRUNC is in effect. For symbolic links, the length of the *pathname* string substituted for a symbolic link exceeds **PATH_MAX**. The **PATH_MAX** and **NAME_MAX** values are determined using pathconf().

ENOENT
There is no file named *pathname*, or *pathname* is an empty string.

ENOTDIR
A component of the path prefix is not a directory.

EPERM
The effective user ID (UID) of the calling process does not match the owner of the file, and the calling process does not have appropriate privileges.

EROFS
pathname specifies a file that is on a read-only file system.

Example

CELEBC09

```
/* CELEBC09

   This example changes the audit flags.

 */

#define _OPEN_SYS
#include <fcntl.h>
#include <sys/stat.h>
#include <sys/types.h>
#include <unistd.h>
#undef _OPEN_SYS
#include <stdio.h>
```

```
main() {
  int fd;
  char fn[]="chaudit.file";

  if ((fd = creat(fn, S_IRUSR|S_IWUSR)) < 0)
    perror("creat() error");
  else {
    close(fd);
    if (chaudit(fn, AUDTREADFAIL, AUDT_USER) != 0)
      perror("chaudit() error");
    unlink(fn);
  }
}
```

Related information

- "sys/stat.h" on page 75
- "access() — Determine whether a file can be accessed" on page 111
- "chmod() — Change the mode of a file or directory" on page 272
- "chown() — Change the owner or group of a file or directory" on page 275
- "fchaudit() — Change audit flags for a file by descriptor" on page 466
- "stat() — Get file information" on page 1720

chdir() — Change the working directory

Standards

Standards / Extensions	C or C++	Dependencies
POSIX.1 XPG4 XPG4.2 Single UNIX Specification, Version 3	both	

Format

```
#define _POSIX_SOURCE
#include <unistd.h>

int chdir(const char *pathname);
```

General description

Makes *pathname* your new working directory.

Returned value

If successful, chdir() changes the working directory and returns 0.

If unsuccessful, chdir() does not change the working directory, returns -1, and sets errno to one of the following values:

Error Code
Description

EACCES
The process does not have search permission on one of the components of *pathname*.

ELOOP
A loop exists in symbolic links. This error is issued if the number of

symbolic links detected in the resolution of *pathname* is greater than POSIX_SYMLOOP (a value defined in the limits.h header file).

ENAMETOOLONG
pathname is longer than **PATH_MAX** characters, or some component of *pathname* is longer than **NAME_MAX** characters while _POSIX_NO_TRUNC is in effect. For symbolic links, the length of the *pathname* string substituted for a symbolic link exceeds **PATH_MAX**. The **PATH_MAX** and **NAME_MAX** values are determined using pathconf().

ENOENT
pathname is an empty string, or the specified directory does not exist.

ENOTDIR
Some component of *pathname* is not a directory.

Example

CELEBC10

```
/* CELEBC10 */
#define _POSIX_SOURCE
#include <unistd.h>
#undef _POSIX_SOURCE
#include <stdio.h>

main() {
  if (chdir("/tmp") != 0)
    perror("chdir() to /tmp failed");
  if (chdir("/chdir/error") != 0)
    perror("chdir() to /chdir/error failed");
}
```

Output

```
chdir() to /chdir/error failed: No such file or directory
```

Related information

- “limits.h” on page 39
- “unistd.h” on page 82
- “closedir() — Close a directory” on page 291
- “getcwd() — Get path name of the working directory” on page 697
- “mkdir() — Make a directory” on page 1068
- “opendir() — Open a directory” on page 1158
- “readdir() — Read an entry from a directory” on page 1382
- “rewinddir() — Reposition a directory stream to the beginning” on page 1446

__check_resource_auth_np() — Determine access to MVS resources

Standards

Standards / Extensions	C or C++	Dependencies
z/OS UNIX	both	

Format

```
#include <unistd.h>

int __check_resource_auth_np( char *principal_uuid,
                              char *cell_uuid,
                              char *userid,
                              char *security_class,
                              char *entity_name,
                              int access_type);
```

General description

The __check_resource_auth_np() function is used to check the access a user has to an MVS resource.

For authorization to use this function, the caller must have read permission to the BPX.SERVER Facility class, or if BPX.SERVER is not defined, the caller must be a superuser (UID=0).

The user identity can be specified in several forms. The identities are scanned in the order below, and the access check will be made with the first identity that is found:
- userid
- principal UUID and if known, a cell UUID
- caller's thread-level (task) security context, if one exists
- caller's process-level (address space) security context

Note:
- When no identity is specified by the caller and the caller's task has an ACEE created with pthread_security_np()for a SURROGATE (non-password) client, both the task and address space level ACEEs are used in determining the type of access permitted to a resource.
- The __check_resource_auth_np() function supports the general resources only. In particular, the security_class parameter can not specify DATASET. For system using RACF, the class name specified must be in the RACF class descriptor table.

The parameters supported are:

principal_uuid
Specifies a 36-byte principal UUID. A value of NULL indicates that no principal UUID is specified.

cell_uuid
Specifies a 36-byte cell UUID. A value of NULL indicates that no cell UUID is specified.

userid Specifies a user ID. A value of NULL indicates that no user ID is specified. The *userid* must be 1-8 characters in length.

security_class
Specifies the name of a class of resources. The access check will be made on a resource in this security class. The *security_class* must be 1-8 characters in length.

entity_name
Specifies the name of a resource profile name. The access check will be made on the resource specified by the resource profile name. The *entity_name* must be 1-246 characters in length.

access Specifies a numeric value that identifies the type of access to check for. Possible access values are:

__READ_RESOURCE
check if the specified user has read access to the resource.

__UPDATE_RESOURCE
check if the specified user has update access to the resource.

__CONTROL_RESOURCE
check if the specified user has control access to the resource.

__ALTER_RESOURCE
check if the specified user has alter access to the resource.

Returned value

If successful, __check_resource_auth_np() returns 0.

If unsuccessful, __check_resource_auth_np() returns -1 and sets errno to one of the following values:

Error Code
Description

EINVAL
One of the following errors was detected:
- Aaccess_type specified is undefined.
- Userid was not 1 to 8 characters in length.
- Security_class was not 1 to 8 characters in length.
- Eentity_name was not 1 to 246 characters in length.

EMVSERR
An MVS internal or environmental error occurred.

EMVSSAF2ERR
One of the following errors was detected:
- Received an unexpected return code for the security product.
- The security product detected an error in the input parameters.
- An internal error occurred in the security product.

ENOSYS
One of the following errors was detected:
- No security product is installed on the system.
- The security product does not have support for this function.

EPERM
One of the following errors was detected:
- The caller is not permitted to use this service.
- Do not have the access_type specified to the resource.
- Not permitted in address spaces where a load from an unauthorized library has been performed.

ESRCH
One of the following errors was detected:
- No mapping exists between a UUID and Userid.
- The resource specified is not defined to the security product.
- The DCEUUIDS class is not active.
- The userid is not defined to the security product.

Related information

- “unistd.h” on page 82

CheckSchEnv() — Check WLM scheduling environment

Standards

Standards / Extensions	C or C++	Dependencies
z/OS UNIX	both	

Format

```
#include <sys/__wlm.h>

int CheckSchEnv(const char *sched_env,
                const char *system_name);
```

General description

The CheckSchEnv() function provides the ability for an application to connect to check the WLM scheduling environment.

***sched_env**
: Points to a 16 byte character string that represents the WLM scheduling environment to be queried. If the environment name is less than 16 characters, the name should be right padded with blanks.

***sys_name**
: Points to a 8 bytes character string that represents the system name to be queried. If the system name is less than 8 characters, the name should be right padded with blanks.

Returned value

If successful, CheckSchEnv() returns 0.

If unsuccessful, CheckSchEnv() returns -1 and sets errno to one of the following values:

Error Code
: **Description**

EFAULT
: An argument of this function contained an address that was not accessible to the caller.

EINVAL
: An argument of this function contained an incorrect value.

EMVSSAF2ERR
: An error occurred in the security product.

EMVSWLMERROR
: The WLM check scheduling environment failed. Use __errno2() to obtain the WLM service reason code for the failure.

EPERM
: The calling thread's address space is not permitted to the BPX.WLMSERVER Facility class. The caller's address space must be permitted to the BPX.WLMSERVER Facility class if it is defined. If BPX.WLMSERVER is not defined, the calling process is not defined as a superuser (UID=0).

Related information

- "sys/__wlm.h" on page 77
- "ConnectServer() — Connect to WLM as a server manager" on page 313
- "ConnectWorkMgr() — Connect to WLM as a work manager" on page 315
- "ContinueWorkUnit() — Continue WLM work unit" on page 323
- "CreateWorkUnit() — Create WLM work unit" on page 344
- "DeleteWorkUnit() — Delete a WLM work unit" on page 378
- "DisconnectServer() — Disconnect from WLM server" on page 383
- "JoinWorkUnit() — Join a WLM work unit" on page 927
- "LeaveWorkUnit() — Leave a WLM work unit" on page 945
- "QueryMetrics() — Query WLM system information" on page 1368
- "QuerySchEnv() — Query WLM scheduling environment" on page 1369

chmod() — Change the mode of a file or directory

Standards

Standards / Extensions	C or C++	Dependencies
POSIX.1 XPG4 XPG4.2 Single UNIX Specification, Version 3	both	

Format

```
#define _POSIX_SOURCE
#include <sys/stat.h>

int chmod(const char *pathname, mode_t mode);
```

General description

Changes the mode of the file or directory specified in *pathname*.

The *mode* argument is created with one of the following symbols defined in the sys/stat.h header file.

Any mode flags that are not defined will be turned off, and the function will be allowed to proceed.

S_IRGRP
Read permission for the file's group.

S_IROTH
Read permission for users other than the file owner.

S_IRUSR
Read permission for the file owner.

S_IRWXG
Read, write, and search or execute permission for the file's group. S_IRWXG is the bitwise inclusive-OR of S_IRGRP, S_IWGRP, and S_IXGRP.

S_IRWXO
Read, write, and search or execute permission for users other than the file owner. S_IRWXO is the bitwise inclusive-OR of S_IROTH, S_IWOTH, and S_IXOTH.

S_IRWXU
Read, write, and search, or execute, for the file owner; S_IRWXG is the bitwise inclusive-OR of S_IRUSR, S_IWUSR, and S_IXUSR.

S_ISGID
Privilege to set group ID (GID) for execution. When this file is run through an exec function, the effective group ID of the process is set to the group ID of the file. The process then has the same authority as the file owner, rather than the authority of the actual invoker.

S_ISUID
Privilege to set the user ID (UID) for execution. When this file is run through an exec function, the effective user ID of the process is set to the owner of the file. The process then has the same authority as the file owner, rather than the authority of the actual invoker.

S_ISVTX
The sticky bit indicating shared text. Keep loaded as an executable file in storage.

S_IWGRP
Write permission for the file's group.

S_IWOTH
Write permission for users other than the file owner.

S_IWUSR
Write permission for the file owner.

S_IXGRP
Search permission (for a directory) or execute permission (for a file) for the file's group.

S_IXOTH
Search permission for a directory, or execute permission for a file, for users other than the file owner.

S_IXUSR
Search permission (for a directory) or execute permission (for a file) for the file owner.

Special behavior for XPG4.2: If a directory is writable and the mode bit S_ISVTX is set on the directory, a process may remove or rename files within that directory only if one or more of the following is true:

- The effective user ID of the process is the same as that of the owner ID of the file.
- The effective user ID of the process is the same as that of the owner ID of the directory.
- The process has appropriate privileges.

A process can set mode bits only if the effective user ID of the process is the same as the file's owner or if the process has appropriate privileges (superuser authority). chmod() automatically clears the S_ISGID bit in the file's mode bits if all these conditions are true:

- The calling process does not have appropriate privileges, that is, superuser authority (UID=0).
- The group ID of the file does not match the group ID or supplementary group IDs of the calling process.

- One or more of the S_IXUSR, S_IXGRP, or S_IXOTH bits of the file mode are set to 1.

Returned value

If successful, chmod() marks for update the `st_ctime` field of the file and returns 0.

If unsuccessful, chmod() returns -1 and sets errno to one of the following values:

Error Code
    Description

EACCES
    The process does not have search permission on some component of the *pathname* prefix.

ELOOP
    A loop exists in symbolic links. This error is issued if the number of symbolic links detected in the resolution of *pathname* is greater than POSIX_SYMLOOP (a value defined in the limits.h header file).

ENAMETOOLONG
    pathname is longer than **PATH_MAX** characters, or some component of *pathname* is longer than **NAME_MAX** characters while _POSIX_NO_TRUNC is in effect. For symbolic links, the length of the *pathname* string substituted for a symbolic link exceeds **PATH_MAX**. The **PATH_MAX** and **NAME_MAX** values are determined using pathconf().

ENOENT
    There is no file named *pathname*, or the *pathname* argument is an empty string.

ENOTDIR
    Some component of the *pathname* prefix is not a directory.

EPERM
    The effective user ID (UID) of the calling process does not match the owner of the file, and the calling process does not have appropriate privileges (superuser authority).

EROFS
    pathname is on a read-only file system.

Example

CELEBC11

```
/* CELEBC11

   This example changes the permission from the file owner to the file's
   group.

 */
#define _POSIX_SOURCE
#include <sys/stat.h>
#include <sys/types.h>
#include <unistd.h>
#undef _POSIX_SOURCE
#include <stdio.h>

main() {
  char fn[]="./temp.file";
  FILE *stream;
  struct stat info;

  if ((stream = fopen(fn, "w")) == NULL)
    perror("fopen() error");
  else {
    fclose(stream);
    stat(fn, &info);
    printf("original permissions were: %08x\n", info.st_mode);
    if (chmod(fn, S_IRWXU|S_IRWXG) != 0)
      perror("chmod() error");
    else {
      stat(fn, &info);
      printf("after chmod(), permissions are: %08x\n", info.st_mode);
    }
    unlink(fn);
  }
}
```

Output

```
original permissions were: 030001b6
after chmod(), permissions are: 030001f8
```

Related information

- “sys/stat.h” on page 75
- “sys/types.h” on page 75
- “chown() — Change the owner or group of a file or directory”
- “fchmod() — Change the mode of a file or directory by descriptor” on page 469
- “mkdir() — Make a directory” on page 1068
- “mkfifo() — Make a FIFO special file” on page 1071
- “open() — Open a file” on page 1152
- “stat() — Get file information” on page 1720

chown() — Change the owner or group of a file or directory

Standards

Standards / Extensions	C or C++	Dependencies
POSIX.1 XPG4 XPG4.2 Single UNIX Specification, Version 3	both	

Format

```
#define _POSIX_SOURCE
#include <unistd.h>

int chown(const char *pathname, uid_t owner, gid_t group);
```

General description

Changes the owner or group (or both) of a file. *pathname* is the name of the file whose owner or group you want to change. *owner* is the user ID (UID) of the new owner of the file. *group* is the group ID (GID) of the new group for the file.

If _POSIX_CHOWN_RESTRICTED is defined in the unistd.h header file, a process can change the group of a file only if one of these is true:

1. The process has appropriate privileges.
2. Or all of the following are true:

a. The effective user ID of the process is equal to the user ID of the file owner.
b. The *owner* argument is equal to the user ID of the file owner or `(uid_t)-1`,
c. The *group* argument is either the effective group ID or a supplementary group ID of the calling process.

If *pathname* is a regular file and one or more of the S_IXUSR, S_IXGRP, or S_IXOTH bits of the file mode are set, chown() clears the set-user-ID (S_ISUID) and set-group-ID (S_ISGID) bits of the file mode and returns successfully.

If *pathname* is not a regular file and one or more of the S_IXUSR, S_IXGRP, or S_IXOTH bits of the file mode are set, chown() clears the set-user-ID (S_ISUID) and set-group-ID (S_ISGID) bits of the file.

When chown() completes successfully, it marks the `st_ctime` field of the file to be updated.

Special behavior for XPG4.2: If *owner* or *group* is specified as `(uid_t)-1` or `(gid_t)-1` respectively, the corresponding ID of the file is unchanged.

Returned value

If successful, chown() updates the owner, group, and change time for the file and returns 0.

If unsuccessful, chown() returns -1 and sets errno to one of the following values:

Error Code
Description

EACCES
The process does not have search permission on some component of the *pathname* prefix.

EINTR
Added for XPG4.2: The chown() function was interrupted by a signal which was caught.

EINVAL
owner or *group* is not a valid user ID (UID) or group ID (GID).

EIO **Added for XPG4.2:** An I/O error occurred while reading or writing to the file system.

ELOOP
A loop exists in symbolic links. This error is issued if the number of symbolic links detected in the resolution of *pathname* is greater than POSIX_SYMLOOP (a value defined in the limits.h header file).

ENAMETOOLONG
pathname is longer than **PATH_MAX** characters, or some component of *pathname* is longer than **NAME_MAX** characters while _POSIX_NO_TRUNC is in effect. For symbolic links, the length of the *pathname* string substituted for a symbolic link exceeds **PATH_MAX**. The **PATH_MAX** and **NAME_MAX** values can be determined using pathconf().

ENOENT
There is no file named *pathname*, or the *pathname* argument is an empty string.

ENOTDIR
Some component of the *pathname* prefix is not a directory.

EPERM
The effective user ID of the calling process does not match the owner of the file, or the calling process does not have appropriate privileges, that is, superuser authority (UID=0).

EROFS
pathname is on a read-only file system.

Example

CELEBC12

```
/* CELEBC12

   This example changes the owner and group of a file.

 */
#define _POSIX_SOURCE
#include <unistd.h>
#include <sys/stat.h>
#include <sys/types.h>
#undef _POSIX_SOURCE
#include <stdio.h>

main() {
  char fn[]="./temp.file";
  FILE *stream;
  struct stat info;

  if ((stream = fopen(fn, "w")) == NULL)
    perror("fopen() error");
  else {
    fclose(stream);
    stat(fn, &info);
    printf("original owner was %d and group was %d\n", info.st_uid,
           info.st_gid);
    if (chown(fn, 25, 0) != 0)
      perror("chown() error");
    else {
      stat(fn, &info);
      printf("after chown(), owner is %d and group is %d\n",
             info.st_uid, info.st_gid);
    }
    unlink(fn);
  }
}
```

Output

```
original owner was 0 and group was 0
after chown(), owner is 25 and group is 0
```

Related information

- “limits.h” on page 39
- “unistd.h” on page 82
- “chmod() — Change the mode of a file or directory” on page 272
- “fchown() — Change the owner or group by file descriptor” on page 471
- “fstat() — Get status information about a file” on page 651
- “lstat() — Get status of file or symbolic link” on page 1022
- “stat() — Get file information” on page 1720

chpriority() — Change the scheduling priority of a process

Standards

Standards / Extensions	C or C++	Dependencies
z/OS UNIX	both	

Format

```
#define _OPEN_SOURCE 2
#include <sys/resource.h>

int chpriority(int which, id_t who, int prioritytype, int priority);
```

General description

The chpriority() function changes the scheduling priority of a process, process group or user.

Processes are specified by the values of the *which* and *who* arguments. The *which* argument may be one of the following values: PRIO_PROCESS, PRIO_PGRP, or PRIO_USER, indicating that the *who* argument is to be interpreted as a process ID, a process group ID, or a user ID, respectively. A 0 (zero) value for the *who* argument specifies the current process, process group or user ID.

If more than one process is specified, the chpriority() function changes the priorities of all of the specified processes.

The default priority is 0; negative priorities cause more favorable scheduling. The range of legal priority values is -20 to 19. If the CPRIO_ABSOLUTE value is specified for the *prioritytype* argument and the priority value specified to chpriority() is less than the system's lowest supported priority value, the system's lowest supported value is used; if it is greater than the system's highest supported value, the system's highest supported value is used. If the CPRIO_RELATIVE value is specified on the *prioritytype* argument, request for values above or below the legal limits result in the priority value being set to the corresponding limit.

The changing of a process's scheduling priority value has the equivalent effect of a process's nice value, since they both represent the process's relative CPU priority. For example, changing one's scheduling priority value using the chpriority() function to its maximum value (19) has the equivalent effect of increasing one's nice value to its maximum value 2*{NZERO}-1, and will be reflected on the nice(), getpriority(), chpriority(), and setpriority() functions.

Only a process with appropriate privilege can lower its priority. In addition to lowering the priority value, a process with appropriate privilege has the ability to change the priority of any process regardless of the process's saved set-user-ID value.

Returned value

If successful, chpriority() returns 0.

If unsuccessful, chpriority() returns -1 and sets errno to one of the following values:

Error Code
Description

EACCES
The priority is being changed to a lower value and the current process does not have the appropriate privilege.

EINVAL
The value of the *which* argument was not recognized, or the value of the *who* argument is not a valid process ID, process group ID or user ID, or the value of the *prioritytype* argument was not recognized.

ENOSYS
The system does not support this function.

EPERM
A process was located, but the save set-user-ID of the executing process does not match the saved set-user-ID of the process whose priority is to be changed.

ESRCH
No process could be located using the *which* and *who* argument values specified.

Related information

- “sys/resource.h” on page 74
- “getpriority() — Get process scheduling priority” on page 755
- “nice() — Change priority of a process” on page 1145
- “setpriority() — Set process scheduling priority” on page 1567

chroot() — Change root directory

Standards

Standards / Extensions	C or C++	Dependencies
XPG4 XPG4.2	both	

Format

```
#define _XOPEN_SOURCE
#include <unistd.h>

int chroot(const char *path);
```

General description

The *path* argument points to a path name naming a directory. The chroot() function causes the named directory to become the root directory, that is the starting point for path searches for path names beginning with /. The process's working directory is unaffected by chroot(). Only a superuser can request chroot().

The dot-dot entry in the root directory is interpreted to mean the root directory. Thus, dot-dot cannot be used to access files outside the subtree rooted at the root directory.

Note:

This function is kept for historical reasons. It was part of the Legacy Feature in Single UNIX Specification, Version 2, but has been withdrawn and is not supported as part of Single UNIX Specification, Version 3.

If it is necessary to continue using this function in an application written for Single UNIX Specification, Version 3, define the feature test macro _UNIX03_WITHDRAWN before including any standard system headers. The macro exposes all interfaces and symbols removed in Single UNIX Specification, Version 3.

Returned value

If successful, chroot() changes the root directory, and returns 0.

If unsuccessful, chroot() returns -1 and sets errno to one of the following values:

Error Code
: **Description**

EACCES
: Search permission is denied for a component of *path*

ELOOP
: A loop exists in symbolic links. This error is issued if the number of symbolic links detected in the resolution of path name is greater than POSIX_SYMLOOP (a value defined in the limits.h header file).

ENAMETOOLONG
: Path name is longer that **PATH_MAX** characters, or some component of path name is longer that **NAME_MAX** characters while _POSIX_NO_TRUNC is in effect. For symbolic links, the length of the path name string substituted for a symbolic link exceeds **PATH_MAX**. The **PATH_MAX** and **NAME_MAX** values are determined using pathconf().

ENOENT
: A component of *path* does not name an existing directory or *path* is an empty string.

ENOTDIR
: A component of the *path* name is not a directory.

EPERM
: The effective user ID does not have appropriate privileges.

Related information

- “unistd.h” on page 82
- “chdir() — Change the working directory” on page 267
- “closedir() — Close a directory” on page 291
- “mkdir() — Make a directory” on page 1068
- “opendir() — Open a directory” on page 1158

cimag(), cimagf(), cimagl() — Calculate the complex imaginary part

Standards

Standards / Extensions	C or C++	Dependencies
C99 Single UNIX Specification, Version 3	both	z/OS V1R7 a compiler that is designed to support C99

Format

```
#include <complex.h>

double cimag(double complex z);
float cimagf(float complex z);
long double cimagl(long double complex z);
```

General description

The cimag() family of functions compute the imaginary part of *z*.

Note: The following table shows the viable formats for these functions. See “IEEE binary floating-point” on page 94 for more information about IEEE Binary Floating-Point.

Function	Hex	IEEE
cimag	X	X
cimagf	X	X
cimagl	X	X

Returned value

The cimag() family of functions return the imaginary part value (as a real).

Example

For an example of a similar function see cacos(), cexp() or cpow().

Related information

- “complex.h” on page 19
- “carg(), cargf(), cargl() — Calculate the argument” on page 233
- “conj(), conjf(), conjl() — Calculate the complex conjugate” on page 307
- “cproj(), cprojf(), cprojl() — Calculate the projection” on page 338
- “creal(), crealf(), creall() — Calculate the complex real part” on page 340

clearenv() — Clear environment variables

Standards

Standards / Extensions	C or C++	Dependencies
POSIX.1a Language Environment	both	

Format

POSIX - C only:

```
#define _POSIX1_SOURCE 2
#include <env.h>

int clearenv(void);
```

Non-POSIX:

```
#include <stdlib.h>

int clearenv(void);
```

General description

Clears all environment variables from the environment table and frees the associated storage.

clearenv() also resets all behavior modified by z/OS XL C/C++ specific environment variables back to their defaults. For example, if a binary file was opened, then it would support seeking by byte offsets, regardless of record format. If the file is a Variable Record format MVS DASD file, then clearing the environment variable causes seeking by encoded values the next time it is opened.

To avoid infringing on the user's name space, the non-POSIX version of this function has two names. One name is prefixed with two underscore characters, and one name is not. The name without the prefix underscore characters is exposed only when you use LANGLVL(EXTENDED).

To use this function, you must either invoke the function using its external entry point name (that is, the name that begins with two underscore characters), or compile with LANGLVL(EXTENDED). When you use LANGLVL(EXTENDED) any relevant information in the header is also exposed.

For details about environment variables, see “Using Environment Variables” in *z/OS XL C/C++ Programming Guide*.

Special behavior for POSIX C: clearenv() can change the value of the pointer `environ`. Therefore, a copy of that pointer made before a call to clearenv() may no longer be valid after the call to clearenv().

Returned value

If successful, clearenv() returns 0.

If unsuccessful, clearenv() returns nonzero and sets errno to one of the following values:

Error Code
    Description

ENOMEM
    The process requires more space than is available.

Example

CELEBC13

```
/* CELEBC13

   This C/MVS example needs to be run with POSIX(ON).
   It clears the process environment variable list.

 */
#define _POSIX_SOURCE 1
#include <env.h>
#include <stdio.h>

extern char **environ;
```

```
int count_env() {
  int num;

  for (num=0; environ[num] != NULL; num++);
  return num;
}

main() {
  printf("before clearenv(), there are %d environment variables\n",
         count_env());
  if (clearenv() != 0)
    perror("clearenv() error");
  else {
    printf("after clearenv(), there are %d environment variables\n",
           count_env());
    setenv("var1", "value1", 1);
    setenv("var-two", "Value Two", 1);
    printf("after setenv()'s, there are %d environment variables\n",
           count_env());
    if (clearenv() != 0)
      perror("clearenv() error");
    else
      printf("after clearenv(), there are %d environment variables\n",
             count_env());
  }
}
```

Output

```
before clearenv(), there are 9 environment variables
after clearenv(), there are 0 environment variables
after setenv()'s, there are 2 environment variables
after clearenv(), there are 0 environment variables
```

CELEBC14

```
/* CELEBC14

   This example is for a non-POSIX environment, and thus will work under
   C++/MVS.

 */
#include <stdio.h>
#include <stdlib.h>

int main(void)
{
   char *x;

   /* set 3 environment variables to "Y" */
   setenv("_EDC_ANSI_OPEN_DEFAULT","Y",1);
   setenv("_EDC_BYTE_SEEK","Y",1);
   setenv("_EDC_COMPAT","3",1);

   /* query the setting of _EDC_BYTE_SEEK */
   x = getenv("_EDC_BYTE_SEEK");

   if (x != NULL)
      printf("_EDC_BYTE_SEEK = %s\n",x);
   else
      printf("_EDC_BYTE_SEEK is undefined\n");

/* clear the environment variable table */
   clearenv();

   /* query the setting of _EDC_BYTE_SEEK */
   x = getenv("_EDC_BYTE_SEEK");
   if (x != NULL)
      printf("_EDC_BYTE_SEEK = %s\n",x);
   else
      printf("_EDC_BYTE_SEEK is undefined\n");
}
```

Output

```
     _EDC_BYTE_SEEK = Y
     _EDC_BYTE_SEEK is undefined
```

Related information

- "Using Environmental Variables" in *z/OS XL C/C++ Programming Guide*
- "env.h" on page 23
- "stdlib.h" on page 70
- "getenv() — Get value of environment variables" on page 704
- "__getenv() — Get an environment variable" on page 705
- "putenv() — Change or add an environment variable" on page 1348
- "setenv() — Add, delete, and change environment variables" on page 1528

clearerr() — Reset error and end of file (EOF)

Standards

Standards / Extensions	C or C++	Dependencies
ISO C POSIX.1 XPG4 XPG4.2 C99 Single UNIX Specification, Version 3 Language Environment	both	

Format

```
#include <stdio.h>

void clearerr(FILE *stream);

#define _OPEN_SYS_UNLOCKED_EXT 1
#include <stdio.h>

void clearerr_unlocked (FILE *stream);
```

General description

Resets the error indicator and EOF indicator for the stream that `stream` points to. Generally, the indicators for a stream remain set until your program calls clearerr() or rewind().

clearerr_unlocked() is functionally equivalent to clearerr() with the exception that it is not thread-safe. This function can safely be used in a multithreaded application if and only if it is called while the invoking thread owns the (FILE*) object, as is the case after a successful call to either the flockfile() or ftrylockfile() function.

Returned value

clearerr() returns no values.

Example

CELEBC15

```
/* CELEBC15

   This example reads a data stream and then checks that a read
   error has not occurred.

 */
#include <stdio.h>

int main(void)
{
   char string[100];
   FILE *stream;
   int eofvalue;

   stream = fopen("myfile.dat", "r");

   /* scan an input stream until an end-of-file character is read */
   while (!feof(stream))
      fscanf(stream,"%s",&string[0]);

   /* print EOF value: will be nonzero */
   eofvalue=feof(stream);
   printf("feof value=%i\n",eofvalue);

   /* print EOF value-after clearerr, will be equal to zero */
   clearerr(stream);
   eofvalue=feof(stream);
   printf("feof value=%i\n",eofvalue);
}
```

Related information

- “stdio.h” on page 68
- “feof() — Test end of file (EOF) indicator” on page 500
- “ferror() — Test for read and write errors” on page 503
- “fseek() — Change file position” on page 639
- “rewind() — Set file position to beginning of file” on page 1445

clock() — Determine processor time

Standards

Standards / Extensions	C or C++	Dependencies
ISO C POSIX.1 XPG4 XPG4.2 C99 Single UNIX Specification, Version 3	both	

Format

```
#include <time.h>

clock_t clock(void);
```

General description

Approximates the processor time used by the program, since the beginning of an implementation-defined time period that is related to the program invocation. To measure the time spent in a program, call the clock() function at the start of the program, and subtract its returned value from the value returned by subsequent calls to clock(). Then, to obtain the time in seconds, divide the value returned by clock() by `CLOCKS_PER_SEC`.

If you use the system() function in your program, do not rely on clock() for program timing, because calls to system() may reset the clock.

In a multithread POSIX C application, if you are creating threads with a function that is based on a POSIX.4a draft standard, the clock() function is thread-scoped.

Returned value

If the time is available and can be represented, clock() returns the calculated time.

If unsuccessful, clock() returns (clock_t)-1. clock() may return -1 when running with STIMER REAL TQE present on MVS/ESA Version 3 Release 1 Modification 2 (or earlier) system.

Special behavior for XPG4: If _XOPEN_SOURCE or _XOPEN_SOURCE_EXTENDED are defined when your application is compiled, CLOCKS_PER_SEC is defined as 1000000. Also, in this case, the following C/370 pragma in the <time.h> header is used to compile your application:

```
#pragma map ( clock(), "@@OCLCK")
```

Because of this pragma, when your application executes, it will attempt to access an XPG4 version of clock() which returns a clock_t value in units of 1000000 CLOCKS_PER_SEC. The XPG4 version of clock() is only available if POSIX(ON) is specified for execution of your application.

If _XOPEN_SOURCE or _XOPEN_SOURCE_EXTENDED are defined when you compile your program AND your application is run with POSIX(OFF), clock() will return `(clock_t)-1`.

If neither _XOPEN_SOURCE or _XOPEN_SOURCE_EXTENDED are defined when you compile your application, the historical C/370 value of CLOCKS_PER_SEC will be used and clock() calls in your application will be mapped to the historical C/370 version of clock() which returns a clock_t value in historical C/370 CLOCKS_PER_SEC units whether your application executes with POSIX(ON) or POSIX(OFF).

Example

```
/*  This example prints the time elapsed since the program was invoked.  */
#include <time.h>
#include <stdio.h>

double time1, timedif;   /* use doubles to show small values */

int main(void)
{
    time1 = (double) clock();            /* get initial time */
    time1 = time1 / CLOCKS_PER_SEC;      /*    in seconds    */
⋮
    /* call clock a second time */
    timedif = ( ((double) clock()) / CLOCKS_PER_SEC) - time1;
    printf("The elapsed time is %f seconds\n", timedif);
}
```

Related information

- "time.h" on page 79
- "time(),time64() — Determine current UTC time" on page 1878

clog(), clogf(), clogl() — Calculate the complex natural logarithm

Standards

Standards / Extensions	C or C++	Dependencies
C99 Single UNIX Specification, Version 3	both	z/OS V1R7 a compiler that is designed to support C99

Format

```
#include <complex.h>

double complex clog(double complex z);
float complex clogf(float complex z);
long double complex clogl(long double complex z);
```

General description

The clog() family of functions compute the complex natural (base-e) logarithm of *z*, with a branch cut along the negative real axis.

Note: The following table shows the viable formats for these functions. See "IEEE binary floating-point" on page 94 for more information about IEEE Binary Floating-Point.

Function	Hex	IEEE
clog	X	X
clogf	X	X
clogl	X	X

Returned value

The clog() family of functions return the complex natural logarithm value, in the range of a strip, mathematically unbounded along the real axis and in the interval [-i *π*, +i *π*] along the imaginary axis.

Example

For an example of a similar function see cacos(), cexp() or cpow().

Related information

- "complex.h" on page 19
- "cexp(), cexpf(), cexpl() — Calculate the complex exponential" on page 253

close() — Close a file

Standards

Standards / Extensions	C or C++	Dependencies
POSIX.1 XPG4 XPG4.2 Single UNIX Specification, Version 3	both	

Format

```
#define _POSIX_SOURCE
#include <unistd.h>

int close(int fildes);
```

X/Open:

```
#define _XOPEN_SOURCE_EXTENDED 1
#include <unistd.h>

int close(int socket);
```

Berkeley sockets:

```
#define _OE_SOCKETS
#include <unistd.h>

int close(int socket);
```

General description

Closes a file descriptor, *fildes*. This frees the file descriptor to be returned by future open() calls and other calls that create file descriptors. The *fildes* argument must represent a hierarchical file system (HFS) file.

When the last open file descriptor for a file is closed, the file itself is closed. If the file's link count is 0 at that time, its space is freed and the file becomes inaccessible.

When the last open file descriptor for a pipe or FIFO file is closed, any data remaining in the pipe or FIFO file is discarded.

close() unlocks (removes) all outstanding record locks that a process has on the associated file.

Behavior for sockets: close() call shuts down the socket associated with the socket descriptor *socket*, and frees resources allocated to the socket. If *socket* refers to an open TCP connection, the connection is closed. If a stream socket is closed when there is input data queued, the TCP connection is reset rather than being cleanly closed.

Parameter	Description
socket	The descriptor of the socket to be closed.

Note: All sockets should be closed before the end of your process. You should issue a shutdown() call before you issue a close() call for a socket.

For AF_INET and AF_INET6 stream sockets (SOCK_STREAM) using SO_LINGER socket option, the socket does not immediately end if data is still present when a close is issued. The following structure is used to set or unset this option, and it can be found in **sys/socket.h**.

```
struct linger {
   int l_onoff;      /* zero=off, nonzero=on */
   int l_linger;     /* time is seconds to linger */
};
```

If the l_onoff switch is nonzero, the system attempts to deliver any unsent messages. If a linger time is specified, the system waits for *n* seconds before flushing the data and terminating the socket.

For AF_UNIX, when closing sockets that were bound, you should also use unlink() to delete the file created at bind() time.

Special behavior for XPG4.2: If a STREAMS-based *fildes* is closed and the calling process was previously registered to receive a SIGPOLL signal for events associated with that STREAM, the calling process will be unregistered for events associated with the STREAM. The last close() for a STREAM causes the STREAM associated with *fildes* to be dismantled. If O_NONBLOCK is not set and there have been no signals posted for the STREAM, and if there is data on the module's write queue, close() waits for an unspecified time (for each module and driver) for any output to drain before dismantling the STREAM. The time delay can be changed using an I_SETCLTIME ioctl() request. If the O_NONBLOCK flag is set, or if there are any pending signals, close() does not wait for output to drain, and dismantles the STREAM immediately.

Note: z/OS UNIX services do not supply any STREAMS devices or pseudodevices. See "open() — Open a file" on page 1152 for more information.

If *fildes* refers to the master side of a pseudoterminal, a SIGHUP signal is sent to the process group, if any, for which the slave side of the pseudoterminal is the controlling terminal.

If *fildes* refers to the slave side of a pseudoterminal, a zero-length message will be sent to the master.

If *fildes* refers to a socket, close() causes the socket to be destroyed. If the socket is connection-oriented and the SO_LINGER option is set for the socket and the socket has untransmitted data, then close() will block for up to the current linger interval until all data is transmitted.

Returned value

If successful, close() returns 0.

If unsuccessful, close() returns -1 and sets errno to one of the following values:

Error Code
  Description

EAGAIN
  The call did not complete because the specified socket descriptor is currently being used by another thread in the same process.

  For example, in a multithreaded environment, close() fails and returns EAGAIN when the following sequence of events occurs (1) thread is blocked in a read() or select() call on a given file or socket descriptor and (2) another thread issues a simultaneous close() call for the same descriptor.

EBADF
fildes is not a valid open file descriptor, or the *socket* parameter is not a valid socket descriptor.

EBUSY
The file cannot be closed because it is blocked.

EINTR
close() was interrupted by a signal. The file may or may not be closed.

EIO **Added for XPG4.2:** An I/O error occurred while reading from or writing to the file system.

ENXIO
fildes does not exist. The minor number for the file is incorrect.

Example

```
#define _POSIX_SOURCE
#include <fcntl.h>
#include <unistd.h>
#include <stdio.h>
#include <string.h>

main() {
  int fd;
  char out[20]="Test string";
  if ((fd = creat("./myfile", S_IRUSR | S_IWUSR)) < 0)
    perror("creat error");
  else {
    if (write(fd, out, strlen(out)+1) == -1)
    perror("write() error");

    if (fd = 0) perror("write() error");
    close(fd);
  }
}
```

Related information

- "unistd.h" on page 82
- "accept() — Accept a new connection on a socket" on page 105
- "creat() — Create a new file or rewrite an existing one" on page 340
- "dup() — Duplicate an open file descriptor" on page 404
- "exec functions" on page 436
- "fclose() — Close file" on page 473
- "fcntl() — Control open file descriptors" on page 474
- "fork() — Create a new process" on page 571
- "getsockopt() — Get the options associated with a socket" on page 777
- "open() — Open a file" on page 1152
- "pipe() — Create an unnamed pipe" on page 1179
- "setsockopt() — Set options associated with a socket" on page 1578
- "socket() — Create a socket" on page 1688
- "shutdown() — Shut down all or part of a duplex connection" on page 1606
- "unlink() — Remove a directory entry" on page 1962

closedir() — Close a directory

Standards

Standards / Extensions	C or C++	Dependencies
POSIX.1 XPG4 XPG4.2 Single UNIX Specification, Version 3	both	

Format

```
#define _POSIX_SOURCE
#include <dirent.h>

int closedir(DIR *dir);
```

General description

Closes the directory indicated by *dir*. It frees the buffer that readdir() uses when reading the directory stream.

Returned value

If successful, closedir() returns 0.

If unsuccessful, closedir() returns -1 and sets errno to one of the following values:

Error Code
Description

EBADF
dir does not refer to an open directory stream.

EINTR
closedir() was interrupted by a signal. The directory may or may not be closed.

Example

CELEBC18

```
/* CELEBC18

   This example closes a directory.

 */
#define _POSIX_SOURCE
#include <dirent.h>
#include <sys/types.h>
#undef _POSIX_SOURCE
#include <stdio.h>

main() {
  DIR *dir;
  struct dirent *entry;
  int count;

  if ((dir = opendir("/")) == NULL)
    perror("opendir() error");
  else {
    count = 0;
    while ((entry = readdir(dir)) != NULL) {
      printf("directory entry %03d: %s\n", ++count, entry->d_name);
    }
    closedir(dir);
  }
}
```

Output

```
directory entry 001: .
directory entry 002: ..
directory entry 003: bin
directory entry 004: dev
directory entry 005: etc
directory entry 006: lib
directory entry 007: tmp
directory entry 008: u
directory entry 009: usr
```

Related information

- “dirent.h” on page 22
- “stdio.h” on page 68
- “sys/types.h” on page 75
- “opendir() — Open a directory” on page 1158
- “readdir() — Read an entry from a directory” on page 1382
- “rewinddir() — Reposition a directory stream to the beginning” on page 1446
- “seekdir() — Set position of directory stream” on page 1469
- “telldir() — Current location of directory stream” on page 1866

closelog() — Close the control log

Standards

Standards / Extensions	C or C++	Dependencies
XPG4.2 Single UNIX Specification, Version 3	both	

Format

```
#define _XOPEN_SOURCE_EXTENDED 1
#include <syslog.h>

void closelog(void);
```

General description

The closelog() function closes the log file.

Returned value

closelog() neither accepts an input nor returns a result. The system control log is closed for this process.

No errors are defined.

Related information

- “syslog.h” on page 72
- “openlog() — Open the system control log” on page 1162
- “setlogmask() — Set the mask for the control log” on page 1561

- "syslog() — Send a message to the control log" on page 1803

clrmemf() — Clear memory files

Standards

Standards / Extensions	C or C++	Dependencies
Language Environment	both	

Format

```
#include <stdio.h>

int clrmemf(int level);
```

General description

Removes memory files created by the current program and any program that was called using a non-POSIX system() call. clrmemf() can remove memory files regardless of whether they are open or not.

To avoid infringing on the user's name space, this nonstandard function has two names. One name is prefixed with two underscore characters, and one name is not. The name without the prefix underscore characters is exposed only when you use LANGLVL(EXTENDED).

To use this function, you must either invoke the function using its external entry point name (that is, the name that begins with two underscore characters), or compile with LANGLVL(EXTENDED). When you use LANGLVL(EXTENDED) any relevant information in the header is also exposed.

The argument *level* indicates which memory files are to be removed. The level can be one of the following:

__LOWER
Removes memory files that were created in other programs and called from this program using system().

__CURRENT
Removes only the memory files created at the current level.

__CURRENT_LOWER
Removes all the memory files created by the current program and by all the programs called at the current level.

Special behavior for multiple shared PICI C environments: Only files created by the C environment from which the clrmemf() function is called will be cleared.

Returned value

If successful, clrmemf() returns 0.

If unsuccessful, clrmemf() returns nonzero.

Example

```
/*
   In this example, when Program2 calls clrmemf()(__CURRENT) only
   A3.FILE and A4.FILE will be removed.
```

```
 */
/*****  Program1  *****/
⋮
   fp1 = fopen ("A1.FILE", "w,type=memory(hiperspace)");
   fp2 = fopen ("A2.FILE", "w,type=memory(hiperspace)");
   system("Program2");
⋮
/*****  Program2  *****/
⋮
   fp3 = fopen("A3.FILE","w,type=memory");
   fp4 = fopen("A4.FILE","w,type=memory");
   system("Program3");
⋮
   clrmemf(__CURRENT);
⋮
/*****  Program3  *****/
⋮
   fp5 = fopen("A5.FILE","w,type=memory");
   fp6 = fopen("A6.FILE","w,type=memory");
⋮
```

Related information

- "stdio.h" on page 68
- "fopen() — Open a file" on page 565
- "system() — Execute a command" on page 1805

__cnvblk() — Convert block

Standards

Standards / Extensions	C or C++	Dependencies
z/OS UNIX	both	

Format

```
#define _XOPEN_SOURCE
#include <unistd.h>

void __cnvblk(char bits[8], char bytes[64], int flag);
```

General description

The __cnvblk() function maps an 8 character array, *bits*, of bits to or from a 64 character array, *bytes*, of bytes depending on the value of *flag*.

If the value of *flag* is 0, __cnvblk() sets all bytes in the *bytes* array to 0x00 for which the corresponding bits in the *bits* array have value 0, and it sets all bytes in the *bytes* array to 0x01 for which the corresponding bits in the *bits* array have value 1.

If the value of *flag* is **not** 0, __cnvblk() sets all bits in the *bits* array to 0 for which corresponding bytes in the *bytes* array have value 0x00, and it sets all bits in the *bits* array to 1 for which the corresponding bytes in the *bytes* array have value 0x01.

This function may be used to prepare input to setkey() or encrypt() functions and to map results back to 8 bit characters.

Returned value

If the value of *flag* is zero, __cnvblk() functions without error checking.

If the value of *flag* is nonzero, __cnvblk() checks for errors, and if found, sets errno to one of the following values:

Error Code
Description

EINVAL
The value of a byte in the array *bytes* is not 0x00 or 0x01.

Note: Because __cnvblk() returns no values, applications wishing to check for errors should set errno to 0, call __cnvblk(), then test errno and, if it is nonzero, assume an error has occurred.

Related information

- "unistd.h" on page 82
- "encrypt() — Encoding function" on page 419
- "setkey() — Set encoding key" on page 1551

collequiv() — Return a list of equivalent collating elements

Standards

Standards / Extensions	C or C++	Dependencies
Language Environment	both	

Format

```
#include <collate.h>

int collequiv(collel_t c, collel_t **list);
```

General description

Finds all the collating elements whose primary weight is the same as the primary weight of c. It then updates the list to point to the first element of the array in which all the found elements are stored. The list of elements is valid until the next call to setlocale(), with categories LC_ALL, LC_COLLATE, or LC_CTYPE.

Another call to collequiv() may override the current list.

For information about the effect of setlocale() and locale.h, see "Internationalization: Locales and Character Sets" in *z/OS XL C/C++ Programming Guide*.

Returned value

If successful, collequiv() returns the number of collating elements found.

If the value of *c* is not in the valid range of collating elements in the current locale, collequiv() returns -1.

Notes:

1. If the collating element passed is specified with the weight of IGNORE in the LC_COLLATE category, the list returned will contain all the characters specified as IGNORE.
2. The list will only contain characters defined in the charmap file in the current locale.

Example

CELEBC22

```
/* CELEBC22

   This example prints the collating elements that have an
   equivalent weight as the collating element passed in
   argv[1].

 */
#include "stdio.h"
#include "locale.h"
#include "collate.h"
#include "stdlib.h"
#include "wctype.h"
#include "wchar.h"

main(int argc, char *argv[]) {
   collel_t e, *rp;
   int i;

   setlocale(LC_ALL, "");
   if ((e = strtocoll(argv[1])) == (collel_t)-1) {
      printf("'%s' collating element not defined\n", argv[1]);
      exit(1);
   }
   if ((i = collequiv(e, &rp)) == -1) {
      printf("Invalid collating element '%s'\n", argv[1]);
      exit(1);
   }
   for (; i-- > 0; rp++) {
      if (ismccollel(*rp))
         printf("'%s' ", colltostr(*rp));
      else if (iswprint(*rp))
         printf("'%lc' ", *rp);
      else
         printf("'%x' ", *rp);
   }
}
```

Related information

- “collate.h” on page 19
- “cclass() — Return characters in a character class” on page 243
- “collorder() — Return list of collating elements” on page 297
- “collrange() — Calculate the range list of collating elements” on page 298
- “colltostr() — Return a string for a collating element” on page 299
- “getmccoll() — Get next collating element from string” on page 734
- “getwmccoll() — Get next collating element from wide string” on page 803
- “ismccollel() — Identify a multicharacter collating element” on page 914
- “maxcoll() — Return maximum collating element” on page 1039
- “strtocoll() — Return collating element for string” on page 1763

collorder() — Return list of collating elements

Standards

Standards / Extensions	C or C++	Dependencies
Language Environment	both	

Format

```
#include <collate.h>

int collorder(collel_t **list);
```

General description

Finds the number of collating elements in the collate order list and sets a pointer to the list. The list returned is valid until another call to setlocale().

Notes:

1. Collating elements specified with the weight of IGNORE in the LC_COLLATE category are defined as having the lowest weight.
2. The list will only contain characters defined in the charmap file in the current locale.

Example

CELEBC23

```
/* CELEBC23

   This example creates a list of all the collating elements using
   the &collo. function.

 */
#include <stdio.h>
#include <locale.h>
#include <collate.h>
#include <wchar.h>
#include <wctype.h>

main(int argc, char *argv[]) {
   collel_t e, *rp;
   int i;

   setlocale(LC_ALL, "TEXAN.IBM-1024");
   i = collorder(&rp);
   for (; i-- > 0; rp++) {
      if (ismccollel(*rp))
         printf("'%s' ", colltostr(*rp));
      else if (iswprint(*rp))
         printf("'%lc' ", *rp);
      else
         printf("'%x' ", *rp);
   }
}
```

Related information

- “collate.h” on page 19
- “cclass() — Return characters in a character class” on page 243
- “collequiv() — Return a list of equivalent collating elements” on page 295
- “collrange() — Calculate the range list of collating elements” on page 298

- “colltostr() — Return a string for a collating element” on page 299
- “getmccoll() — Get next collating element from string” on page 734
- “getwmccoll() — Get next collating element from wide string” on page 803
- “ismccollel() — Identify a multicharacter collating element” on page 914
- “maxcoll() — Return maximum collating element” on page 1039
- “strtocoll() — Return collating element for string” on page 1763

collrange() — Calculate the range list of collating elements

Standards

Standards / Extensions	C or C++	Dependencies
Language Environment	both	

Format

```
#include <collate.h>

int collrange(collel_t start, collel_t end, collel_t **list);
```

General description

Finds a list of collating elements whose primary weights are between the *start* and *end* points, inclusive. The number returned is the number of elements in the list, whose pointer is returned.

This value will be zero if the end point collates earlier than the *start* point. The list returned is valid until the next call to setlocale().

Returned value

If successful, collrange() returns the number of elements in the list, whose pointer is returned.

If either *start* or *end* are out of range, collrange() returns -1.

Notes:
1. Collating elements specified with the weight of IGNORE in the LC_COLLATE category are defined having the lowest weight. Therefore, such elements can only be specified as the starting collating element.
2. The list will only contain characters defined in the charmap file in the current locale.

Example

CELEBC24

```
/* CELEBC24

   This example prints the collating elements in the range
   between the start and end points passed in
   argv[1] and argv[2], using the
   &collrap. function.

 */
#include <stdio.h>
#include <locale.h>
#include <collate.h>
```

```
#include <stdlib.h>
#include <wctype.h>
#include <wchar.h>

main(int argc, char *argv[]) {
   collel_t s, e, *rp;
   int i;

   setlocale(LC_ALL, "TEXAN.IBM-1024");
   if ((s = strtocoll(argv[1])) == (collel_t)-1) {
      printf("'%s' collating element not defined\n", argv[1]);
      exit(1);
   }
   if ((e = strtocoll(argv[2])) == (collel_t)-1) {
      printf("'%s' collating element not defined\n", argv[2]);
      exit(1);
   }
   if ((i = collrange(s, e, &rp)) == -1) {
      printf("Invalid range for '%s' to '%s'\n", argv[1], argv[2]);
      exit(1);
   }
   for (; i-- > 0; rp++) {
      if (ismccollel(*rp))
         printf("'%s' ", colltostr(*rp));
      else if (iswprint(*rp))
         printf("'%lc' ", *rp);
      else
         printf("'%x' ", *rp);
   }
}
```

Related information

- “collate.h” on page 19
- “cclass() — Return characters in a character class” on page 243
- “collequiv() — Return a list of equivalent collating elements” on page 295
- “collorder() — Return list of collating elements” on page 297
- “colltostr() — Return a string for a collating element”
- “getmccoll() — Get next collating element from string” on page 734
- “getwmccoll() — Get next collating element from wide string” on page 803
- “ismccollel() — Identify a multicharacter collating element” on page 914
- “maxcoll() — Return maximum collating element” on page 1039
- “strtocoll() — Return collating element for string” on page 1763

colltostr() — Return a string for a collating element

Standards

Standards / Extensions	C or C++	Dependencies
Language Environment	both	

Format

```
#include <collate.h>

char *colltostr(collel_t c);
```

General description

Converts `c` to the string of the collating element. The colltostr() function is the inverse of strtocoll().

An application program can use the returned array from collrange() or collequiv(), calling ismccollel() on each element, only calling colltostr() if ismccollel() is true for the element. The string returned is valid until another call to setlocale().

Returned value

If a value is passed representing a single character or a value that is not in range, colltostr() returns NULL.

Example

CELEBC25

```
/* CELEBC25

   This example prints all the collating elements in the
   collating sequence, using the &colltop. function to get the
   string for the multi-character collating elements.

 */

#include <collate.h>
#include <locale.h>
#include <stdio.h>
#include <wchar.h>
#include <wctype.h>

main(int argc, char *argv[]) {
   collel_t e, *rp;
   int i;

   setlocale(LC_ALL, "");
   i = collorder(&rp);
   for (; i-- > 0; rp++) {
      if (ismccollel(*rp))
         printf("'%s' ", colltostr(*rp));
      else if (iswprint(*rp))
         printf("'%lc' ", *rp);
      else
         printf("'%x' ", *rp);
   }
}
```

Related information

- “collate.h” on page 19
- “cclass() — Return characters in a character class” on page 243
- “collequiv() — Return a list of equivalent collating elements” on page 295
- “collorder() — Return list of collating elements” on page 297
- “collrange() — Calculate the range list of collating elements” on page 298
- “getmccoll() — Get next collating element from string” on page 734
- “getwmccoll() — Get next collating element from wide string” on page 803
- “ismccollel() — Identify a multicharacter collating element” on page 914
- “maxcoll() — Return maximum collating element” on page 1039
- “strtocoll() — Return collating element for string” on page 1763

compile() — Compile regular expression

Standards

Standards / Extensions	C or C++	Dependencies
XPG4 XPG4.2	both	

Format

```
#define INIT declarations
#define GETC() getc_code
#define PEEK() peek_code
#define UNGETC() ungetc_code
#define RETURN(ptr) return_code
#define ERROR(val) error_code

#define _XOPEN_SOURCE
#include <regexp.h>

char *compile(char *instring, char *expbuf, const char *endbuf, int eof);
```

General description

Restriction: This function is not supported in AMODE 64.

The compile() function takes as input a simple regular expression and produces a compiled expression that can be used with the step() and advance() functions.

The first parameter *instring* is never used explicitly by compile(). It is a pointer to a character string defining a source regular expression. It is useful for programs that pass down different pointers to input characters. Programs which invoke functions to input characters or have characters in an external array can pass down **(char *)0** for this parameter.

expbuf is a pointer to the place where the compiled regular expression will be placed.

endbuf points to one more than the highest address where the compiled regular expression may be placed. If the compiled expression cannot fit in (*endbuf*-*expbuf*) bytes, a call to ERROR(50) is made. (See "Returned Value" below.)

eof is the character which marks the end of the regular expression.

The z/OS UNIX services implementation of the compile() function does **not** accept internationalized simple expressions as input. Internationalized simple expressions (for example, [[=*c*=]] (an equivalence class)) may yield unpredictable results.

Programs must have the following five macros declared before the **#include <regexp.h>** statement. The macros GETC(), PEEKC() and UNGETC() operate on the regular expression given as input to compile().

GETC()
: This macro returns the value of the next character (byte) in the regular expression pattern. Successive calls to GETC() should return successive characters of the regular expression.

PEEK()
This macro returns the next character (byte) in the regular expression pattern. Immediate successive calls to PEEK() should return the same byte, which should also be the next character returned by GETC().

UNGETC(*c*)
This macro causes the argument *c* to be returned by the next call to GETC(). No more than one character is ever needed and this character is guaranteed to be the last character read by GETC(). The value of the macro UNGETC() is always ignored.

RETURN(*ptr*)
This macro is used on normal exit of the compile() function. The value of the argument *ptr* is a pointer to the character after the last character of the compiled regular expression.

ERROR(*val*)
This macro is the abnormal return from compile(). The argument *val* is an error number. (See "Returned Value" below for meanings.) This call should never return.

Notes:

1. z/OS UNIX services do not provide any default macros if the above user macros are not provided.
2. Each program that includes the **<regexp.h>** must have a **#define** statement for INIT. It is used for dependent declarations and initializations. For example, it can be used to set a variable to point to the beginning of the regular expression so that this variable can be used in the declarations for GETC(), PEEK(), and UNGETC().
3. The external variables *cirf*, *sed*, and *nbra* are reserved.
4. The application must provide the proper serialization for the compile(), step(), and advance() functions if they are run under a multithreaded environment.

Simple regular expressions

A Simple Regular Expression (SRE) specifies a set of character strings. The simplest form of regular expression is a string of characters with no special meaning. A small set of special characters, known as metacharacters, do have special meaning when encountered in patterns.

Expression
Meaning

c
The character *c* where *c* is not a special character.

\c
The character *c* where *c* is any special character. For example, `a\.e` is equivalent to `a.e`.

^
The beginning of the string being compared

$
The dollar symbol matches the end of the string.

.
The period symbol matches any one character.

[*string*]
A string within square brackets specifies any of the characters in *string*. Thus, `[abc]`, if compared to other strings, would match any which contained a, b, or c.

The] (right bracket) can be used alone within a pair of brackets, but only if it immediately follows either the opening left bracket or if it immediately follows [^.

Ranges may be specified as *c*–*c*. The hyphen symbol, within square brackets, means "through". It fills in the intervening characters according to the collating sequence. For example, [a–z] is equivalent to [abc...xyz]. If the end character in the range is lower in collating sequence to the start character, then only the range start and range end characters are accepted in the search pattern. For example, [9–1] is equivalent to [91]. Note that ranges in Simple Regular Expressions are only valid if the LC_COLLATE category is set to the C locale.

The – (hyphen) can be used by itself, but only if it is the first or last character in the expression. For example, the expression []*a-f*] matches either the] or one of the characters `a` through *f*.

[^*string*]
The caret symbol, when inside square brackets, negates the characters within the square brackets. Thus, `[^abc]`, if compared to other strings, would *fail* to match any which contains even one a, b, or c.

Note: Characters **.**, *, [, and \ (period, asterisk, left square bracket, and backslash, respectively) have special meaning, except when they appear within square brackets ([]), or are preceded by \.

* The asterisk symbol indicates 0 or more of any preceding characters. For example, (`a*e`) will match any of the following: `e, ae, aae, aaae, ....` The longest leftmost match is chosen.

rx The occurrence of regular expression *r* followed by the occurrence of regular expression *x*.

\{*m*\} \{*m*,\} \{*m*,*u*\}
Integer values enclosed in \{\} indicate the number of times to apply the preceding regular expression. *m* is the minimum number and *u* is the maximum number. *u* must be less than 256. If you specify only *m*, it indicates the exact number of times to apply the regular expression.

\{*m*,\} is equivalent to \{*m*,*255*\}. They both match *m* or more occurrences of the expression. The * (asterisk) operation is equivalent to \{0,\}.

The maximum number of occurrences is matched.

\(*r***\)** The regular expression *r*. The \(and \) sequences are ignored.

\n When \n (where 1 <= *n* <= 9) appears in a concatenated regular expression, it stands for the regular expression *x*, where *x* is the *n*th regular expression enclosed in \(and \) sequences that appeared earlier in the concatenated regular expression. For example, in the pattern \\(*c*\\)*onc*\\(*ate*\\)*n*\2, the \2 is equivalent to *ate*, giving *concatenate*.

The character ^ at the beginning of an expression permits a successful match only immediately after a newline or at the beginning of each of the string to which a match is to be applied. The character $ at the end of an expression requires a trailing newline.

Notes:

1. The compile() function is physically embedded in the **regexp.h** header. This header will be protected from multiple invocations just like other c headers.

2. The compile(), step(), and advance() functions are provided for historical reasons. These functions were part of the Legacy Feature in Single UNIX Specification, Version 2. They have been withdrawn and are not supported as part of Single UNIX Specification, Version 3. New applications should use the newer functions fnmatch(), glob(), regcomp() and regexec(), which provide full internationalized regular expression functionality compatible with IEEE Std 1003.1-2001.

Returned value

If successful, compile() exits using the user-provided macro RETURN(*ptr*). The value of the argument *ptr* is a pointer to the character after the last character of the compiled regular expression.

If unsuccessful, compile() exits using the user-provided macro ERROR(*val*). The argument *val* is an error number identifying the error. The following error numbers are defined:

Errcode	**Description String**
11	Range endpoint too large
16	Bad number
25	\digit out of range
36	Illegal or missing delimiter
41	No remembered search string
42	\\(\\) imbalance
43	Too many \\(
44	More than two numbers given in \\{ \\}
45	} expected after \\
46	First number exceeds second in \\{ \\}
49	[ ] imbalance
50	Regular expression overflow

Related information

- “regexp.h” on page 61
- “advance() — Pattern match given a compiled regular expression” on page 141
- “fnmatch() — Match file name or path name” on page 564
- “glob() — Generate path names matching a pattern” on page 807
- “regcomp() — Compile regular expression” on page 1415
- “regexec() — Execute compiled regular expression” on page 1421
- “step() — Pattern match with regular expression” on page 1726

confstr() — Get configurable variables

Standards

Standards / Extensions	C or C++	Dependencies
XPG4 XPG4.2 Single UNIX Specification, Version 3	both	

Format

```
#define _XOPEN_SOURCE
#include <unistd.h>

size_t confstr(int name, char *buf, size_t len);
```

General description

The confstr() function provides a method for applications to get configuration-defined string values. Its use and purpose are similar to the sysconf() function, but it is used where string values rather than numeric values are returned.

The *name* argument represents the system variable to be queried. It may be any one of the following symbolic constants, defined in <unistd.h>:

_CS_PATH
Request the value of the **PATH** environment variable that will find all standard utilities.

_CS_POSIX_V6_ILP32_OFF32_CFLAGS
Request the value of the set of initial options to be given to the c99 utility to build an application using a programming model with 32-bit types.

_CS_POSIX_V6_ILP32_OFF32_LDFLAGS
Request the value of the set of final options to be given to the c99 utility to build an application using a programming model with 32-bit types.

_CS_POSIX_V6_ILP32_OFF32_LIBS
Request the value of the set of libraries to be given to the c99 utility to build an application using a programming model with 32-bit int, long, pointer, and off_t types.

_CS_POSIX_V6_ILP32_OFFBIG_CFLAGS
Request the value of the set of initial options to be given to the c99 utility to build an application using a programming model with 32-bit int, long, and pointer types, and an off_t type using at least 64 bits.

_CS_POSIX_V6_ILP32_OFFBIG_LDFLAGS
Request the value of the set of final options to be given to the c99 utility to build an application using a programming model with 32-bit int, long, and pointer types, and an off_t type using at least 64 bits.

CS_POSIX_V6_ILP32_OFFBIG_LIBS
Request the value of the set of libraries to be given to the c99 utility to build an application using a programming model with 32-bit int, long, and pointer types, and an off_t type using at least 64 bits.

_CS_POSIX_V6_LP64_OFF64_CFLAGS
Request the value of the set of initial options to be given to the c99 utility to build an application using a programming model with 32-bit int and 64-bit long, pointer, and off_t types.

_CS_POSIX_V6_LP64_OFF64_LDFLAGS
Request the value of the set of final options to be given to the c99 utility to build an application using a programming model with 32-bit int and 64-bit long, pointer, and off_t types.

_CS_POSIX_V6_LP64_OFF64_LIBS
Request the value of the set of libraries to be given to the c99 utility to build an application using a programming model with 32-bit int and 64-bit long, pointer, and off_t types.

_CS_POSIX_V6_LPBIG_OFFBIG_CFLAGS
Request the value of the set of initial options to be given to the c99 utility to build an application using a programming model with an int type using at least 32 bits and long, pointer, and off_t types using at least 64 bits.

_CS_POSIX_V6_LPBIG_OFFBIG_LDFLAGS
Request the value of the set of final options to be given to the c99 utility to build an application using a programming model with an int type using at least 32 bits and long, pointer, and off_t types using at least 64 bits.

_CS_POSIX_V6_LPBIG_OFFBIG_LIBS
Request the value of the set of libraries to be given to the c99 utility to build an application using a programming model with an int type using at least 32 bits and long, pointer, and off_t types using at least 64 bits.

_CS_POSIX_V6_WIDTH_RESTRICTED_ENVS
Request the list of names of programming environments supported by the implementation in which the widths of the blksize_t, cc_t, mode_t, nfds_t, pid_t, ptrdiff_t, size_t, speed_t, ssize_t, suseconds_t, tcflag_t, useconds_t, wchar_t, and wint_t types are no greater than the width of type long.

z/OS UNIX services use the following constant:

_CS_SHELL
Request the fully qualified name of the default shell.

If the *len* argument is not zero, and if the *name* argument has a configuration-defined value, confstr() copies that value into the buffer pointed to by the *buf* argument. If the value to be returned is longer than *len* bytes, including the . terminating NULL, then confstr() truncates the string to *len*-1 bytes and NULL-terminates the results. The application can detect that the string was truncated by comparing the value returned by confstr() with *len*.

If the *len* argument is zero, and the *buf* argument is a NULL pointer, then confstr() still returns the integer value defined below, but does not return a string. If the *len* argument is zero, but the *buf* argument is not a NULL pointer the results are unspecified.

Returned value

If *name* has a configuration-defined value, confstr() returns the size of the buffer that would be needed to hold the entire configuration-defined string value. If this return value is greater than *len*, the string returned in *buf* is truncated.

If *name* does not have a configuration-defined value, confstr() returns 0 and leaves errno unchanged.

If *name* is not valid, confstr() returns 0 and sets errno to one of the following values:

Error Code
Description

EINVAL
The value of the *name* argument is not valid.

Related information

- “unistd.h” on page 82
- “fpathconf() — Determine configurable path name variables” on page 581
- “pathconf() — Determine configurable path name variables” on page 1169
- “sysconf() — Determine system configuration options” on page 1798

conj(), conjf(), conjl() — Calculate the complex conjugate

Standards

Standards / Extensions	C or C++	Dependencies
C99 Single UNIX Specification, Version 3	both	z/OS V1R7 a compiler that is designed to support C99

Format

```
#include <complex.h>

double complex conj(double complex z);
float complex conjf(float complex z);
long double complex conjl(long double complex z);
```

General description

The conj() family of functions compute the complex conjugate of *z* by reversing the sign of its imaginary part.

Note: The following table shows the viable formats for these functions. See “IEEE binary floating-point” on page 94 for more information about IEEE Binary Floating-Point.

Function	Hex	IEEE
conj	X	X
conjf	X	X
conjl	X	X

Returned value

If successful, they return the complex conjugate value.

Example

```
/*
 *  This example illustrates the complex conjugate function
 */
#include <complex.h>
#include <stdio.h>

void main()
{
   long double complex z = -2.99 - I*3.99, zres;
   zres = conjl(z);
   printf("The complex conjugate of %Lf + %Lf*I is %Lf + %Lf*I\n",creall(z), cimagl(z), zres);
}

Output
The complex conjugate of -2.990000 + -3.990000*I is -2.990000 + 3.990000*I
```

Related information

- “complex.h” on page 19
- “carg(), cargf(), cargl() — Calculate the argument” on page 233
- “cimag(), cimagf(), cimagl() — Calculate the complex imaginary part” on page 280
- “cproj(), cprojf(), cprojl() — Calculate the projection” on page 338
- “creal(), crealf(), creall() — Calculate the complex real part” on page 340

connect() — Connect a socket

Standards

Standards / Extensions	C or C++	Dependencies
XPG4.2 Single UNIX Specification, Version 3	both	

Format

X/Open:

```
#define _XOPEN_SOURCE_EXTENDED 1
#include <sys/socket.h>

int connect(int socket, const struct sockaddr *address, socklen_t address_len);
```

Berkeley sockets:

```
#define _OE_SOCKETS
#include <sys/types.h>
#include <sys/socket.h>

int connect(int socket, struct sockaddr *address, int address_len);
```

General description

For stream sockets, the connect() call attempts to establish a connection between two sockets. For datagram sockets, the connect() call specifies the peer for a socket. The *socket* parameter is the socket used to originate the connection request. The connect() call performs two tasks when called for a stream socket. First, it completes the binding necessary for a stream socket (in case it has not been previously bound using the bind() call). Second, it attempts to make a connection to another socket.

Note: For the X/Open socket function, the *socket* description applies to *socket*, *address* to *address*, and *address_len* to *address_len*. **const** is added to **struct sockaddr**.

Parameter
Description

socket The socket descriptor.

address The pointer to a socket address structure containing the address of the socket to which a connection will be attempted.

address_len
The size of the socket address pointed to by *address* in bytes.

The connect() call on a stream socket is used by the client application to establish a connection to a server. The server must have a passive open pending. A server that is using sockets must successfully call bind() and listen() before a connection can be accepted by the server with accept(). Otherwise, connect() returns -1 and the error code is set to ECONNREFUSED.

If *socket* is in blocking mode, the connect() call blocks the caller until the connection is set up, or until an error is received. If *socket* is in non-blocking mode, the connect returns immediately with a return code of -1 and an errno of EINPROGRESS. The caller can test the completion of the connection setup by calling select() and testing for the ability to write to the socket.

When called for a datagram socket, connect() specifies the peer with which this socket is associated. This gives the application the ability to use data transfer calls reserved for sockets that are in the connected state. In this case, read(), write(), readv(), writev(), send(), and recv() calls are then available in addition to sendto(), recvfrom(), sendmsg(), and recvmsg() calls. Stream sockets can call connect() only once, but datagram sockets can call connect() multiple times to change their association. Datagram sockets can dissolve their association by connecting to an incorrect address, such as the NULL address (all fields zeroed).

The *address* parameter is a pointer to a buffer containing the name of the peer to which the application needs to connect. The *address_len* parameter is the size, in bytes, of the buffer pointed to by *address*.

Servers in the AF_INET domain: If the server is in the AF_INET domain, the format of the name buffer is expected to be **sockaddr_in**, as defined in the include file **netinet/in.h**.

```
struct in_addr
{
        ip_addr_t s_addr;
};

 struct sockaddr_in {
     unsigned char  sin_len;
     unsigned char  sin_family;
     unsigned short sin_port;
     struct in_addr sin_addr;
     unsigned char  sin_zero[8];

};
```

The *sin_family* field must be set to AF_INET. The *sin_port* field is set to the port to which the server is bound. It must be specified in network byte order. The *sin_zero* field is not used and must be set to all zeros.

Servers in the AF_INET6 domain: If the server is in the AF_INET6 domain, the format of the name buffer is expected to be **sockaddr_in6**, as defined in the **netinet/in.h**:

```
 struct sockaddr_in6 {
     uint8_t char    sin6_len;
     sa_family_t     sin6_family;
     in_port_t       sin6_port;
     uint32_t        sin6_flowinfo;
     struct in6_addr sin6_addr;
     uint32_t        sin6_scope_id;
];
```

The *sin6_family* must be set to AF_INET6.

Servers in the AF_UNIX domain: If the server is in the AF_UNIX domain, the format of the name buffer is expected to be **sockaddr_un**, as defined in the include file **un.h**.

```
 struct sockaddr_un {
    unsigned char  sun_len;
    unsigned char  sun_family;
             char  sun_path[108];       /* path name */
};
```

The *sun_family* field is set to AF_UNIX. The *sun_path* field contains the NULL-terminated path name, and *sun_len* contains the length of the path name.

Special behavior for C++: To use this function with C++, you must use the _XOPEN_SOURCE_EXTENDED 1 feature test macro.

Note: The connect() function has a dependency on the level of the Enhanced ASCII Extensions. See "Enhanced ASCII support" on page 2113 for details.

Returned value

If successful, connect() returns 0.

If unsuccessful, connect() returns -1 and sets errno to one of the following values:

Error Code
Description

EADDRNOTAVAIL
The specified address is not available from the local machine.

EAFNOSUPPORT
The address family is not supported.

EALREADY
The socket descriptor *socket* is marked nonblocking, and a previous connection attempt has not completed.

EBADF
The *socket* parameter is not a valid socket descriptor.

ECONNREFUSED
The connection request was rejected by the destination host.

EFAULT
Using *address* and *address_len* would result in an attempt to copy the address into a portion of the caller's address space to which data cannot be written.

EINTR
The attempt to establish a connection was interrupted by delivery of a signal that was caught. The connection will be established asynchronously.

EINVAL
The *address_len* parameter is not a valid length.

EIO There has been a network or a transport failure.

EISCONN
The socket descriptor *socket* is already connected.

ENETUNREACH
The network cannot be reached from this host.

ENOTSOCK
The descriptor refers to a file, not a socket.

EOPNOTSUPP
The *socket* parameter is not of type SOCK_STREAM.

EPERM
connect() caller was attempting to extract a user's identity and the caller's process was not verified to be a server. To be server-verified, the caller's process must have permission to the BPX.SERVER profile (or superuser

and BPX.SERVER is undefined) and have called either the __passwd() or pthread_security_np() services before calling connect() to propagate identity.

EPROTOTYPE
The protocol is the wrong type for this socket.

ETIMEDOUT
The connection establishment timed out before a connection was made.

The following are for AF_UNIX only:

Error Code
Description

EACCES
Search permission is denied for a component of the path prefix, or write access to the named socket is denied.

EIO An I/O error occurred while reading from or writing to the file system.

ELOOP
Too many symbolic links were encountered in translating the path name in *address*.

ENAMETOOLONG
A component of a path name exceeded **NAME_MAX** characters, or an entire path name exceeded **PATH_MAX** characters.

ENOENT
A component of the path name does not name an existing file or the path name is an empty string.

ENOTDIR
A component of the path prefix of the path name in *address* is not a directory.

Example

The following are examples of the connect() call. The Internet address and port must be in network byte order. To put the port into network byte order, the htons() utility routine is called to convert a short integer from host byte order to network byte order. The *address* field is set using another utility routine, inet_addr(), which takes a character string representing the dotted-decimal address of an interface and returns the binary Internet address representation in network byte order. Finally, it is a good idea to zero the structure before using it to ensure that the name requested does not set any reserved fields. These examples could be used to connect to the servers shown in the examples listed with the call, “bind() — Bind a name to a socket” on page 210.

```
int s;
struct sockaddr_in inet_server;
struct sockaddr_un unix_server;
int rc;
int connect(int s, struct sockaddr *name, int namelen);

/* Connect to server bound to a specific interface in the
Internet domain */
/* make sure the sin_zero field is cleared */
memset(&inet_server, 0, sizeof(inet_server));
inet_server.sin_family = AF_INET;
inet_server.sin_addr.s_addr = inet_addr("129.5.24.1");
/* specific interface */
```

```
inet_server.sin_port = htons(1024);
⋮
rc = connect(s, (struct sockaddr *) &inet_server,
sizeof(inet_server));

/* Connect to a server bound to a name in the UNIX domain */
/* make sure the sunix_addr, sunix_port, sunix_nodeid fields are cleared
*/
memset(&unix_server, 0, sizeof(unix_server));
unix_server.sun_family = AF_UNIX;
strncpy(unix_server.sun_path, "mysocket");
unix_server.sun_len = sizeof(unix_server.sun_len);
strncpy(mvsservername.sunix_name, "APPL    ", 8);
⋮
rc = connect(s, (struct sockaddr *)
&unix_server, sizeof(unix_server));
```

Related information

- “sys/socket.h” on page 74
- “accept() — Accept a new connection on a socket” on page 105
- “bind() — Bind a name to a socket” on page 210
- “htons() — Translate an unsigned short integer into network byte order” on page 820
- “inet_addr() — Translate an Internet address into network byte order” on page 856
- “listen() — Prepare the server for incoming client requests” on page 972
- “select(), pselect() — Monitor activity on files or sockets and message queues” on page 1470
- “selectex() — Monitor activity on files or sockets and message queues” on page 1479
- “socket() — Create a socket” on page 1688

ConnectExportImport() — WLM connect for export or import use

Standards

Standards / Extensions	C or C++	Dependencies
z/OS UNIX	both	OS/390 V2R9

Format

```
#include <sys/__wlm.h>

unsigned long ConnextExportImport(const char *subsystype,
                                  const char *subsysname);
```

AMODE 64:

```
#include <sys/__wlm.h>

unsigned int ConnextExportImport(const char *subsystype,
                                 const char *subsysname);
```

General description

Provides the ability for an application to connect to WLM to use the ExportWorkUnit(), UndoExportWorkUnit(), ImportWorkUnit(), and UndoImportWorkUnit() functions.

Note that if you need to use the CreateWorkUnit() function, you should use ConnectWorkMgr() instead.

The ConnectExportImport() function uses the following parameters:

**subsystype*
: Points to a NULL-terminated character string containing the subsystem type by which to identify the connector. The export and import functions do not use the string. A meaningful string should be used since it can appear in WLMDATA IPCS reports. The character string can be up to 4 bytes in length.

**subsysname*
: Points to a NULL-terminated character string containing the subsystem name by which to identify the connector. The export and import functions do not use the string. A meaningful string should be used since it can appear in WLMDATA IPCS reports. The character string can be up to 8 bytes in length.

Returned value

If successful, ConnectExportImport() returns 0.

If unsuccessful, ConnectExportImport() returns -1 and sets errno to one of the following values:

Error Code
: **Description**

EFAULT
: An argument of this function contained an address that was not accessible to the caller.

EINVAL
: An argument of this function contained a value that is not correct.

EMVSWLMERROR
: A WLM service failed. Use __errno2() to obtain the WLM service reason code for the failure.

Related information

- "sys/__wlm.h" on page 77
- "ExportWorkUnit() — WLM export service" on page 452
- "ImportWorkUnit() — WLM import service" on page 840
- "QueryWorkUnitClassification() — WLM query enclave classification service" on page 1370
- "UnDoExportWorkUnit() — WLM undo export service" on page 1955
- "UnDoImportWorkUnit() — WLM undo import service" on page 1956
- For more information, see *z/OS MVS Programming: Workload Management Services, SC34-2663*.

ConnectServer() — Connect to WLM as a server manager

Standards

Standards / Extensions	C or C++	Dependencies
z/OS UNIX	both	

Format

```
#include <sys/__wlm.h>

unsigned long ConnectServer(const char *subsystype,
                            const char *subsysname,
                            const char *applenv,
                            int *paralleleu);
```

AMODE 64:

```
#include <sys/__wlm.h>

unsigned int ConnectServer(const char *subsystype,
                           const char *subsysname,
                           const char *applenv,
                           int *paralleleu);
```

General description

The ConnectServer function provides the ability for an application to connect to WLM as a WLM server manager to perform WLM server manager functions.

**subsystype*
: Points to a NULL-terminated character string containing the generic subsystem type (CICS, IMS, WEB, etc.). This is the primary category under which WLM classification rules are grouped. The character string can be up to 4 bytes in length.

**subsysname*
: Points to a NULL-terminated character string containing the subsystem name used for classifying work requests. The character string can be up to 8 bytes in length.

**applenv*
: Points to a NULL-terminated character string that contains the name of the application environment under which work requests are processed. The character string can be up to 32 bytes in length.

**paralleleu*
: Points to an integer which contains the maximum number of tasks within the address space which will be created to process concurrent work requests.

Returned value

If successful, ConnectServer() returns a nonzero value representing a WLM connect token.

If unsuccessful, ConnectServer() returns -1 and sets errno to one of the following values:

Error Code
: **Description**

EFAULT
: An argument of this function contained an address that was not accessible to the caller.

EINVAL
: An argument of this function contained an incorrect value.

EMVSSAF2ERR
An error occurred in the security product.

EMVSWLMERROR
The WLM connect failed. Use __errno2() to obtain the WLM service reason code for the failure.

EPERM
The calling thread's address space is not permitted to the BPX.WLMSERVER Facility class. The caller's address space must be permitted to the BPX.WLMSERVER Facility class if it is defined. If BPX.WLMSERVER is not defined, the calling process is not defined as a superuser (UID=0).

Related information

- "sys/__wlm.h" on page 77
- "CheckSchEnv() — Check WLM scheduling environment" on page 271
- "ConnectWorkMgr() — Connect to WLM as a work manager"
- "ContinueWorkUnit() — Continue WLM work unit" on page 323
- "CreateWorkUnit() — Create WLM work unit" on page 344
- "DeleteWorkUnit() — Delete a WLM work unit" on page 378
- "DisconnectServer() — Disconnect from WLM server" on page 383
- "JoinWorkUnit() — Join a WLM work unit" on page 927
- "LeaveWorkUnit() — Leave a WLM work unit" on page 945
- "QueryMetrics() — Query WLM system information" on page 1368
- "QuerySchEnv() — Query WLM scheduling environment" on page 1369

ConnectWorkMgr() — Connect to WLM as a work manager

Standards

Standards / Extensions	C or C++	Dependencies
z/OS UNIX	both	

Format

```
#include <sys/__wlm.h>

unsigned long ConnectWorkMgr(const char *subsystype,
                              const char *subsysname);
```

AMODE 64:

```
#include <sys/__wlm.h>

unsigned int ConnectWorkMgr(const char *subsystype,
                             const char *subsysname);
```

General description

The ConnectServer function provides the ability for an application to connect to WLM as a WLM work manager to perform WLM work manager functions.

**subsystype*
Points to a NULL-terminated character string containing the generic subsystem type (CICS, IMS, WEB, etc.). This is the primary category under which WLM classification rules are grouped. The character string can be up to 4 bytes in length.

**subsysname*
Points to a NULL-terminated character string containing the subsystem name used for classifying work requests. The character string can be up to 8 bytes in length.

Returned value

If successful, ConnectServer() returns a nonzero value representing a WLM connect token.

If unsuccessful, ConnectServer() returns -1 and sets errno to one of the following values:

Error Code
Description

EFAULT
An argument of this function contained an address that was not accessible to the caller.

EINVAL
An argument of this function contained an incorrect value.

EMVSSAF2ERR
An error occurred in the security product.

EMVSWLMERROR
The WLM connect failed. Use __errno2() to obtain the WLM service reason code for the failure.

EPERM
The calling thread's address space is not permitted to the BPX.WLMSERVER Facility class. The caller's address space must be permitted to the BPX.WLMSERVER Facility class if it is defined. If BPX.WLMSERVER is not defined, the calling process is not defined as a superuser (UID=0).

Related information

- "sys/__wlm.h" on page 77
- "CheckSchEnv() — Check WLM scheduling environment" on page 271
- "ConnectServer() — Connect to WLM as a server manager" on page 313
- "ContinueWorkUnit() — Continue WLM work unit" on page 323
- "CreateWorkUnit() — Create WLM work unit" on page 344
- "DeleteWorkUnit() — Delete a WLM work unit" on page 378
- "DisconnectServer() — Disconnect from WLM server" on page 383
- "JoinWorkUnit() — Join a WLM work unit" on page 927
- "LeaveWorkUnit() — Leave a WLM work unit" on page 945
- "QueryMetrics() — Query WLM system information" on page 1368
- "QuerySchEnv() — Query WLM scheduling environment" on page 1369

__console() — Console communication services

Standards

Standards / Extensions	C or C++	Dependencies
z/OS UNIX	both	

Format

```
#include <sys/__messag.h>

int __console(struct __cons_msg *cons, char *modstr, int *concmd);
```

General description

The __console() function is used to communicate with the operator's console. The __console() function allows users to send messages to the operator's console and to wait on a modify/stop request from the console.

The parameters supported are:

cons
If the argument is not NULL, it points to a structure specifying the message that is to be sent to the operator's console. If the argument is NULL then no message is sent.

modstr
Specifies the string where __console() returns the data entered at the operator's console. If modstr is not NULL the invoker will wait until an operator MODIFY's the invoking job and specifies 'APPL=' parameter. The length of modstr should be 128-byte. See *z/OS MVS System Commands* for more information on the MODIFY console command. If the argument is NULL then the __console() function will not wait on operator console commands.

concmd
If a console command was issued against the invoking job, the __console() function will set the command type. Valid types are, _CC_modify (function received a modify request) and _CC_stop (function received a stop request).

The cons structure is defined in the <sys/__messag.h> header and has the following format.

```
    struct __cons_msg   {
      short __reserved0;
      char  __reserved1[2];
      union   {
         struct   {
            int  __msg_length;
            char *__msg;
            char __reserved2[8];
         } __f1;
      }  __format;
    };
```

__reserved0
Reserved for future use.

__reserved1[2]
Reserved for future use.

__format.__f1.__msg_length
Length of message, not including the NULL terminator.

__format.__f1.__msg
A character string containing the message to be sent to the operator console.

__format.__f1.__reserved2[8]
Reserved for future use.

Note: The length of the message must be between 1 and 17850 characters for invokers with appropriate privileges, and between 1 and 17780 for invokers without appropriate privileges. The number of lines written to the console is limited to 255. In the case of an unprivileged user, one of those lines is used for the message ID and the invoker's login name. If the message length is exceeded, no lines are written and the service returns an EINVAL. If the number of lines is exceeded, the service returns an EINVAL, but the first 255 lines are written to the console.

Returned value

If successful, __console() returns 0.

If unsuccessful, __console() returns -1 and sets errno to one of the following values:

Error Code
: **Description**

EFAULT
: One of the following errors was detected:
- All or part of the cons structure is not addressable by the caller.
- All or part of the modstr string is not addressable by the caller.

EINTR
: __console() was interrupted by a signal.

EINVAL
: The *cons* structure contains errors.

EMVSERR
: z/OS environmental or internal error has occurred.

Example

CELEBC41

```
/* CELEBC41

   This example prints a simple message to the console using the
   __console() function.

 */
#include <sys/__messag.h>
#include <errno.h>
#include <string.h>
#include <stdio.h>

int main(int argc, char** argv) {
   struct __cons_msg cmsg;
   char buf[256] = "A message on the console";
   int rc;
   int cmsg_cmd = 0;

   /* fill in the __cons_msg structure */
   cmsg.__format.__f1.__msg = buf;
   cmsg.__format.__f1.__msg_length = strlen(buf);

   rc = __console(&cmsg,NULL,&cmsg_cmd);
   if(rc == -1) {
      printf("__console() failed\n");
      printf("%s\n",strerror(errno));
   }
   else {
      printf("__console() successful. Check console for message.\n");
   }

   return 0;
}
```

Related information

- “sys/__messag.h” on page 73
- “__console2() — Enhanced console communication services”

__console2() — Enhanced console communication services

Standards

Standards / Extensions	C or C++	Dependencies
z/OS UNIX	both	OS/390 V2R10

Format

```
#include <sys/__messag.h>

int __console2(struct __cons_msg2 *cons, char *modstr, int *concmd);
```

General description

The __console2() function is used to communicate with the operator console. The __console2() function allows users to send messages to the operator console with the ability to specify routing codes and message descriptor codes, wait on a modify or stop request from the console, and to delete messages from operator console using either a message ID or token.

The __console2() function parameters are as follows:

cons
Specifies the address of the structure containing the console communication information. The mapping of the structure is provided below. If this parameter is NULL, then no message is sent to the operator console and no messages are deleted from the console.

modstr
Address of a 128-byte buffer that is to be used to receive a string of EBCDIC data from the console MODIFY command. All characters that appear to the right of the “APPL=” are placed into this buffer, left justified. The data returned is folded to uppercase. If this parameter is NULL, then the __console2() function does not wait on operator console commands.

concmd
Address of a 32-bit integer where the __console2() function returns the type of command that was issued on the console. If this parameter is set to NULL, the __console2() function will fail with EFAULT. The command types are:

_CC_modify
Function received a modify request.

_CC_stop
Function received a stop request.

The console communication information is specified in a structure pointed to by the *cons* parameter. The structure contains the following fields:

__cm2_format

Specifies the format of the structure. This field must be set to one of the following:

__CONSOLE_FORMAT_2

Used to indicate structure format 2.

__CONSOLE_FORMAT_3

Used to indicate structure format 3.

__cm2_msglength

The length of the message to be written to the console. A value of zero indicates that no message is to be sent to the operator console.

Notes:

1. The length of the message must be between 1 and 17850 characters for authorized users, and between 1 and 17780 for unauthorized users. The number of lines written to the console is limited to 255. In the case of an unauthorized user, one of those lines is used for the message ID and the user ID of the user. If the message length is exceeded, no lines are written and the service returns an EINVAL error code. If the number of lines is exceeded, the service returns an EINVAL error code, but the first 255 lines are written to the console.
2. An authorized user is one with appropriate privileges, as described in the "Authorization" topic in *z/OS UNIX System Services Programming: Assembler Callable Services Reference*.

__cm2_msg

Pointer to a NULL terminated string containing the message to be written to the console. A value of NULL indicates no message is to be sent to the operator console.

__cm2_routcde

Pointer to an unsigned integer array containing the routing codes to be assigned to the message. The array is terminated by a zero value. Allowable routing codes are 1 to 128 for authorized users, and 1 to 28 for unauthorized users. For more information on routing codes, see *z/OS MVS Programming: Authorized Assembler Services Reference SET-WTO*.

Note: An authorized user is one with appropriate privileges, as described in the "Authorization" topic in *z/OS UNIX System Services Programming: Assembler Callable Services Reference*.

__cm2_descr

Pointer to an unsigned integer array containing the message descriptor codes to be assigned to the message. The array is terminated by a zero value. Allowable descriptor codes are 1 to 13. Descriptor codes 1 through 6, 11, and 12 are mutually exclusive. Codes 7 through 10, and 13 can be assigned in combination with any other code. For more information on descriptor codes, see *z/OS MVS Programming: Authorized Assembler Services Reference SET-WTO*.

__cm2_mcsflag

Specifies one or more of the following flags:

__CONSOLE_HRDCPY

Queue the message for hard copy only. The message will not be displayed on the console.

__cm2_token
Specifies a 4-byte token to be associated with this message. This field is used to identify a group of messages which can be deleted using the DOM feature of the __console2() function. The token must be unique within an address space and can be any value. A token value of zero indicates no token is specified and the message issued will not be associated with any token.

__cm2_msgid
An unsigned 32-bit integer field where the __console2() function will place the message ID associated with the message last sent to the console. This message ID can be used to delete a message when it is no longer needed by specifying it using the DOM feature of the __console2() function. A value of NULL indicates that the message ID is not to be returned.

Note: The value returned in __cm2_msgid is an internal message identifier associated with the message written to the console. The value is not text and it should not be confused with any textual part of a message that might otherwise be considered a message ID.

__cm2_dom_token
Specifies a 4-byte token which represents a message or group of messages to be deleted from the console. All messages previously issued with this token will be deleted from the console. This field is mutually exclusive with __cm2_dom_msgid. A value of zero indicates that no token is specified.

__cm2_dom_msgid
Pointer to an unsigned integer array containing message IDs to be deleted from the console. A maximum of 60 message IDs can be in the array. The array is terminated by a zero value. The array terminator is not part of the 60 message IDs. This field is mutually exclusive with __cm2_dom_token. A value of NULL indicates that no message IDs are specified.

The console communication information structure includes the following additional fields when using the __CONSOLE_FORMAT_3 format. These fields are not available when using the __CONSOLE_FORMAT_2 format.

__cm2_mod_cartptr
Pointer to the 8-byte command and response token (CART) returned by a MODIFY or STOP command.

__cm2_mod_considptr
Pointer to the 4-byte console ID returned by a MODIFY or STOP command.

__cm2_msg_cart
Specifies an 8-byte CART to be used on WTO when the message is issued.

__cm2_msg_consid
Specifies a 4-byte console ID to be used on WTO when the message is issued.

Note: All operations can be done in a single request. The order of operation is to issue messages, delete messages, and then wait for a MODIFY or STOP command.

Returned value

If successful, __console2() returns 0.

If unsuccessful, __console2() returns -1 and sets ERRNO to one of the following values:

Error Code
Description

EFAULT
The __console2() function was unable to address all or part of the *cons* structure, all of part of the routing codes array, all or part of the descriptor codes array, all or part of the array of DOM message IDs, all or part of the *modstr*, or the *concmd* parameter was NULL.

Another possible cause is that __cm2_msgid points to storage which is not accessible.

EINTR
The __console2() function was interrupted by a signal.

EINVAL
The structure pointed to by *cons* contains errors.

For example, mutually exclusive parameters were specified, a non-valid routing code was specified, a non-valid descriptor code or mutually exclusive descriptor codes were specified, or there were more than 60 entries in the array of DOM message IDs.

EMVSERR
A z/OS environmental or internal error has occurred. Use the __errno2() function to obtain diagnostic information that will help determine the cause of the problem.

EPERM
An unauthorized user specified a routing code in the range 29 through 128. Only authorized (superuser) users (UID=0) can specify routing codes in that range.

Example

CELEBC42

```
/* CELEBC42

   This example prints a simple message to the console using the
   __console2() function.  A routing and descriptor code are also
   assigned to the message.

 */
#include <sys/__messag.h>
#include <errno.h>
#include <string.h>
#include <stdio.h>

int main(int argc, char** argv) {
   struct __cons_msg2 cmsg;
   char buf[256] = "A message on the console";
   int rc;
   int cmsg_cmd;
   unsigned int cmsg_rout[2] = {1,0};
   unsigned int cmsg_desc[2] = {12,0};

   /* Fill in the __cons_msg2 struct */
   cmsg.__cm2_format = __CONSOLE_FORMAT_2;
   cmsg.__cm2_msg = buf;
   cmsg.__cm2_msglength = strlen(buf);
   cmsg.__cm2_routcde = cmsg_rout;
   cmsg.__cm2_descr = cmsg_desc;
   cmsg.__cm2_token = 0;
   cmsg.__cm2_msgid = NULL;
   cmsg.__cm2_dom_token = 0;

   rc = __console2(&cmsg,NULL,&cmsg_cmd);
   if(rc == -1) {
      printf("__console2() failed\n");
      printf("%s\n",strerror(errno));
   }
   else {
      printf("__console2() successful. Check console for message\n");
   }

   return 0;
}
```

Related information

- "sys/__messag.h" on page 73
- "__console() — Console communication services" on page 316

ContinueWorkUnit() — Continue WLM work unit

Standards

Standards / Extensions	C or C++	Dependencies
z/OS UNIX	both	

Format

```
#include <sys/__wlm.h>

int ContinueWorkUnit(wlmetok_t *enclavetoken);
```

General description

The ContinueWorkUnit function provides the ability for an application to create a WLM work unit that represents a continuation of the work unit associated with the current home address space.

**enclavetoken*
: Points to a data field of type wlmetok_t where the ContinueWorkUnit() function is to return the WLM work unit enclave token.

Returned value

If successful, ContinueWorkUnit() returns 0.

If unsuccessful, ContinueWorkUnit() returns -1 and sets errno to one of the following values:

Error Code
: **Description**

EFAULT
: An argument of this function contained an address that was not accessible to the caller.

EINVAL
: An argument of this function contained an incorrect value.

EMVSSAF2ERR
An error occurred in the security product.

EMVSWLMERROR
The WLM create enclave failed. Use __errno2() to obtain the WLM service reason code for the failure.

EPERM
The calling thread's address space is not permitted to the BPX.WLMSERVER Facility class. The caller's address space must be permitted to the BPX.WLMSERVER Facility class if it is defined. If BPX.WLMSERVER is not defined, the calling process is not defined as a superuser (UID=0).

Related information

- “sys/__wlm.h” on page 77
- “CheckSchEnv() — Check WLM scheduling environment” on page 271
- “ConnectServer() — Connect to WLM as a server manager” on page 313
- “ConnectWorkMgr() — Connect to WLM as a work manager” on page 315
- “CreateWorkUnit() — Create WLM work unit” on page 344
- “DeleteWorkUnit() — Delete a WLM work unit” on page 378
- “DisconnectServer() — Disconnect from WLM server” on page 383
- “ExtractWorkUnit() — Extract enclave service” on page 457
- “JoinWorkUnit() — Join a WLM work unit” on page 927
- “LeaveWorkUnit() — Leave a WLM work unit” on page 945
- “QueryMetrics() — Query WLM system information” on page 1368
- “QuerySchEnv() — Query WLM scheduling environment” on page 1369

__convert_id_np() — Convert between DCE UUID and user ID

Standards

Standards / Extensions	C or C++	Dependencies
z/OS UNIX	both	

Format

```
#include <unistd.h>

int __convert_id_np(int function_code,
                    char *principal_uuid,
                    char *cell_uuid,
                    char *userid);
```

General description

The __convert_id_np() function is used to retrieve the DCE UUID associated with a userid or the userid associated with a DCE UUID.

This function is intended for DCE servers which process requests from multiple clients. For example, DCE RPC requests from clients are identified by a DCE UUID only. This function enables servers to extract the userid of the requester.

The parameters supported are:

function_code
Identifies whether extracting a userid or UUID. Possible function codes are:

__GET_USERID
Return the userid associated with the specified UUIDs.

__GET_UUID
Return the UUIDs associated with the specified userid.

principal_uuid
When __GET_USERID is specified, *principal_uuid* contains the UUID of the user for the specified userid. When __GET_UUID is specified, *principal_uuid* returns the extracted UUID for the userid specified. The caller must provide a 36-byte field for the returned *principal_uuid*.

cell_uuid
When _GET_USERID is specified, *cell_uuid* should contain the cell UUID if known. If not known, *cell_uuid* must be NULL. When __GET_UUID is specified, *cell_uuid* will return the extracted cell UUID, if it is defined for the specified userid. The caller must provide a 36-byte field for the returned *cell_uuid*.

userid When _GET_USERID is specified, *userid* will return the extracted userid for the specified UUID. The caller must provide a 9 byte field for the returned userid. When __GET_UUID is specified, *userid* contains the userid for whom the UUID should be extracted. The userid must be 1 to 8 characters in length.

Returned value

If successful, __convert_id_np() returns 0.

If unsuccessful, __convert_id_np() returns -1 and sets errno to one of the following values:

Error Code
Description

EINVAL
One of the following errors was detected:
- function_code specified is undefined.
- __GET_UUID was specified and userid is not in the range 1 to 8 characters long.
- __GET_USERID was specified and userid was not 9 character long

EMVSERR
An MVS environmental or internal error occurred.

EMVSSAF2ERR
One of the following errors was detected:
- Received an unexpected return code for the security product.
- The security product detected an error in the input parameters.
- An internal error occurred in the security product.

ENOSYS
One of the following errors was detected:
- No security product is installed on the system.
- The security product does not have support for this function.

ESRCH
One of the following errors was detected:
- No mapping exists between a UUID and Userid.
- No mapping exists between a Userid and UUID.
- The DCEUUIDS class is not active.

- __GET_UUID was specified and no cell UUID is defined for the userid.
- The userid is not defined to the security product.

Related information

- "unistd.h" on page 82

copysign(), copysignf(), copysignl() — Copy the sign from one floating-point number to another

Standards

Standards / Extensions	C or C++	Dependencies
Language Environment C99 Single UNIX Specification, Version 3 C++ TR1 C99	both	OS/390 V2R6

Format

```
#define _AIX_COMPATIBILITY
#include <math.h>
#include <float.h>

double copysign(double x, double y);
```

C99:

```
#define _ISOC99_SOURCE
#include <math.h>
#include <float.h>

float copysignf(float x, float y);
long double copysignl(long double x, long  double y);
```

C++ TR1 C99:

```
#define _TR1_C99
#include <math.h>

float copysign(float x, float y);
long double copysign(long double x, long double y);
```

General description

The copysign functions produce a value with the magnitude of *x* and the sign of *y*.

Restriction: The copysignf() function does not support the _FP_MODE_VARIABLE feature test macro.

Note: The following table shows the viable formats for these functions. See "IEEE binary floating-point" on page 94 for more information about IEEE Binary Floating-Point.

Function	Hex	IEEE
copysign	X	X
copysignf	X	X
copysignl	X	X

Returned value

The copysign functions return a value with the magnitude of *x* and the sign of *y*.

Related information

- "float.h" on page 29
- "math.h" on page 44
- "ilogb(), ilogbf(), ilogbl() — Integer unbiased exponent" on page 835
- "logb(), logbf(), logbl() — Unbiased exponent" on page 991
- "nextafter(), nextafterf(), nextafterl() — Next representable double float" on page 1136
- "scalb() — Load exponent" on page 1463

copysignd32(), copysignd64(), copysignd128() — Copy the sign from one floating-point number to another

Standards

Standards / Extensions	C or C++	Dependencies
C/C++ DFP	both	z/OS V1.8

Format

```
#define __STDC_WANT_DEC_FP__
#include <math.h>

_Decimal32  copysignd32(_Decimal32 x, _Decimal32 y);
_Decimal64  copysignd64(_Decimal64 x, _Decimal64 y);
_Decimal128 copysignd128(_Decimal128 x, _Decimal128 y);

_Decimal32  copysign(_Decimal32 x, _Decimal32 y);   /C++ only */
_Decimal64  copysign(_Decimal64 x, _Decimal64 y);   /C++ only */
_Decimal128 copysign(_Decimal128 x, _Decimal128 y); /C++ only */
```

General description

The copysign functions produce a value with the magnitude of *x* and the sign of *y*.

Notes:

1. To use IEEE decimal floating-point, the hardware must have the Decimal Floating-Point Facility installed.
2. These functions work in IEEE decimal floating-point format. See "IEEE Decimal Floating-Point" for more information.

Returned value

The copysign functions return a value with the magnitude of *x* and the sign of *y*.

Example

```
/* CELEBC47

   This example illustrates the copysignd64() function.

*/

#define  __STDC_WANT_DEC_FP__
#include <math.h>
```

```
#include <stdio.h>

int main(void)
{
   _Decimal64 x = 1.2DD, y = -1.0DD , z;

   z = copysignd64(x, y);

   printf("The result of copysignd64(%Df,%Df) is %Df\n",x,y,z);
}
```

Related information

- "math.h" on page 44
- "copysign(), copysignf(), copysignl() — Copy the sign from one floating-point number to another" on page 326
- "ilogbd32(), ilogbd64(), ilogbd128() — Integer unbiased exponent" on page 837
- "logbd32(), logbd64(), logbd128() — Unbiased exponent" on page 993
- "nextafterd32(), nextafterd64(), nextafterd128() — Next representable decimal floating-point value" on page 1137

cos(), cosf(), cosl() — Calculate cosine

Standards

Standards / Extensions	C or C++	Dependencies
ISO C POSIX.1 XPG4 XPG4.2 ISO/ANSI C++ C99 Single UNIX Specification, Version 3 C++ TR1 C99	both	

Format

```
#include <math.h>

double cos(double x);
float cos(float x);                 /* C++ only */
long double cos(long double x);     /* C++ only */
float cosf(float x);
long double cosl(long double x);
```

General description

Calculates the cosine of *x*. The value *x* is expressed in radians.

Note: These functions work in both IEEE Binary Floating-Point and hexadecimal floating-point formats. See "IEEE binary floating-point" on page 94 for more information about IEEE Binary Floating-Point.

Returned value

Returns the calculated value.

If *x* is outside prescribed limits, the value is not calculated. Instead, the function returns 0 and sets the errno to ERANGE. If the correct value would cause an underflow, zero is returned and the value ERANGE is stored in errno.

Special behavior for XPG4.2: The following error is added:

Error Code
Description

EDOM
The argument exceeded an internal limit for the function (approximately 2^{50}).

Example

CELEBC26

```
/* CELEBC26

   This example calculates y to be the cosine of
   x.

 */
#include <math.h>
#include <stdio.h>

int main(void)
{
   double x, y;

   x = 7.2;
   y = cos(x);

   printf("cos( %lf ) = %lf\n", x, y);
}
```

Output

```
cos( 7.200000 ) = 0.608351
```

Related information

- “math.h” on page 44
- “acos(), acosf(), acosl() — Calculate arccosine” on page 135
- “acosh(), acoshf(), acoshl() — Calculate hyperbolic arccosine” on page 139
- “asin(), asinf(), asinl() — Calculate arcsine” on page 185
- “asinh(), asinhf(), asinhl() — Calculate hyperbolic arcsine” on page 188
- “atan(), atanf(), atanl(), atan2(), atan2f(), atan2l() — Calculate arctangent” on page 192
- “atanh(), atanhf(), atanhl() — Calculate hyperbolic arctangent” on page 195
- “cosh(), coshf(), coshl() — Calculate hyperbolic cosine” on page 331
- “sin(), sinf(), sinl() — Calculate sine” on page 1672
- “sinh(), sinhf(), sinhl() — Calculate hyperbolic sine” on page 1674
- “tan(), tanf(), tanl() — Calculate tangent” on page 1815
- “tanh(), tanhf(), tanhl() — Calculate hyperbolic tangent” on page 1818

cosd32(), cosd64(), cosd128() — Calculate cosine

Standards

Standards / Extensions	C or C++	Dependencies
C/C++ DFP	both	z/OS V1.8

Format

```
#define __STDC_WANT_DEC_FP__
#include <math.h>

_Decimal32  cosd32(_Decimal32 x);
_Decimal64  cosd64(_Decimal64 x);
_Decimal128 cosd128(_Decimal128 x);

_Decimal32  cos(_Decimal32 x);  /* C++ only */
_Decimal64  cos(_Decimal64 x);  /* C++ only */
_Decimal128 cos(_Decimal128 x); /* C++ only */
```

General description

Calculates the cosine of *x*. The value *x* is expressed in radians.

Notes:

1. To use IEEE decimal floating-point, the hardware must have the Decimal Floating-Point Facility installed.
2. These functions work in IEEE decimal floating-point format. See "IEEE Decimal Floating-Point" for more information.

Returned value

Returns the calculated value.

If *x* is outside prescribed limits, the value is not calculated. Instead, the function returns 0 and sets errno to EDOM.

Example

```
/* CELEBC48

   This example illustrates the cosd128() function.

*/

#define  __STDC_WANT_DEC_FP__
#include <math.h>
#include <stdio.h>

int main(void)
{
   _Decimal128 x, y;

   x = 7.2DL;
   y = cosd128(x);

   printf("cosd128(%DDf) = %DDf\n", x, y);
}
```

Related information

- “math.h” on page 44
- “cos(), cosf(), cosl() — Calculate cosine” on page 328
- “sind32(), sind64(), sind128() — Calculate sine” on page 1673

cosh(), coshf(), coshl() — Calculate hyperbolic cosine

Standards

Standards / Extensions	C or C++	Dependencies
ISO C POSIX.1 XPG4 XPG4.2 ISO/ANSI C++ C99 Single UNIX Specification, Version 3 C++ TR1 C99	both	

Format

```
#include <math.h>

double cosh(double x);
float cosh(float x);                  /* C++ only */
long double cosh(long double x);      /* C++ only */
float coshf(float x);
long double coshl(long double x);
```

General description

Calculates the hyperbolic cosine of *x*. The value *x* is expressed in radians.

Note: These functions work in both IEEE Binary Floating-Point and hexadecimal floating-point formats. See "IEEE binary floating-point" on page 94 for more information about IEEE Binary Floating-Point.

Returned value

If the result overflows, the function returns +HUGE_VAL and sets errno to ERANGE.

Example

CELEBC27

```
/* CELEBC27

   This example calculates y to be the hyperbolic cosine of x.

 */
#include <math.h>
#include <stdio.h>

int main(void)
{
   double x,y;

   x = 7.2;
   y = cosh(x);

   printf("cosh( %lf ) = %lf\n", x, y);
}
```

Output

```
cosh( 7.200000 ) = 669.715755
```

Related information

- “math.h” on page 44
- “acos(), acosf(), acosl() — Calculate arccosine” on page 135
- “acosh(), acoshf(), acoshl() — Calculate hyperbolic arccosine” on page 139
- “asin(), asinf(), asinl() — Calculate arcsine” on page 185
- “asinh(), asinhf(), asinhl() — Calculate hyperbolic arcsine” on page 188
- “atan(), atanf(), atanl(), atan2(), atan2f(), atan2l() — Calculate arctangent” on page 192
- “atanh(), atanhf(), atanhl() — Calculate hyperbolic arctangent” on page 195
- “cos(), cosf(), cosl() — Calculate cosine” on page 328
- “sin(), sinf(), sinl() — Calculate sine” on page 1672
- “sinh(), sinhf(), sinhl() — Calculate hyperbolic sine” on page 1674
- “tan(), tanf(), tanl() — Calculate tangent” on page 1815
- “tanh(), tanhf(), tanhl() — Calculate hyperbolic tangent” on page 1818

coshd32(), coshd64(), coshd128() - Calculate hyperbolic cosine

Standards

Standards / Extensions	C or C++	Dependencies
C/C++ DFP	both	z/OS V1.10

Format

```
#define __STDC_WANT_DEC_FP__
#include <math.h>

_Decimal32  coshd32(_Decimal32 x);
_Decimal64  coshd64(_Decimal64 x);
_Decimal128 coshd128(_Decimal128 x);
_Decimal32  cosh(_Decimal32 x);     /* C++ only */
_Decimal64  cosh(_Decimal64 x);     /* C++ only */
_Decimal128 cosh(_Decimal128 x);    /* C++ only */
```

General description

Calculates the hyperbolic cosine of *x*. The value *x* is expressed in radians.

These functions work in IEEE decimal floating-point format. See “IEEE decimal floating-point” on page 95 for more information.

Note: To use IEEE decimal floating-point, the hardware must have the Decimal Floating-Point Facility installed.

Returned value

If the result overflows, the function returns +HUGE_VAL_D32, +HUGE_VAL_D64 or +HUGE_VAL_D128 and sets errno to ERANGE.

Example

CELEBC51

```
/* CELEBC51

   This example illustrates the coshd128() function.

   This example calculates y to be the hyperbolic cosine of x.
```

```
*/

#define  __STDC_WANT_DEC_FP__
#include <math.h>
#include <stdio.h>

int main(void)
{
   _Decimal128 x, y;

   x = 7.2DL;
   y = coshd128(x);

   printf("coshd128( %DDf ) = %DDf\n", x, y);
}
```

Related information

- “math.h” on page 44
- “acosd32(), acosd64(), acosd128() - Calculate arccosine” on page 137
- “acoshd32(), acoshd64(), acoshd128() - Calculate hyperbolic arccosine” on page 140
- “asind32(), asind64(), asind128() - Calculate arcsine” on page 187
- “asinhd32(), asinhd64(), asinhd128() - Calculate hyperbolic arcsine” on page 189
- “atand32(), atand64(), atand128(), atan2d32(), atan2d64(), atan2d128() - Calculate arctangent” on page 194
- “atanhd32(), atanhd64(), atanhd128() - Calculate hyperbolic arctangent” on page 197
- “cosd32(), cosd64(), cosd128() — Calculate cosine” on page 329
- “sind32(), sind64(), sind128() — Calculate sine” on page 1673
- “sinhd32(), sinhd64(), sinhd128() - Calculate hyperbolic sine” on page 1676
- “tand32(), tand64(), tand128() - Calculate tangent” on page 1817
- “tanhd32(), tanhd64(), tanhd128() - Calculate hyperbolic tangent” on page 1820

__cospid32(), __cospid64(), __cospid128() — Calculate cosine of pi *x

Standards

Standards / Extensions	C or C++	Dependencies
Language Environment	both	z/OS V1.8

Format

```
#define __STDC_WANT_DEC_FP__
#include <math.h>

_Decimal32  __cospid32(_Decimal32 x);
_Decimal64  __cospid64(_Decimal64 x);
_Decimal128 __cospid128(_Decimal128 x);
```

General description

Calculates the cosine of pi * *x*. The value *x* is expressed in radians.

Notes:

1. To use IEEE decimal floating-point, the hardware must have the Decimal Floating-Point Facility installed.
2. These functions work in IEEE decimal floating-point format. See "IEEE Decimal Floating-Point" for more information.

Returned value

Returns the calculated value.

If *x* is outside the prescribed limits, the value is not calculated. Instead, the function either returns 1, or returns 0 and sets errno to ERANGE.

Example

```
/* CELEBC49

   This example illustrates the __cospid32() function.

*/

#define  __STDC_WANT_DEC_FP__
#include <math.h>
#include <stdio.h>

int main(void)
{
   _Decimal32 x, y;

   x = 1.0DF;
   y = __cospid32(x);

   printf("__cospid32(%Hf) = %Hf\n", x, y);
}
```

Related information

- "math.h" on page 44
- "cos(), cosf(), cosl() — Calculate cosine" on page 328
- "__sinpid32(), __sinpid64(), __sinpid128() — Calculate sine of pi * x" on page 1677

__cotan(), __cotanf(), __cotanl() — Calculate cotangent

Standards

Standards / Extensions	C or C++	Dependencies
Language Environment	both	z/OS V1R5

Format

```
#include <math.h>

double __cotan(double x);
float __cotanf(float x);
long double __cotanl(long double x);
```

General description

The __cotan functions compute the cotangent of x.

Note: The following table shows the viable formats for these functions. See "IEEE binary floating-point" on page 94 for more information about IEEE Binary Floating-Point.

Function	Hex	IEEE
__cotan	X	X

Function	Hex	IEEE
__cotanf	X	X
__cotanl	X	X

Returned value

The __cotan functions return the cotangent of x.

Related information

- "math.h" on page 44

__cpl() — CPL interface service

Standards

Standards / Extensions	C or C++	Dependencies
Language Environment	both	OS/390 V2R9

Format

```
#include <sys/__cpl.h>

int __cpl(int functioncode, int bufferlen, char *buffer);
```

General description

__cpl() is currently called by the CFSizer (Coupling Facility structure sizer) tool. An IBM customer answers a minimum set of questions from an IBM web page, about one or more IBM products and then clicks the submit button. The *submit* invokes a C *cgi* program that parses the data, calls __cpl() to Query the Coupling Facility or size one or more Coupling Facility structures and then display the results back to the web client browser.

functioncode
A value that specifies what function BPX1CPL will perform. The following function codes are defined.

CPL_QUERY (equates to value of 1)

CPL_CFSIZER (equates to a value of 2)

CPL_CFSIZER_W_LVL (equates to a value of 3)

CFlevel8 or higher is required to use the computesize function. To provide a consistent result, the code must loop through all online CFs and find the one at the highest CF level. Issuing computesize against CFs at different levels gives different sizes back to the user resulting in inconsistent results when multiple requests are issued.

bufferlen
The length of the input/output storage area (*buffer*) for BPX1CPL

buffer Storage area for input/output for BPX1CPL

__cpl() is an interface to the BPX1CPL Assembler Callable Service. For more information on parameters and behavior of BPX1CPL, please refer to *z/OS UNIX System Services Programming: Assembler Callable Services Reference, SA23-2281*.

Returned value

If successful, __cpl() returns 0.

If unsuccessful, __cpl() returns -1 and sets errno to one of the following values:

Error Code
 Description

EFAULT
 One of the parameters contained an address that was not accessible to the caller.

EINVAL
 The *functioncode* parameter contains a value that is not correct.

EMVSCPLERROR
 A __cpl() service failed.

ENOSYS
 The __cpl() service failed because the system is not at the correct level.

EPERM
 The calling thread's address space is not permitted.

Usage notes

1. Access to __cpl() is controlled using a new RACF class profile BPX.CF. For any of these cases to run, a BPX.CF class profile must be created and access level provided.

   __cpl() is only valid on a Parallel Sysplex® enabled system with a CFlevel 8 or higher Coupling Facility. Most installations run with two or more Coupling Facilities for availability and recoverability reasons. As such, the code was designed to provide the flexibility of allowing the caller to specify a CF or if not specified, MVS will select the first CF at CFlevel 8 it finds.

Related information

- “sys/__cpl.h” on page 73

cpow(), cpowf(), cpowl() — Calculate the complex power

Standards

Standards / Extensions	C or C++	Dependencies
C99 Single UNIX Specification, Version 3	both	z/OS V1R7 a compiler that is designed to support C99

Format

```
#include <complex.h>

double complex cpow(double complex x, double complex y);
float complex cpowf(float complex x, float complex y);
long double complex cpowl(long double complex x, long double complex y);
```

General description

The cpow() family of functions computes the complex value of *x* to the power of *y*, with a branch cut for the first parameter along the negative real axis.

Note: The following table shows the viable formats for these functions. See "IEEE binary floating-point" on page 94 for more information about IEEE Binary Floating-Point.

Function	Hex	IEEE
cpow	X	X
cpowf	X	X
cpowl	X	X

Returned value

The cpow() family of functions return the complex power value.

Example

```
/*
 *  This example illustrates the complex power of complex number 'z'
 */
#include <complex.h>
#include <stdio.h>

void main()
{
   long double complex zl=-0.5 + I*0.5,  zpowl=(long double complex)3.0;
   double complex zd=(double complex)zl, zpowd=(double complex)zpowl;
   float complex zf=(float complex)zd,   zpowf=(float complex)zpowl;

   long double resl;
   double resd;
   float resf;

   printf("Illustrates the cpow function. Expected result is 0.25 in all cases\n");
   resd = cpow(zd,zpowd);
   resf = cpowf(zf,zpowf);
   resl = cpowl(zl,zpowl);
   printf("\tcpow(%f + I*%f,%f + I*%f) = %f\n",creal(zd), cimag(zd),
                                           creal(zpowd), cimag(zpowd), resd);
   printf("\tcpowf(%f + I*%f,%f + I*%f) = %f\n",crealf(zd), cimagf(zd),
                                           crealf(zpowf), cimagf(zpowf), resf);
   printf("\tcpowl(%Lf + I*%Lf,%Lf + I*%Lf) = %Lf\n",creall(zl), cimagl(zl),
                                             creall(zpowl), cimagl(zpowl), resl);
}
Output

Illustrates the cpow function. Expected result is 0.25
        cpow(-0.500000 + I*0.500000,3.000000 + I*0.000000) = 0.250000
        cpowf(-0.500000 + I*0.500000,3.000000 + I*0.000000) = 0.250000
        cpowl(-0.500000 + I*0.500000,3.000000 + I*0.000000) = 0.250000
```

Related information

- "complex.h" on page 19
- "cabs(), cabsf(), cabsl() — Calculate the complex absolute value" on page 228

cproj(), cprojf(), cprojl() — Calculate the projection

Standards

Standards / Extensions	C or C++	Dependencies
C99 Single UNIX Specification, Version 3	both	z/OS V1R7 a compiler that is designed to support C99

Format

```
#include <complex.h>

double complex cproj(double complex z);
float complex cprojf(float complex z);
long double complex cprojl(long double complex z);
```

General description

The cproj() family of functions compute a projection of *z* onto the Riemann sphere: *z* projects to *z* except that all complex infinities (even those with one infinite part and one NaN part) project to positive infinity on the real axis. If *z* has an infinite part, then cproj(*z*) is equivalent to:

INFINITY +I * copysign(0.0, cimag(*z*))

Note: The following table shows the viable formats for these functions. See "IEEE binary floating-point" on page 94 for more information about IEEE Binary Floating-Point.

Function	Hex	IEEE
cproj	X	X
cprojf	X	X
cprojl	X	X

Returned value

The cproj() family of functions return the value of the projection onto the Riemann sphere.

For a variable *z* of complex type, z == creal(*z*) + cimag(*z*)*I.

Special behavior for hex: On hexadecimal floating point mode, cproj(z) family of functions always returns z .

Example

```
#include <complex.h>
#include <stdio.h>

/*
 * Illustrates complex function cproj().
 *
 * NOTE:       When compiled in HEX(FLOAT), cproj(z) should
 *             always return z
 */
```

```
#define INFINITEL 1.0L/0.0L   /* An infinite number */

void InitReal ( long double complex *z, long double RealPart)
{
  union LongDoubleComplexMap {
     long double complex w;
     long double ldarray[2];
  }z1;

  z1.w = *z;
  z1.ldarray[0] = RealPart;
  *z = z1.w;
}


main() {

  long double complex z,w;
  z = 2.5 + I*(-3.999);

  printf("Ilustrates function cproj() ");

  #ifdef _BFP_
    printf ("(IEEE mode)");
  #else
    printf ("(HFP mode)");
  #endif

  printf("\n\n z = %Lf + I*%Lf\n\n",creall(z), cimagl(z));
  w = cprojl(z);
  printf(" cproj(z) = %Lf + I*%Lf\n",creall(w), cimagl(w));


  printf(" Initializing z(real) with infinity ...\n");
  InitReal(z,INFINITEL);
  printf(" z = %Lf + %Lf*I\n",creall(z),cimagl(z));


  w = cprojl(z);
  printf(" cproj(z) = %Lf + %Lf*I\n",creall(w),cimagl(w));
}
```

```
Output

Ilustrates function cproj() (IEEE mode)

 z = 2.500000 + I*-3.999000

 cproj(z) = 2.500000 + I*-3.999000
 Initializing z(real) with infinity ...
 z = INF + -3.999000*I
 cproj(z) = INF + -0.000000*I
```

Related information

- "complex.h" on page 19
- "carg(), cargf(), cargl() — Calculate the argument" on page 233
- "cimag(), cimagf(), cimagl() — Calculate the complex imaginary part" on page 280
- "conj(), conjf(), conjl() — Calculate the complex conjugate" on page 307
- "creal(), crealf(), creall() — Calculate the complex real part" on page 340

creal(), crealf(), creall() — Calculate the complex real part

Standards

Standards / Extensions	C or C++	Dependencies
C99 Single UNIX Specification, Version 3	both	z/OS V1R7 a compiler that is designed to support C99

Format

```
#include <complex.h>

double creal(double complex z);
float crealf(float complex z);
long double creall(long double complex z);
```

General description

The creal() family of functions compute the real part of *z*.

Note: The following table shows the viable formats for these functions. See "IEEE binary floating-point" on page 94 for more information about IEEE Binary Floating-Point.

Function	Hex	IEEE
creal	X	X
crealf	X	X
creall	X	X

Returned value

The creal() family of functions return the real part value (as a real).

Example

For an example of a similar function see cacos(), cexp() or cpow().

Related information

- "complex.h" on page 19
- "carg(), cargf(), cargl() — Calculate the argument" on page 233
- "cimag(), cimagf(), cimagl() — Calculate the complex imaginary part" on page 280
- "conj(), conjf(), conjl() — Calculate the complex conjugate" on page 307
- "cproj(), cprojf(), cprojl() — Calculate the projection" on page 338

creat() — Create a new file or rewrite an existing one

Standards

Standards / Extensions	C or C++	Dependencies
POSIX.1 XPG4 XPG4.2 Single UNIX Specification, Version 3	both	

Format

```
#define _POSIX_SOURCE
#include <fcntl.h>

int creat(const char *pathname, mode_t mode);
```

General description

The function call: `creat(pathname,mode)` is equivalent to the call:

```
open(pathname, O_CREAT|O_WRONLY|O_TRUNC, mode);
```

Thus the file named by *pathname* is created, unless it already exists. The file is then opened for writing only, and is truncated to zero length. See "open() — Open a file" on page 1152 for further information.

The *mode* argument specifies the file permission bits to be used in creating the file. Here is a list of symbols that can be used for a mode:

S_IRGRP
Read permission for the file's group.

S_IROTH
Read permission for users other than the file owner.

S_IRUSR
Read permission for the file owner.

S_IRWXG
Read, write, and search, or execute permission for the file's group. S_IRWXG is the bitwise inclusive-OR of S_IRGRP, S_IWGRP, and S_IXGRP.

S_IRWXO
Read, write, and search, or execute permission for users other than the file owner. S_IRWXO is the bitwise inclusive-OR of S_IROTH, S_IWOTH, and S_IXOTH.

S_IRWXU
Read, write, and search, or execute, for the file owner; S_IRWXG is the bitwise inclusive-OR of S_IRUSR, S_IWUSR, and S_IXUSR.

S_ISGID
Privilege to set group ID (GID) for execution. When this file is run through an exec function, the effective group ID of the process is set to the group ID of the file, so that the process has the same authority as the file owner rather than the authority of the actual invoker.

S_ISUID
Privilege to set the user ID (UID) for execution. When this file is run through an exec function, the effective user ID of the process is set to the owner of the file, so that the process has the same authority as the file owner rather than the authority of the actual invoker.

S_ISVTX
Indicates shared text. Keep loaded as an executable file in storage.

S_IWGRP
Write permission for the file's group.

S_IWOTH
Write permission for users other than the file owner.

S_IWUSR
Write permission for the file owner.

S_IXGRP
Search permission (for a directory) or execute permission (for a file) for the file's group.

S_IXOTH
Search permission for a directory, or execute permission for a file, for users other than the file owner.

S_IXUSR
Search permission (for a directory) or execute permission (for a file) for the file owner.

Large file support for z/OS UNIX files: Large z/OS UNIX files are supported automatically for AMODE 64 C/C++ applications. AMODE 31 C/C++ applications must be compiled with the option LANGLVL(LONGLONG) and define the _LARGE_FILES feature test macro before any headers are included to enable this function to operate on z/OS UNIX files that are larger than 2 GB in size. File size and offset fields are enlarged to 63 bits in width. Therefore, any other function operating on the file is required to define the _LARGE_FILES feature test macro as well.

Returned value

If successful, creat() returns a file descriptor for the open file.

If unsuccessful, creat() returns -1 and sets errno to one of the following values:

Error Code
Description

EACCES
One of the following error conditions exists:
- The process did not have search permission on a component in *pathname*.
- The file exists but the process did not have appropriate permissions to open the file in the way specified by the flags.
- The file does not exist, and the process does not have write permission on the directory where the file is to be created.
- O_TRUNC was specified, but the process does not have write permission on the file.

EINTR
open() was interrupted by a signal.

EISDIR
pathname is a directory, and *options* specifies write or read/write access.

ELOOP
A loop exists in symbolic links. This error is issued if the number of symbolic links detected in the resolution of *pathname* is greater than POSIX_SYMLOOP.

EMFILE
The process has reached the maximum number of file descriptors it can have open.

ENAMETOOLONG

pathname is longer than PATH_MAX characters or some component of *pathname* is longer than NAME_MAX characters while _POSIX_NO_TRUNC is in effect. For symbolic links, this error occurs if the length of a *pathname* string substituted for a symbolic link in the *pathname* argument exceeds PATH_MAX. The PATH_MAX and NAME_MAX values can be determined with pathconf().

ENFILE

The system has reached the maximum number of file descriptors it can have open.

ENOENT

O_CREAT is specified, and either the path prefix does not exist or the *pathname* argument is an empty string.

ENOSPC

The directory or file system intended to hold a new file has insufficient space.

ENOTDIR

A component of *pathname* is not a directory.

EOVERFLOW

The named file is a regular file and the size of the file cannot be represented correctly in an object of type off_t

EROFS

pathname is on a read-only file system.

Example

CELEBC28

```
/* CELEBC28

   This example creates a file.

 */
#define _POSIX_SOURCE
#include <fcntl.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <unistd.h>
#undef _POSIX_SOURCE
#include <stdio.h>

main() {
  char fn[]="creat.file", text[]="This is a test";
  int fd;

  if ((fd = creat(fn, S_IRUSR | S_IWUSR)) < 0)
    perror("creat() error");
  else {
    write(fd, text, strlen(text));
    close(fd);
    unlink(fn);
  }
return(fd);
}
```

Related information

- “fcntl.h” on page 27
- “sys/stat.h” on page 75

- “sys/types.h” on page 75
- “close() — Close a file” on page 288
- “open() — Open a file” on page 1152
- “unlink() — Remove a directory entry” on page 1962

CreateWorkUnit() — Create WLM work unit

Standards

Standards / Extensions	C or C++	Dependencies
z/OS UNIX	both	

Format

```
#include <sys/__wlm.h>

int CreateWorkUnit(wlmetok_t *enclavetoken,
                   server_classify_t classify,
                   char *arrival_time,
                   char *func_name);
```

General description

The CreateWorkUnit function provides the ability for an application to create a WLM work unit.

**enclavetoken*
: Points to a data field of type wlmetok_t where the CreateWorkUnit() function is to return the WLM work unit enclave token.

**classify*
: Points to a server_classify_t structure that contains the classification information for the work request macro.

**arrival_time*
: Address of a doubleword (unsigned long long) field that contains the arrival time of the work request in STCK format.

**func_name*
: Points to a NULL-terminated character string that represents the descriptive function name of the associated work request.

Returned value

If successful, CreateWorkUnit() returns a pointer to work unit enclave token of type wlmetok_t.

If unsuccessful, CreateWorkUnit() returns -1 and sets errno to one of the following values:

Error Code
: **Description**

EFAULT
: An argument of this function contained an address that was not accessible to the caller.

EINVAL
: An argument of this function contained an incorrect value.

EMVSSAF2ERR
An error occurred in the security product.

EMVSWLMERROR
The WLM create failed. Use __errno2() to obtain the WLM service reason code for the failure.

EPERM
The calling thread's address space is not permitted to the BPX.WLMSERVER Facility class. The caller's address space must be permitted to the BPX.WLMSERVER Facility class if it is defined. If BPX.WLMSERVER is not defined, the calling process is not defined as a superuser (UID=0).

Related information

- "sys/__wlm.h" on page 77
- "CheckSchEnv() — Check WLM scheduling environment" on page 271
- "ConnectServer() — Connect to WLM as a server manager" on page 313
- "ConnectWorkMgr() — Connect to WLM as a work manager" on page 315
- "ContinueWorkUnit() — Continue WLM work unit" on page 323
- "DeleteWorkUnit() — Delete a WLM work unit" on page 378
- "DisconnectServer() — Disconnect from WLM server" on page 383
- "ExtractWorkUnit() — Extract enclave service" on page 457
- "JoinWorkUnit() — Join a WLM work unit" on page 927
- "LeaveWorkUnit() — Leave a WLM work unit" on page 945
- "QueryMetrics() — Query WLM system information" on page 1368
- "QuerySchEnv() — Query WLM scheduling environment" on page 1369

crypt() — String encoding function

Standards

Standards / Extensions	C or C++	Dependencies
XPG4 XPG4.2 Single UNIX Specification, Version 3	both	

Format

```
#define _XOPEN_SOURCE
#include <unistd.h>

char *crypt(const char *key, const char *salt);
```

General description

The crypt() function encodes the string pointed to by the *key* argument. It perturbs the Data Encryption Standard (DES) encryption algorithm with the first two characters in the string pointed to by the *salt* argument to perform this encoding. The first two *salt* characters must be chosen from the set:

```
a b c d e f g h i j k l m n o p q r s t u v w x y z
A B C D E F G H I J K L M N O P Q R S T U V W X Y Z
0 1 2 3 4 5 6 7 8 9 . /
```

This function can be called from any thread.

Returned value

If successful, crypt() returns a pointer to a thread specific encoded string. The first two characters of the returned value are those of the *salt* argument.

Notes:

1. The return value of crypt() points to a thread-specific buffer which is overwritten each time crypt() is called from the same thread.
2. The values returned by crypt() are not portable to other X/Open-conforming systems.

If unsuccessful, crypt() returns a NULL pointer and sets errno to indicate the error.

Special behavior for z/OS UNIX Services: The crypt() function will fail if:

Error Code
Description

EINVAL
First two characters of *salt* argument are not from the *salt* set.

ENOMEM
Storage for crypt() output buffer is not available for thread from which crypt() has been invoked.

Related information

- "unistd.h" on page 82
- "encrypt() — Encoding function" on page 419
- "setkey() — Set encoding key" on page 1551

cs() — Compare and swap

Standards

Standards / Extensions	C or C++	Dependencies
Language Environment	both	

Format

```
#include <stdlib.h>

int cs(cs_t *oldptr, cs_t *curptr, cs_t newword);
```

General description

The cs() built-in function compares the 4-byte value pointed to by *oldptr* to the 4-byte value pointed to by *curptr*. If they are equal, the 4-byte value, *newword*, is copied into the location pointed to by *curptr*. If they are unequal, the value pointed to by *curptr* is copied into the location pointed to by *oldptr*.

If this function is used in a multi-threading environment, then it is the users responsibility to protect the *oldptr variable*. The user can create a local variable per thread to contain this value or provide locking code to protect the global variable used. The *oldptr* variable may not reflect the *curptr* variable if the *curptr* variable changes via another thread before the user has a chance to examine *oldptr*.

To avoid infringing on the user's name space, this nonstandard function is exposed only when you use the compiler option LANGLVL(EXTENDED). When you use LANGLVL(EXTENDED) any relevant information in the header is also exposed.

The function uses the COMPARE SWAP (CS) instructions, which can be used in multiprogramming or multiprocessing environments to serialize access to counters, flags, control words, and other common storage areas. For a detailed description, refer to the appendixes in the *z/Architecture Principles of Operation* on number representation and instruction.

Returned value

cs() returns 0 if the 4-byte value pointed to by *oldptr* is equal to the 4-byte value pointed to by *curptr*.

If the value is not equal, cs() returns 1.

Related information

- *z/Architecture Principles of Operation*
- “stdlib.h” on page 70
- “cds() — Compare double and swap” on page 247

csid() — Character set ID for multibyte character

Standards

Standards / Extensions	C or C++	Dependencies
Language Environment	both	

Format

```
#include <stdlib.h>

int csid(const char *c)
```

External entry point: `@@CSID`, `__csid`;

General description

Determines the character set identifier for the specified multibyte character pointed to by *c*, that begins in the initial shift state.

To avoid infringing on the user's name space, this nonstandard function has two names. One name is prefixed with two underscore characters, and one name is not. The name without the prefix underscore characters is exposed only when you use LANGLVL(EXTENDED).

To use this function, you must either invoke the function using its external entry point name (that is, the name that begins with two underscore characters), or compile with LANGLVL(EXTENDED). When you use LANGLVL(EXTENDED) any relevant information in the header is also exposed.

Returned value

If successful, csid() returns the character-set identifier for the multibyte character.

If the character is not valid, csid() returns -1.

Note: The multibyte character passed must begin in the initial shift state.

Example

CELEBC29

```
/* CELEBC29

   This example checks character set ID for a character.

 */
#include "locale.h"
#include "stdio.h"
#include "stdlib.h"

main() {
   char *string = "A";
   int   rc;

   rc = csid(string);
   printf("character '%s' is in character set id %i\n", string, rc);
}
```

Output

```
character 'A' is in character set id 0
```

Related information

- "stdlib.h" on page 70

csin(), csinf(), csinl() — Calculate the complex sine

Standards

Standards / Extensions	C or C++	Dependencies
C99 Single UNIX Specification, Version 3	both	z/OS V1R7 a compiler that is designed to support C99

Format

```
#include <complex.h>

double complex csin(double complex z);
float complex csinf(float complex z);
long double complex csinl(long double complex z);
```

General description

The csin() family of functions compute the complex sine of *z*.

Note: The following table shows the viable formats for these functions. See "IEEE binary floating-point" on page 94 for more information about IEEE Binary Floating-Point.

Function	Hex	IEEE
csin	X	X
csinf	X	X
csinl	X	X

Returned value

The csin() family of functions return the complex sine value.

Example

For an example of a similar function see cacos(), cexp() or cpow().

Related information

- "complex.h" on page 19
- "csinh(), csinhf(), csinhl() — Calculate the complex hyperbolic sine"
- "casinh(), casinhf(), casinhl() — Calculate the complex arc hyperbolic sine" on page 235
- "casin(), casinf(), casinl() — Calculate the complex arc sine" on page 234
- "ccos(), ccosf(), ccosl() — Calculate the complex cosine" on page 244
- "ccosh(), ccoshf(), ccoshl() — Calculate the complex hyperbolic cosine" on page 245

csinh(), csinhf(), csinhl() — Calculate the complex hyperbolic sine

Standards

Standards / Extensions	C or C++	Dependencies
C99 Single UNIX Specification, Version 3	both	z/OS V1R7 a compiler that is designed to support C99

Format

```
#include <complex.h>

double complex csinh(double complex z);
float complex csinhf(float complex z);
long double complex csinhl(long double complex z);
```

General description

The csinh() family functions compute the complex hyperbolic sine of *z*.

Note: The following table shows the viable formats for these functions. See "IEEE binary floating-point" on page 94 for more information about IEEE Binary Floating-Point.

Function	Hex	IEEE
csinh	X	X
csinhf	X	X
csinhl	X	X

Returned value

The csinh() family functions return the complex hyperbolic sine value.

Example

For an example of a similar function see cacos(), cexp() or cpow().

Related information

- "complex.h" on page 19
- "csin(), csinf(), csinl() — Calculate the complex sine" on page 348
- "casinh(), casinhf(), casinhl() — Calculate the complex arc hyperbolic sine" on page 235
- "casin(), casinf(), casinl() — Calculate the complex arc sine" on page 234
- "ccos(), ccosf(), ccosl() — Calculate the complex cosine" on page 244
- "ccosh(), ccoshf(), ccoshl() — Calculate the complex hyperbolic cosine" on page 245
- "cacos(), cacosf(), cacosl() — Calculate the complex arc cosine" on page 229
- "cacosh(), cacoshf(), cacoshl() — Calculate the complex arc hyperbolic cosine" on page 230

__CSNameType() — Return codeset name type

Standards

Standards / Extensions	C or C++	Dependencies
z/OS UNIX	both	z/OS V1R2

Format

```
#include <_Ccsid.h>

__csType __CSNameType(char *codesetName);
```

General description

The __CSNameType() function returns a __csType value which indicates the corresponding codeset name type.

Returned value

If *codesetName* is valid, __CSNameType() returns one of the following __csType values, which are defined in <_Ccsid.h>:

- _CSTYPE_EBCDIC
- _CSTYPE_ASCII
- _CSTYPE_UCS2
- _CSTYPE_UTF8
- _CSTYPE_UTF16
- _CSTYPE_UTF32

If *codesetName* is not valid, __CSNameType() returns _CSTYPE_INVALID.

Related information

- "_Ccsid.h" on page 17
- "__CcsidType() — Return coded character set ID type" on page 246

- "__toCcsid() — Convert codeset name to coded character set ID" on page 1896
- "__toCSName() — Convert coded character set ID to codeset name" on page 1897

csnap() — Request a condensed dump

Standards

Standards / Extensions	C or C++	Dependencies
Language Environment	both	

Format

```
#include <ctest.h>

int csnap(char *dumptitle);
```

General description

Creates a display of the activation stack, including the Dynamic Storage Area (DSA), for each presently active function. Other environmental control blocks that may be required by IBM Service are also displayed. Under Language Environment, these consist of the Common Anchor Area (CAA) and the z/OS XL C/C++ CAA information. The output is identified with *dumptitle*. See the CEE3DMP Language Environment callable service in *z/OS Language Environment Programming Guide, SA38-0682*, to determine where the output is written to.

To avoid infringing on the user's name space, this nonstandard function has two names. One name is prefixed with two underscore characters, and one name is not. The name without the prefix underscore characters is exposed only when you use LANGLVL(EXTENDED).

To use this function, you must either invoke the function using its external entry point name (that is, the name that begins with two underscore characters), or compile with LANGLVL(EXTENDED). When you use LANGLVL(EXTENDED) any relevant information in the header is also exposed.

Returned value

If successful, csnap() returns 0.

If unsuccessful, csnap() returns nonzero.

Example

```
#include <ctest.h>

int main(void) {

   int rc;
   rc = csnap("Sample csnap output");
}
```

Related information

- *IBM Language Environment Programming Guide*
- "ctest.h" on page 21
- "cdump() — Request a main storage dump" on page 247
- "ctrace() — Request a traceback" on page 363

__csplist — Retrieve CSP parameters

Standards

Standards / Extensions	C or C++	Dependencies
Language Environment	C only	

Format

```
#include <csp.h>

__csplist;
```

General description

Restriction: This function is not supported in AMODE 64.

__csplist is a macro intended to be used to access the parameter list passed from Cross System Product (CSP) to your C Library program. The macro evaluates to the address of the first element of the parameter list. You can use array indexing to extract the subsequent parameters, casting each parameter to the expected type, as shown in the example below. If no parameters are passed, `__csplist[0]` equals NULL.

You must include the #pragma `runopts(plist(ims))` directive if CSP is used to invoke a z/OS XL C program.

`argc` will always be 1. See *z/OS XL C/C++ User's Guide* for information about the PLIST compiler option.

If you are expecting an integer and then a structure of type `s_type`, you should have the statements:

```
int_var = (int *) __csplist[0];
s_var   = (s_type *) __csplist[1];
```

Related information

- "csp.h" on page 20

csqrt(), csqrtf(), csqrtl() — Calculate the complex square root

Standards

Standards / Extensions	C or C++	Dependencies
C99 Single UNIX Specification, Version 3	both	z/OS V1R7 a compiler that is designed to support C99

Format

```
##include <complex.h>

double complex csqrt(double complex z);
float complex csqrtf(float complex z);
long double complex csqrtl(long double complex z);
```

General description

The csqrt() family of functions compute the complex square root of *z*, with a branch cut along the negative real axis.

Note: The following table shows the viable formats for these functions. See "IEEE binary floating-point" on page 94 for more information about IEEE Binary Floating-Point.

Function	Hex	IEEE
csqrt	X	X
csqrtf	X	X
csqrtl	X	X

Returned value

The csqrt() family of functions return the complex square root value, in the range of the right half-plane (including the imaginary axis).

Example

For an example of a similar function see cacos(), cexp() or cpow().

Related information

- "complex.h" on page 19
- "cabs(), cabsf(), cabsl() — Calculate the complex absolute value" on page 228
- "cpow(), cpowf(), cpowl() — Calculate the complex power" on page 336

ctan(), ctanf(), ctanl()— Calculate the complex tangent

Standards

Standards / Extensions	C or C++	Dependencies
C99 Single UNIX Specification, Version 3	both	z/OS V1R7 a compiler that is designed to support C99

Format

```
#include <complex.h>

double complex ctan(double complex z);
float complex ctanf(float complex z);
long double complex ctanl(long double complex z);
```

General description

The ctan() family of functions compute the complex tangent of *z*.

Note: The following table shows the viable formats for these functions. See "IEEE binary floating-point" on page 94 for more information about IEEE Binary Floating-Point.

Function	Hex	IEEE
ctan	X	X
ctanf	X	X
ctanl	X	X

Returned value

The ctan() family of functions return the complex tangent value.

Example

For an example of a similar function see cacos(), cexp() or cpow().

Related information

- “complex.h” on page 19
- “ctanh(), ctanhf(), ctanhl() — Calculate the complex hyperbolic tangent”
- “catanh(), catanhf(), catanhl() — Calculate the complex arc hyperbolic tangent” on page 237
- “catan(), catanf(), catanl() — Calculate the complex arc tangent” on page 236
- “csin(), csinf(), csinl() — Calculate the complex sine” on page 348
- “csinh(), csinhf(), csinhl() — Calculate the complex hyperbolic sine” on page 349

ctanh(), ctanhf(), ctanhl() — Calculate the complex hyperbolic tangent

Standards

Standards / Extensions	C or C++	Dependencies
C99 Single UNIX Specification, Version 3	both	z/OS V1R7 a compiler that is designed to support C99

Format

```
#include <complex.h>

double complex ctanh(double complex z);
float complex ctanhf(float complex z);
long double complex ctanhl(long double complex z);
```

General description

The ctanh() family of functions compute the complex hyperbolic tangent of *z*.

Note: The following table shows the viable formats for these functions. See “IEEE binary floating-point” on page 94 for more information about IEEE Binary Floating-Point.

Function	Hex	IEEE
ctanh	X	X
ctanhf	X	X
ctanhl	X	X

Returned value

The ctanh() family of functions return the complex hyperbolic tangent value.

Example

For an example of a similar function see cacos(), cexp() or cpow().

Related information

- "complex.h" on page 19
- "ctan(), ctanf(), ctanl()— Calculate the complex tangent" on page 353
- "catan(), catanf(), catanl() — Calculate the complex arc tangent" on page 236
- "catanh(), catanhf(), catanhl() — Calculate the complex arc hyperbolic tangent" on page 237
- "casinh(), casinhf(), casinhl() — Calculate the complex arc hyperbolic sine" on page 235
- "casin(), casinf(), casinl() — Calculate the complex arc sine" on page 234
- "csin(), csinf(), csinl() — Calculate the complex sine" on page 348
- "csinh(), csinhf(), csinhl() — Calculate the complex hyperbolic sine" on page 349
- "ccos(), ccosf(), ccosl() — Calculate the complex cosine" on page 244
- "ccosh(), ccoshf(), ccoshl() — Calculate the complex hyperbolic cosine" on page 245
- "cacos(), cacosf(), cacosl() — Calculate the complex arc cosine" on page 229
- "cacosh(), cacoshf(), cacoshl() — Calculate the complex arc hyperbolic cosine" on page 230

ctdli() — Call to DL/I

Standards

Standards / Extensions	C or C++	Dependencies
Language Environment	both	

Format

C only:

```
#pragma runopts(env(IMS),plist(OS))
#include <ims.h>             /* or #include <cics.h> */

#define _CTDLI_PARMCOUNT     /* First arg is an explicit parameter count */
int ctdli(int parmcount, const char *function, ...);

or

#define _CTDLI_NOPARMCOUNT   /*Parameter count is implicit in varargs */
int ctdli(const char *function,...);
```

C++:

```
#include <ims.h>        /* or #include <cics.h> */

int ctdli(int parmcount, const char *function, ...);

or

#define _CTDLI_NOPARMCOUNT
int ctdli(const char *function, ...);
```

General description

Restriction: This function is not supported in AMODE 64.

Invokes DL/I facilities. The *parmcount* argument is optional. The *parmcount* value specifies the number of arguments in the variable argument list for the ctdli() call to function.

In C, when specifying a *parmcount*, use the _CTDLI_PARMCOUNT feature test macro. Otherwise, for C or C++, define _CTDLI_NOPARMCOUNT and make *function* the first argument. If the compile unit contains both types of call (sometimes passing *parmcount* and sometimes not), and if you want to avoid messages when compiling with the checkout option, define _CTDLI_NOPARMCOUNT and always cast the first argument to (const char *).

The *function* argument specifies the DL/I function you want to perform. Because the format of the ctdli() call depends on the function selected, all of the variations are not given here. For complete details on the available functions, refer to the COBOL publications.

To avoid infringing on the user's name space, this nonstandard function has two names. One name is prefixed with two underscore characters, and one name is not. The name without the prefix underscore characters is exposed only when you use LANGLVL(EXTENDED).

To use this function, you must either invoke the function using its external entry point name (that is, the name that begins with two underscore characters), or compile with LANGLVL(EXTENDED). When you use LANGLVL(EXTENDED) any relevant information in the header is also exposed.

To invoke ctdli() from an IMS transaction, you need either the #pragma runopts(env(ims),plist(os)), or you need to specify the compiler options TARGET(IMS) and PLIST(OS).

Returned value

The Program Control Block (PCB) status field (2 bytes) is stored as an `unsigned int` and used as the returned value for ctdli().

If the PCB status field contains blanks (hex '4040'), ctdli() returns 0.

Example

```
/* The following program demonstrates the use of the ctdli() function.
   It is a skeleton of a message processing program that calls ctdli()
   to retrieve messages and data.

   Do use the TARGET(IMS) and PLIST(IMS) compile options for C++
   applications.
 */
#ifndef __cplusplus
#pragma runopts(env(ims),plist(os))
#endif

#include <stdlib.h>
#include <ims.h>
#define n     20          /* I/O area size - Application dependent  */
typedef struct {PCB_STRUCT(10)} PCB_10_TYPE;
```

```
int main(void)
{
   /* Function codes for ctdli */
   static const char func_GU[4]   = "GU  ";
   static const char func_ISRT[4] = "ISRT";

   char ssa_name[] = "ORDER   (ORDERKEY   = 666666)";

   int rc;

   char msg_seg_io_area[n];
   char db_seg_io_area[n];
   char alt_msg_seg_out[n];

   PCB_STRUCT_8_TYPE *alt_pcb;
   PCB_10_TYPE *db_pcb;
   IO_PCB_TYPE *io_pcb;

   io_pcb = (IO_PCB_TYPE *)(__pcblist)[0];
   alt_pcb = __pcblist[1];
   db_pcb = (PCB_10_TYPE *) __pcblist[2];
⋮
   /* Get first message segment from message region */
   rc = ctdli(func_GU, io_pcb, msg_seg_io_area);
⋮
   /* Get the data from the database having the specified key value */
   rc = ctdli(func_GU, db_pcb, db_seg_io_area, ssa_name);
⋮
   /* Build output message in program's I/O area */
   rc = ctdli(func_ISRT, alt_pcb, alt_msg_seg_out);
⋮
}
```

Related information

- "ims.h" on page 32

ctermid() — Generate path name for controlling terminal

Standards

Standards / Extensions	C or C++	Dependencies
POSIX.1 XPG4 XPG4.2 Single UNIX Specification, Version 3	both	

Format

```
#define _POSIX_SOURCE
#include <unistd.h>

char *ctermid(char *string);
```

General description

string points to a memory location where the ctermid() function stores the name of the current controlling terminal. The memory location must be able to hold at least L_ctermid characters, where L_ctermid is a symbol defined in the stdio.h header file.

ctermid() returns a string that can be used as a path name to refer to the controlling terminal for the current process. If *string* is not NULL, ctermid() stores the path name in the specified location and returns the value of *string*. Otherwise, ctermid() uses a location of its own and returns a pointer to that location.

The path name returned can be used to access the controlling terminal, if the process has a controlling terminal.

Returned value

ctermid() is always successful; it returns a string that can be used as a path name to refer to the controlling terminal for the current process.

There are no documented errno values.

Example

CELEBC32

```
/* CELEBC32

   This example refers to the controlling terminal for
   the current process.

 */
#define _POSIX_SOURCE
#include <unistd.h>
#include <stdio.h>

main() {
  char termid[1025];

  if (ctermid(termid) == NULL)
    perror("ctermid() error");
  else
    printf("The control terminal is %s\n", termid);
}
```

Output

```
The control terminal is /dev/tty
```

Related information

- “stdio.h” on page 68
- “unistd.h” on page 82
- “ttyname() — Get the name of a terminal” on page 1931

ctest() — Start debug tool

Standards

Standards / Extensions	C or C++	Dependencies
Language Environment	both	

Format

```
#include <ctest.h>

int ctest(char *command);
```

General description

Invokes the Debug Tool from your application program. The parameter *command* is a character pointer to a list of valid Debug Tool commands, that ctest() uses to invoke Debug Tool.

If you choose not to compile your program with hooks, you can use well-placed ctest() function calls instead. (A *hook* is a conditional exit that transfers control to the debugger, when the code is run under the debugger.) You would create a hook when you compile with the TEST option, causing the exit to be in your generated code waiting to run. A hook has minimal effect on a program that is running without the debugger.

To avoid infringing on the user's name space, this nonstandard function has two names. One name is prefixed with two underscore characters, and one name is not. The name without the prefix underscore characters is exposed only when you use LANGLVL(EXTENDED).

To use this function, you must either invoke the function using its external entry point name (that is, the name that begins with two underscore characters), or compile with LANGLVL(EXTENDED). When you use LANGLVL(EXTENDED) any relevant information in the header is also exposed.

Returned value

If successful, ctest() returns 0.

If unsuccessful, ctest() returns nonzero.

Examples

To let the debug tool gain control of your program, issue the command: `ctest(NULL)`.

To display the call chain from within a program and then let the program continue execution, issue the function call: `ctest("list calls; go;")`. To set a breakpoint from within a ctest() call, try:

```
char *cmd = "at line 17 list my_struct; go;";
ctest(cmd);
```

Related information

- "ctest.h" on page 21

ctime(), ctime64() — Convert time to character string

Standards

Standards / Extensions	C or C++	Dependencies
ISO C POSIX.1 XPG4 XPG4.2 C99 Single UNIX Specification, Version 3 Language Environment	both	

Format

```
#include <time.h>

char *ctime(const time_t *timer);
```

```
#define _LARGE_TIME_API
#include <time.h>

char  *ctime64 (const time64_t *timer);
```

General description

Converts the calendar time pointed to by *timer* to local time in the form of a character string. A value for *timer* is usually obtained by a call to the time() function.

The ctime() function is equivalent to the function call: `asctime(localtime(timer))`

The function ctime64() will behave exactly like ctime() except it will convert a time64_t value pointing to a calendar time beyond 03:14:07 UTC on January 19, 2038 with a limit of 23:59:59 UTC on December 31, 9999.

Returned value

If successful, ctime() returns a pointer to a date and time string. The string returned by ctime() contains exactly 26 characters and has the format:

```
   "%.3s %.3s%3d %.2d:%.2d:%.2d %d\n"
```

For example: `Mon Jul 16 02:03:55 1987\n\0`

If an error occurs, ctime() returns no value.

Notes:

1. This function is sensitive to time zone information which is provided by:
   - The TZ environmental variable when POSIX(ON) and TZ is correctly defined, or by the _TZ environmental variable when POSIX(OFF) and _TZ is correctly defined.
   - The LC_TOD category of the current locale if POSIX(OFF) or TZ is not defined.

   The time zone external variables `tzname`, `timezone`, and `daylight` declarations remain feature test protected in time.h.
2. The calendar time returned by a call to the time() function begins at epoch, which was at 00:00:00 Coordinated Universal Time (UTC), January 1, 1970.
3. The ctime() function uses a 24-hour clock format.
4. The days are abbreviated to: `Sun`, `Mon`, `Tue`, `Wed`, `Thu`, `Fri`, and `Sat`.
5. The months are abbreviated to: `Jan`, `Feb`, `Mar`, `Apr`, `May`, `Jun`, `Jul`, `Aug`, `Sep`, `Oct`, `Nov`, and `Dec`.
6. All fields have a constant width.
7. Dates with only one digit are padded with a blank space. Single digit time values are padded with a zero.
8. The newline character (`\n`) and the NULL character (`\0`) occupy the last two positions of the string.

9. The asctime(), ctime(), and other time functions may use a common, statically allocated buffer for holding the return string. Each call to one of these functions may destroy the result of the previous call.

When neither TZ nor _TZ is defined, the current locale is interrogated for time zone information. If neither TZ nor _TZ is defined and LC_TOD time zone information is not present in the current locale, a default value is applied to local time. POSIX programs simply default to Coordinated Universal Time (UTC), while non-POSIX programs establish an offset from UTC based on the setting of the system clock.

For more information about customizing a time zone to work with local time, see “Customizing a time zone” in *z/OS XL C/C++ Programming Guide*.

Error Code
Description

EOVERFLOW
The result cannot be represented.

Example

CELEBC33

```
/* CELEBC33

   This example polls the system clock by using the library
   function &ttime..
   It then prints a message giving the current date and time.

 */
#include <time.h>
#include <stdio.h>

int main(void)
{
   time_t ltime;

   time(&ltime);
   printf("the time is %s", ctime(&ltime));
}
```

Output

```
the time is Fri Jun 16 16:03:38 2006
```

Related information

- “locale.h” on page 40
- “time.h” on page 79
- “asctime(), asctime64() — Convert time to character string” on page 182
- “asctime_r(), asctime64_r() — Convert date and time to a character string” on page 184
- “ctime_r(), ctime64_r() — Convert time value to date and time character string” on page 362
- “gmtime(), gmtime64() — Convert time to broken-down UTC time” on page 810
- “gmtime_r(), gmtime64_r() — Convert a time value to broken-down UTC time” on page 812
- “localdtconv() — Date and time formatting convention inquiry” on page 981
- “localtime(), localtime64() — Convert time and correct for local time” on page 984
- “localtime_r(), localtime64_r() — Convert time value to broken-down local time” on page 986

- “mktime(), mktime64() — Convert local time” on page 1078
- “strftime() — Convert to formatted time” on page 1743
- “time(),time64() — Determine current UTC time” on page 1878
- “tzset() — Set the time zone” on page 1936

ctime_r(), ctime64_r() — Convert time value to date and time character string

Standards

Standards / Extensions	C or C++	Dependencies
Single UNIX Specification, Version 2 Single UNIX Specification, Version 3 Language Environment	both	OS/390 V2R8

Format

```
#define _XOPEN_SOURCE 500
#include <time.h>

char *ctime_r(const time_t *clock, char *buf);
```

```
#define _LARGE_TIME_API
#include <time.h>

char  *ctime64_r (const time64_t *clock, char *buf);
```

General description

The ctime_r() function converts the calendar time pointed to by *clock* to local time in exactly the same form as ctime() and puts the string into the array pointed to by *buf*. (which contains at least 26 bytes) and returns *buf*.

Unlike ctime(), the thread-safe version ctime_r() is not required to set *tzname*.

The function ctime64_r() will behave exactly like ctime_r() except it will convert a time64_t value pointing to a calendar time beyond 03:14:07 UTC on January 19, 2038 with a limit of 23:59:59 UTC on December 31, 9999.

Returned value

If successful, ctime_r() returns a pointer to the string pointed to by *buf*.

If unsuccessful, ctime_r() returns a NULL pointer.

There are no documented errno values.

Related information

- “locale.h” on page 40
- “time.h” on page 79
- “asctime(), asctime64() — Convert time to character string” on page 182
- “asctime_r(), asctime64_r() — Convert date and time to a character string” on page 184
- “ctime(), ctime64() — Convert time to character string” on page 359
- “gmtime(), gmtime64() — Convert time to broken-down UTC time” on page 810
- “gmtime_r(), gmtime64_r() — Convert a time value to broken-down UTC time” on page 812

- "localdtconv() — Date and time formatting convention inquiry" on page 981
- "localtime(), localtime64() — Convert time and correct for local time" on page 984
- "localtime_r(), localtime64_r() — Convert time value to broken-down local time" on page 986
- "mktime(), mktime64() — Convert local time" on page 1078
- "strftime() — Convert to formatted time" on page 1743
- "time(),time64() — Determine current UTC time" on page 1878
- "tzset() — Set the time zone" on page 1936

ctrace() — Request a traceback

Standards

Standards / Extensions	C or C++	Dependencies
Language Environment	both	

Format

```
#include <ctest.h>

int ctrace(char *dumptitle);
```

General description

Requests a traceback. The output is identified with *dumptitle*. ctrace() invokes the CEE3DMP Language Environment callable service with the following options: TRACEBACK, NOFILE, NOBLOCK, NOVARIABLE, NOSTORAGE, STACKFRAME(ALL), NOCOND, NOENTRY. See the CEE3DMP Language Environment callable service in *z/OS Language Environment Programming Guide, SA38-0682*. to determine where the output is written to.

If you compile the code using the GONUMBER option, this function will display, along with the traceback, the statement numbers and the offset information.

To avoid infringing on the user's name space, this nonstandard function has two names. One name is prefixed with two underscore characters, and one name is not. The name without the prefix underscore characters is exposed only when you use LANGLVL(EXTENDED).

To use this function, you must either invoke the function using its external entry point name (that is, the name that begins with two underscore characters), or compile with LANGLVL(EXTENDED). When you use LANGLVL(EXTENDED) any relevant information in the header is also exposed.

Note: The offsets displayed by ctrace() are from the beginning of the functions, whereas by default, compiler listings show offsets from the beginning of the source file. You can override the displayed offsets with the OFFSET compile-time option.

Returned value

If successful, ctrace() returns 0.

If unsuccessful, ctrace() returns nonzero.

Example

CELEBC34

```
/* CELEBC34

   This example shows how ctrace() is used and the output produced.

 */
#include <ctest.h>
int main(void) {

    int rc;
    rc = ctrace("Sample ctrace output");
}
```

Output for C++:

```
CEE3DMP: Sample ctrace output                                Language Environment for MVS
06/16/95 6:13:31 PMPage:    1

Information for enclave ????????

Information for thread 8000000000000000

Traceback:
DSA Addr  Program Unit  PU Addr   PU Offset  Entry         E Addr    E  Offset   Statement  Status
00065280                05337708  +0000011C  __ctrace      05337708  +0000011C              Call
000651E0                052005A8  +0000006C  main          052005A8  +0000006C              Call
000650C8                0533FA26  +000000B4  @@MNINV       0533FA26  +000000B4              Call
00065018  CEEBBEXT      000079D8  +0000013C  CEEBBEXT      000079D8  +0000013C              Call
```

Output for C:

```
CEE3DMP: Sample ctrace output                                Language Environment for MVS
06/16/95 6:12:47 PMPage:    1

Information for enclave ????????

Information for thread 8000000000000000

Traceback:
  DSA Addr  Program Unit  PU Addr   PU Offset  Entry         E Addr    E  Offset   Statement  Status
  00065268                05337708  +0000011C  __ctrace      05337708  +0000011C              Call
  000651E0                052006B8  +0000005E  main          052006B8  +0000005E              Call
  000650C8                0533FA26  +000000B4  @@MNINV       0533FA26  +000000B4              Call
  00065018  CEEBBEXT      000079D8  +0000013C  CEEBBEXT      000079D8  +0000013C              Call
```

Related information

- *z/OS Language Environment Programming Guide, SA38-0682*
- “ctest.h” on page 21
- “cdump() — Request a main storage dump” on page 247
- “csnap() — Request a condensed dump” on page 351

cuserid() — Return character login of the user

Standards

Standards / Extensions	C or C++	Dependencies
XPG4 XPG4.2	both	

Format

```
#define _XOPEN_SOURCE
#include <stdio.h>

char *cuserid(char *s);
```

General description

The cuserid() function generates a character representation of the name associated with the real or effective user ID of the process.

If *s* is a NULL pointer, this representation is generated in an area that may be overwritten by subsequent calls to cuserid(). A pointer to the area is returned. If *s* is not a NULL pointer, *s* is assumed to point to an array of at least {L_cuserid} bytes; the representation is deposited in this array. The symbolic constant {L_cuserid} is defined in <stdio.h> and has a value greater than 0.

Note:

This function and constant L_cuserid are kept for historical reasons. They were part of the Legacy Feature in Single UNIX Specification, Version 2, but have been withdrawn and are not supported as part of Single UNIX Specification, Version 3. New applications should use getpwuid() instead of cuserid().

If it is necessary to continue using this function in an application written for Single UNIX Specification, Version 3, define the feature test macro _UNIX03_WITHDRAWN before including any standard system headers. The macro exposes all interfaces and symbols removed in Single UNIX Specification, Version 3.

Returned value

If *s* is not a NULL pointer, cuserid() returns *s*.

If *s* is not a NULL pointer and the login name cannot be found, the NULL byte '\0' will be placed at **s*.

If *s* is a NULL pointer and the login name cannot be found, cuserid() returns a NULL pointer.

If *s* is a NULL pointer and the login name can be found, cuserid() returns the address of a buffer local to the calling thread containing the login name.

Related information

- "stdio.h" on page 68
- "geteuid() — Get the effective user ID" on page 707
- "getlogin() — Get the user login name" on page 731
- "getpwnam() — Access the user database by user name" on page 760
- "getpwuid() — Access the user database by user ID" on page 763
- "getuid() — Get the real user ID" on page 791

dbm_clearerr() — Clear database error indicator

Standards

Standards / Extensions	C or C++	Dependencies
XPG4.2 Single UNIX Specification, Version 3	both	

Format

```
#define _XOPEN_SOURCE_EXTENDED 1
#include <ndbm.h>

int dbm_clearerr(DBM *db);
```

General description

The dbm_clearerr() function clears the error condition of the database. The argument *db* is a handle to a database previously obtained by dbm_open(). Note that this does not correct any problems with the database due to previous failures. It simply allows dbm_ operations to proceed. The database may be in an inconsistent or damaged state.

Special behavior for z/OS UNIX Services: In a multithreaded environment, the database error indicator is global to all threads using the database handle. Thus, clearing the error indicator affects all threads using the database handle.

Returned value

The return value is unspecified by X/Open.

If successful, dbm_clearerr() returns 0.

If unsuccessful, dbm_clearerr() returns -1 and sets errno to one of the following values:

Error Code
Description

EINVAL
Non-valid database descriptor specified.

Related information

- “ndbm.h” on page 49
- “dbm_close() — Close a database”
- “dbm_delete() — Delete database record” on page 367
- “dbm_error() — Check database error indicator” on page 368
- “dbm_fetch() — Get database content” on page 369
- “dbm_firstkey() — Get first key in database” on page 370
- “dbm_nextkey() — Get next key in database” on page 371
- “dbm_open() — Open a database” on page 372
- “dbm_store() — Store database record” on page 373

dbm_close() — Close a database

Standards

Standards / Extensions	C or C++	Dependencies
XPG4.2 Single UNIX Specification, Version 3	both	

Format

```
#define _XOPEN_SOURCE_EXTENDED 1
#include <ndbm.h>

void dbm_close(DBM *db);
```

General description

The dbm_close() function closes a database. The *db* argument is the database handle returned by a previous call to dbm_open().

Special behavior for z/OS UNIX Services: A dbm_close() function call removes access to the specified database handle to all threads within the process.

Returned value

dbm_close() returns no values.

Related information

- “ndbm.h” on page 49
- “dbm_clearerr() — Clear database error indicator” on page 365
- “dbm_delete() — Delete database record”
- “dbm_error() — Check database error indicator” on page 368
- “dbm_fetch() — Get database content” on page 369
- “dbm_firstkey() — Get first key in database” on page 370
- “dbm_nextkey() — Get next key in database” on page 371
- “dbm_open() — Open a database” on page 372
- “dbm_store() — Store database record” on page 373

dbm_delete() — Delete database record

Standards

Standards / Extensions	C or C++	Dependencies
XPG4.2 Single UNIX Specification, Version 3	both	

Format

```
#define _XOPEN_SOURCE_EXTENDED 1
#include <ndbm.h>

int dbm_delete(DBM *db, datum key);
```

General description

The dbm_delete() function deletes a record and its key from the database. The *db* argument specifies the database handle returned by a previous call to dbm_open(). The *key* argument identifies the record the program is deleting. The *key* datum must contain a `dptr` pointer to the key, and the key length in `dsize`.

After calling dbm_delete(), during a pass through the keys by dbm_firstkey() and dbm_nextkey(), the application positioning must be reset by calling dbm_firstkey(). If not, unpredictable results may occur including retrieval of the same key multiple times, or not at all.

File space is not physically reclaimed by a dbm_delete() operation. That is, the file size is not reduced. However, the space is available for reuse, subject to hashing.

Special behavior for z/OS UNIX Services: In a multithreaded environment, changes made to the database by a dbm_delete() operation affect all threads using the database handle. Thus, all other threads must also reset their positioning by using the dbm_firstkey() function before using dbm_nextkey(). A previously executed dbm_fetch() operation by **another** thread for the same *key* still has correct buffer pointers to the previous data. The dbm_delete() operation does not affect this. All other operations on other threads, such as dbm_fetch() to this (now) deleted *key* will fail.

Returned value

If successful, dbm_delete() returns 0.

If unsuccessful, dbm_delete() returns -1 and sets the error value in errno. Also, the database error indicator may be set.

Related information

- "ndbm.h" on page 49
- "dbm_clearerr() — Clear database error indicator" on page 365
- "dbm_close() — Close a database" on page 366
- "dbm_error() — Check database error indicator"
- "dbm_fetch() — Get database content" on page 369
- "dbm_firstkey() — Get first key in database" on page 370
- "dbm_nextkey() — Get next key in database" on page 371
- "dbm_open() — Open a database" on page 372
- "dbm_store() — Store database record" on page 373

dbm_error() — Check database error indicator

Standards

Standards / Extensions	C or C++	Dependencies
XPG4.2 Single UNIX Specification, Version 3	both	

Format

```
#define _XOPEN_SOURCE_EXTENDED 1
#include <ndbm.h>

int dbm_error(DBM *db);
```

General description

The dbm_error() function returns the error condition of the database. The argument *db* is a handle to a database previously obtained by dbm_open().

Special behavior for z/OS UNIX Services: In a multithreaded environment, the database error indicator is global to all threads using the database handle. Thus, the database error indicator may be set as a result of a database operation by another thread.

Returned value

dbm_error() returns 0 if the error condition is not set.

dbm_error() returns nonzero if the error condition is set.

Related information

- “ndbm.h” on page 49
- “dbm_clearerr() — Clear database error indicator” on page 365
- “dbm_close() — Close a database” on page 366
- “dbm_delete() — Delete database record” on page 367
- “dbm_fetch() — Get database content”
- “dbm_firstkey() — Get first key in database” on page 370
- “dbm_nextkey() — Get next key in database” on page 371
- “dbm_open() — Open a database” on page 372
- “dbm_store() — Store database record” on page 373

dbm_fetch() — Get database content

Standards

Standards / Extensions	C or C++	Dependencies
XPG4.2 Single UNIX Specification, Version 3	both	

Format

```
#define _XOPEN_SOURCE_EXTENDED 1
#include <ndbm.h>

datum dbm_fetch(DBM *db, datum key);
```

General description

The dbm_fetch() function reads a record from the database. The argument *db* is a handle to a database previously obtained by dbm_open(). The argument *key* is a datum that has been initialized by the application program to the value of the key that matches the key of the record the program is fetching. A `datum` is a structure that consists of two members, `dptr` and `dsize`. The member `dptr` is a char pointer to an array of data that is `dsize` bytes in length. (**Note:** The data is arbitrary binary data and is not NULL-terminated.)

The `dptr` is valid only until the next `dbm_` operation by this thread.

Special behavior for z/OS UNIX Services: In a multithreaded environment, the dbm_fetch() function returns a `dptr` in the datum structure to a data area that is thread-specific. This data area is not affected by other threads operations on the database, with the exception of a dbm_close() operation, which invalidates the *datum*.

Returned value

If successful, dbm_fetch() returns the `datum` containing a pointer to the data content `dptr`, and the data length `dsize`.

If unsuccessful, dbm_fetch() returns a NULL pointer in `dptr` and returns the error value in errno. Also, the database error indicator may be set.

Related information

- “ndbm.h” on page 49
- “dbm_clearerr() — Clear database error indicator” on page 365
- “dbm_close() — Close a database” on page 366

- "dbm_delete() — Delete database record" on page 367
- "dbm_error() — Check database error indicator" on page 368
- "dbm_firstkey() — Get first key in database"
- "dbm_nextkey() — Get next key in database" on page 371
- "dbm_open() — Open a database" on page 372
- "dbm_store() — Store database record" on page 373

dbm_firstkey() — Get first key in database

Standards

Standards / Extensions	C or C++	Dependencies
XPG4.2 Single UNIX Specification, Version 3	both	

Format

```
#define _XOPEN_SOURCE_EXTENDED 1
#include <ndbm.h>

datum dbm_firstkey(DBM *db);
```

General description

The dbm_firstkey() function returns the first key in the database. The argument *db* is a handle to a database previously obtained by dbm_open(). Since the keys are arbitrary binary data, the order of key return by dbm_firstkey() and dbm_nextkey() does not reflect any lexical ordering. In addition, the return order does not reflect record insertion ordering. All keys can be retrieved from the database by executing a loop such as:

```
for (key = dbm_firstkey(db); key.dptr !=NULL; key = dbm_nextkey(db))
```

That is, establish positioning to the beginning by use of the dbm_firstkey() function, then loop doing dbm_nextkey() function calls until a NULL `dptr` is returned in the *datum*.

The returned `dptr` is valid only until the next `dbm_` operation by this thread.

Special behavior for z/OS UNIX Services: In a multithreaded environment, the dbm_firstkey() function returns a pointer to data that is thread-specific. In addition, each thread maintains its own positioning information for dbm_nextkey() operations. However, other threads making modifications to the database, for example using dbm_store() or dbm_delete() can cause unpredictable results for threads executing dbm_nextkey(), including keys retrieved multiple times or not at all. The application must reset positioning to the beginning using dbm_firstkey() if another thread has done a modification to the database.

Returned value

If successful, dbm_firstkey() returns the `datum` containing a pointer to the key `dptr`, and the key length `dsize`.

If unsuccessful, dbm_firstkey() returns a NULL pointer in `dptr` and returns the error value in errno. Also, the database error indicator may be set.

Related information

- “ndbm.h” on page 49
- “dbm_clearerr() — Clear database error indicator” on page 365
- “dbm_close() — Close a database” on page 366
- “dbm_delete() — Delete database record” on page 367
- “dbm_error() — Check database error indicator” on page 368
- “dbm_fetch() — Get database content” on page 369
- “dbm_nextkey() — Get next key in database”
- “dbm_open() — Open a database” on page 372
- “dbm_store() — Store database record” on page 373

dbm_nextkey() — Get next key in database

Standards

Standards / Extensions	C or C++	Dependencies
XPG4.2 Single UNIX Specification, Version 3	both	

Format

```
#define _XOPEN_SOURCE_EXTENDED 1
#include <ndbm.h>

datum dbm_nextkey(DBM * db);
```

General description

The dbm_nextkey() function returns the next key in the database. The argument *db* is a handle to a database previously obtained by dbm_open(). Since the keys are arbitrary binary data, the order of key return by dbm_firstkey() and dbm_nextkey() does not reflect any lexical ordering. In addition, the return order does not reflect record insertion ordering. All keys can be retrieved from the database by executing a loop such as:

```
for (key = dbm_firstkey(db); key.dptr !=NULL; key = dbm_nextkey(db))
```

That is, establish positioning to the beginning by use of the dbm_firstkey() function, then loop doing dbm_nextkey() function calls until a NULL `dptr` is returned in *datum*.

The returned `dptr` is valid only until the next `dbm_` operation by this thread.

Special behavior for z/OS UNIX Services: In a multithreaded environment, the dbm_nextkey() function returns a pointer to data that is thread-specific. In addition, each thread maintains its own positioning information for dbm_nextkey() operations. However, other threads making modifications to the database, for example using dbm_store() or dbm_delete() can cause unpredictable results for threads executing dbm_nextkey(), including keys retrieved multiple times or not at all. The application must reset positioning to the beginning using dbm_firstkey() if another thread has done a modification to the database.

Returned value

If successful, dbm_nextkey() returns the `datum` containing a pointer to the key `dptr`, and the key length `dsize`.

If unsuccessful, dbm_nextkey() returns a NULL pointer in dptr and returns the error value in errno. Also, the database error indicator may be set.

Related information

- "ndbm.h" on page 49
- "dbm_clearerr() — Clear database error indicator" on page 365
- "dbm_close() — Close a database" on page 366
- "dbm_delete() — Delete database record" on page 367
- "dbm_error() — Check database error indicator" on page 368
- "dbm_fetch() — Get database content" on page 369
- "dbm_firstkey() — Get first key in database" on page 370
- "dbm_open() — Open a database"
- "dbm_store() — Store database record" on page 373

dbm_open() — Open a database

Standards

Standards / Extensions	C or C++	Dependencies
XPG4.2 Single UNIX Specification, Version 3	both	

Format

```
#define _XOPEN_SOURCE_EXTENDED 1
#include <ndbm.h>

DBM *dbm_open(const char *file, int open_flags, mode_t file_mode);
```

General description

The dbm_open() function opens a database. The *file* argument is the path name of the database, not including the filename suffix (the part after the .). The database is stored in two files. One file is a directory containing a bit map of blocks in use and has .dir as its suffix. The second file contains all the data and has .pag as its suffix. The *open_flags* argument has the same meaning as the *flags* argument of open() except that a database opened for write-only access opens the files for read and write access. The *file_mode* argument has the same meaning as the third argument of open().

The number of records that can be stored in the database is limited by the file space available for the .dir and .pag files, and by the underlying key hashing. If multiple keys hash to the same 32 bit hash value, the number of keys for that hash value is limited to the amount of data (key sizes plus content sizes plus overhead) that can be stored in a single logical block of 1024 bytes.

Special behavior for z/OS UNIX Services: In a multithreaded environment, the dbm_ functions have both POSIX process wide and thread-specific characteristics. z/OS UNIX services provide the following multithreaded behavior:

1. A database handle returned by the dbm_open() function is a process wide resource. This means that multiple threads within the process can access the database using the same database handle.
2. Each thread using a given database handle has its own positioning information for dbm_firstkey() and dbm_nextkey() operations. This means that multiple threads can each be executing a dbm_nextkey() loop.

3. Each thread using a given database handle has its own buffering for dbm_fetch() operations. This means that a pointer to a keys content (as returned by dbm_fetch()) remains valid, even if other threads modify the database.
4. Database modifications are automatically reflected to all of the threads using the same database handle. For example, if a thread adds a key/data pair using dbm_store(), a dbm_fetch() of that key by another thread will be successful.
5. Operations which modify the database, such as dbm_store() and dbm_delete(), can cause unpredictable results to threads executing dbm_nextkey(). If a database modification is done, all threads should reset positioning using a dbm_firstkey() call before executing dbm_nextkey().
6. A dbm_close() operation removes access to the database for all threads that use the database handle.
7. Multiple dbm_open() operations, whether by a single thread, multiple threads within a process, or by multiple processes are permitted, but for read access only. No protection is provided for database modification, and modification can result in unpredictable results, including database destruction.

Returned value

If successful, dbm_open() returns a pointer to the database descriptor.

If unsuccessful, dbm_open() returns a NULL pointer and stores the error value in errno.

Related information

- “ndbm.h” on page 49
- “dbm_clearerr() — Clear database error indicator” on page 365
- “dbm_close() — Close a database” on page 366
- “dbm_delete() — Delete database record” on page 367
- “dbm_error() — Check database error indicator” on page 368
- “dbm_fetch() — Get database content” on page 369
- “dbm_firstkey() — Get first key in database” on page 370
- “dbm_nextkey() — Get next key in database” on page 371
- “dbm_store() — Store database record”
- “open() — Open a file” on page 1152

dbm_store() — Store database record

Standards

Standards / Extensions	C or C++	Dependencies
XPG4.2 Single UNIX Specification, Version 3	both	

Format

```
#define _XOPEN_SOURCE_EXTENDED 1
#include <ndbm.h>

int dbm_store(DBM *db, datum key, datum content, int store_mode);
```

General description

The dbm_store() function writes a record to a database. The *db* argument specifies the database handle returned by a previous call to dbm_open(). The *key* argument identifies the record the program is deleting. The *key* datum must contain a `dptr` pointer to the key, and the key length in `dsize`. The argument *content* is a datum. that describes the data record being stored. record the program is writing. The *content* datum. contains a `dptr` pointer to the data, and the data length in `dsize`.

The argument *store_mode* controls whether dbm_store() replaces a already existing record that has the same key. The *store_mode* argument may be any one of the following set of symbols defined in the <ndbm.h> include file:

DBM_INSERT
Do not add the *key* and *content* pair if the *key* already exists in the database. If the *key* doesn't already exist, add the new *key* and *content* pair.

DBM_REPLACE
Replace the *key* and *content* pair in the database with the new pair if the *key* already exists. If the *key* doesn't already exist, add the new *key* and *content* pair.

After calling dbm_store(), during a pass through the keys by dbm_firstkey() and dbm_nextkey(), the application positioning must be reset by calling dbm_firstkey(). If not, unpredictable results may occur including retrieval of the same key multiple times, or not at all.

The number of records that can be stored in the database is limited by the file space available for the .dir and .pag files, and by the underlying key hashing. If multiple keys hash to the same 32 bit hash value, the number of keys for that hash value is limited to the amount of data (key sizes plus content sizes plus overhead) that can be stored in a single logical block of 1024 bytes.

Special behavior for z/OS UNIX Services: In a multithreaded environment, changes made to the database by a dbm_store() operation affect all threads using the database handle. Thus, all other threads must also reset their positioning by using the dbm_firstkey() function before using dbm_nextkey(). A previously executed dbm_fetch() operation by **another** thread for the same *key* still has correct buffer pointers to the previous data. The dbm_store() operation does not affect this. All other operations, such as dbm_fetch() or dbm_delete(), will automatically have access to the new *key* and *content* pair.

Returned value

If successful, dbm_store() returns 0. If `DBM_INSERT` is specified, and the *key* already exists, dbm_store() returns 1.

If unsuccessful, dbm_store() returns -1 and sets errno to one of the following values. Also, the database error indicator may be set.

Error Code
Description

EFBIG
Seek/Write operation failed attempting to write new block. This errno is not part of the errno set described by X/Open for this function. You may be able to store other *key* and *content* pairs when the *key* hashes to a different value.

ENOSPC

key plus *content* plus block overhead does not fit into a block. This errno is not part of the errno set described by X/Open for this function. The *key* plus *content* underlying data lengths need be less or equal to 1012 bytes in length.

Related information

- "ndbm.h" on page 49
- "dbm_clearerr() — Clear database error indicator" on page 365
- "dbm_close() — Close a database" on page 366
- "dbm_delete() — Delete database record" on page 367
- "dbm_error() — Check database error indicator" on page 368
- "dbm_fetch() — Get database content" on page 369
- "dbm_firstkey() — Get first key in database" on page 370
- "dbm_nextkey() — Get next key in database" on page 371
- "dbm_open() — Open a database" on page 372

decabs() — Decimal absolute value

Standards

Standards / Extensions	C or C++	Dependencies
C Library	C only	

Format

```
#include <decimal.h>

decimal(n,p) decabs(decimal(n,p) pdec);
```

General description

The built-in function decabs() accepts a decimal type expression as an argument and returns the absolute value of the decimal argument, in the same decimal type as the argument. The function does not change the content of the argument.

The parameter *n* can be any integral value between 1 and DEC_DIG. The parameter *p* can be any integral value between 0 and DEC_PRECISION, although it must be less than or equal to *n*. DEC_DIG and DEC_PRECISION are defined inside decimal.h.

If the content of the given argument is not in native packed decimal format, behavior is undefined.

Example

CELEBD01

```
/* CELEBD01 */
#include <decimal.h>

decimal(10,2) p1, p2;
int main(void) {
    p2 = -1234.56d;
    p1 = decabs(p2);
    printf("p1 = %D(10,2), p2 = %D(10,2)\n", p1, p2);
    return(0);
}
```

Output

```
p1 = 1234.56, p2 = -1234.56
```

Related information

- "decimal.h" on page 22
- "decchk() — Check for valid decimal types"
- "decfix() — Fix up a nonpreferred sign variable" on page 377

decchk() — Check for valid decimal types

Standards

Standards / Extensions	C or C++	Dependencies
C Library	C only	

Format

```
#include <decimal.h>

int decchk(decimal(n,p) pdec);
```

General description

The built-in function decchk() accepts a decimal type expression as an argument and returns a status value of type `int`.

The status can be interpreted as follows:

DEC_VALUE_OK
A valid decimal representation value (including the less-preferred but valid sign, A-F).

DEC_BAD_NIBBLE
The leftmost half-byte is not 0 in a decimal type number that has an even number of digits. For example, 123 is stored in decimal(2,0). If such a number is packed, then it is used.

DEC_BAD_DIGIT
Digits not allowed (not 0-9). If such a number is packed, then it is used.

DEC_BAD_SIGN
Sign not allowed (not A-F). If such a number is packed, then it is used.

The function return status can be masked to return multiple status.

The parameter *n* can be any integral value between 1 and DEC_DIG. The parameter *p* can be any integral value between 0 and DEC_PRECISION, although it must be less than or equal to *n*. DEC_DIG and DEC_PRECISION are defined inside decimal.h.

If the content of the given argument is not in native packed decimal format, the behavior is undefined.

Example

```
#include <decimal.h>

decimal(10,2) p1;
char mem2[3] = { 0x12, 0x34, 0x5c };   /* bad half-byte */
char mem3[3] = { 0x02, 0xa4, 0x5c };   /* bad digit */
char mem4[3] = { 0x02, 0x34, 0x56 };   /* bad sign */
char mem5[3] = { 0x12, 0xa4, 0x56 };   /* bad half-byte, digit and sign */
decimal(4,0) *pp2;
decimal(4,0) *pp3;
decimal(4,0) *pp4;
decimal(4,0) *pp5;
int main(void) {
    p1 = 123456.78d;
    pp2 = (decimal(4,0) *) mem2;
    pp3 = (decimal(4,0) *) mem3;
    pp4 = (decimal(4,0) *) mem4;
    pp5 = (decimal(4,0) *) mem5;

    if (decchk(p1) == DEC_VALUE_OK) {
        printf("p1 is a valid decimal representation value.\n");
    }
    if (decchk(*pp2) == DEC_BAD_NIBBLE) {
        printf("pp2 points to a bad half-byte value!\n");
    }
    if (decchk(*pp3) == DEC_BAD_DIGIT) {
        printf("pp3 points to an illegal digit!\n");
    }
    if (decchk(*pp4) == DEC_BAD_SIGN) {
        printf("pp4 points to an illegal sign!\n");
    }
    /* The wrong way ----- */
    if (decchk(*pp5) == DEC_BAD_SIGN) {
        printf("YOU SHOULD NOT GET THIS!!!!!\n");
    }
    /* The right way ----- */
    if ((decchk(*pp5) & DEC_BAD_SIGN) == DEC_BAD_SIGN) {
        printf("pp5 points to an illegal sign!\n");
    }
    return(0);
}
```

Output

```
p1 is a valid decimal representation value.
pp2 points to a bad half-byte value!
pp3 points to an illegal digit!
pp4 points to an illegal sign!
pp5 points to an illegal sign!
```

Related information

- “decimal.h” on page 22
- “decabs() — Decimal absolute value” on page 375
- “decfix() — Fix up a nonpreferred sign variable”

decfix() — Fix up a nonpreferred sign variable

Standards

Standards / Extensions	C or C++	Dependencies
C Library	C only	

Format

```
#include <decimal.h>

decimal(n,p) decfix(decimal(n,p) pdec);
```

General description

The built-in function decfix() accepts a decimal type expression as an argument and returns a decimal value that has the same type and same value as the argument with the correct preferred sign. The function does not change the content of the argument.

The parameter *n* can be any integral value between 1 and DEC_DIG. The parameter *p* can be any integral value between 0 and DEC_PRECISION, though it must be less than or equal to *n*. DEC_DIG and DEC_PRECISION are defined inside decimal.h.

If the content of the given argument is not in native packed decimal format, behavior is undefined.

Example

```
#include <decimal.h>

char *ptr;
char mem[3] = { 0x01, 0x23, 0x4A };
decimal(4,0) *pp;
decimal(4,0) p;
int main(void) {
    pp = (decimal(4,0) *) mem;
    p = decfix(*pp);
    ptr = (char *) p;
    printf("Before decfix : %X%X%X\n", mem[0], mem[1], mem[2]);
    printf("After  decfix : %X%X%X\n", ptr[0], ptr[1], ptr[2]);
    return(0);
}
```

Output

```
Before decfix : 1234A
After  decfix : 1234C
```

Related information

- "decimal.h" on page 22
- "decabs() — Decimal absolute value" on page 375
- "decchk() — Check for valid decimal types" on page 376

DeleteWorkUnit() — Delete a WLM work unit

Standards

Standards / Extensions	C or C++	Dependencies
z/OS UNIX	both	

Format

```
#include <sys/__wlm.h>

int DeleteWorkUnit(wlmetok_t *enclavetoken);
```

General description

The DeleteWorkUnit() function provides the ability for an application to delete a WLM work unit.

******enclavetoken*

Points to a work unit enclave token that was returned from a call to CreateWorkUnit() or ContinueWorkUnit().

Returned value

If successful, DeleteWorkUnit() returns 0.

If unsuccessful, DeleteWorkUnit() returns -1 and sets errno to one of the following values:

Error Code
Description

EFAULT
An argument of this function contained an address that was not accessible to the caller.

EINVAL
An argument of this function contained an incorrect value.

EMVSSAF2ERR
An error occurred in the security product.

EMVSWLMERROR
The WLM delete enclave failed. Use __errno2() to obtain the WLM service reason code for the failure.

EPERM
The calling thread's address space is not permitted to the BPX.WLMSERVER Facility class. The caller's address space must be permitted to the BPX.WLMSERVER Facility class if it is defined. If BPX.WLMSERVER is not defined, the calling process is not defined as a superuser (UID=0).

Related information

- “sys/__wlm.h” on page 77
- “CheckSchEnv() — Check WLM scheduling environment” on page 271
- “ConnectServer() — Connect to WLM as a server manager” on page 313
- “ConnectWorkMgr() — Connect to WLM as a work manager” on page 315
- “ContinueWorkUnit() — Continue WLM work unit” on page 323
- “DisconnectServer() — Disconnect from WLM server” on page 383
- “ExtractWorkUnit() — Extract enclave service” on page 457
- “JoinWorkUnit() — Join a WLM work unit” on page 927
- “LeaveWorkUnit() — Leave a WLM work unit” on page 945
- “QueryMetrics() — Query WLM system information” on page 1368
- “QuerySchEnv() — Query WLM scheduling environment” on page 1369

difftime(), difftime64() — Compute time difference

Standards

Standards / Extensions	C or C++	Dependencies
ISO C XPG4 XPG4.2 C99 Single UNIX Specification, Version 3 Language Environment	both	

Format

Format of the difftime() function:

```
#include <time.h>

double difftime(time64_t time2, time_t time1);
```

Format of the difftime64() function:

```
#define _LARGE_TIME_API
#include <time.h>

double difftime64(time64_t time2, time64_t time1);
```

General description

Computes the difference in seconds between *time2* and *time1*, which are calendar times returned by time().

The difftime() function returns the difference between two calendar times as a double. The return value is hexadecimal floating-point or IEEE Binary Floating-Point format depending on the floating-point mode of the thread invoking difftime(). The difftime() function uses __isBFP() to determine which floating-point format (hexadecimal floating-point or IEEE Binary Floating-Point) to return on the invoking thread.

The function difftime64() will behave exactly like difftime() except it will support calendar times beyond 03:14:07 UTC on January 19, 2038 with a limit of 23:59:59 UTC on December 31, 9999.

Returned value

Returns the elapsed time in seconds from *time1* to *time2* as a double.

Example

CELEBD04

```
/* CELEBD04

   This example shows a timing application using &diff..
   The example calculates how long, on average, it takes a
   user to input some data to the program.

 */
#include <time.h>
#include <stdio.h>

int main(void)
{
   time_t start, finish;
   int i, n, num;
   int answer;

   printf("11 x 55 = ? Enter your answer below\n");
   time(&start);
   scanf("%d",&answer);
   time(&finish);
   printf("You answered %s in %.0f seconds.\n",
           answer == 605 ? "correctly" : "incorrectly",
           difftime(finish,start));
}
```

Output

```
11 x 55 = ? Enter your answer below
605
You answered correctly in 20 seconds
```

Related information

- "time.h" on page 79
- "asctime(), asctime64() — Convert time to character string" on page 182
- "asctime_r(), asctime64_r() — Convert date and time to a character string" on page 184
- "ctime(), ctime64() — Convert time to character string" on page 359
- "ctime_r(), ctime64_r() — Convert time value to date and time character string" on page 362
- "gmtime(), gmtime64() — Convert time to broken-down UTC time" on page 810
- "gmtime_r(), gmtime64_r() — Convert a time value to broken-down UTC time" on page 812
- "__isBFP() — Determine application floating-point format" on page 904
- "localtime(), localtime64() — Convert time and correct for local time" on page 984
- "localtime_r(), localtime64_r() — Convert time value to broken-down local time" on page 986
- "mktime(), mktime64() — Convert local time" on page 1078
- "strftime() — Convert to formatted time" on page 1743
- "time(),time64() — Determine current UTC time" on page 1878

dirname() — Report the parent directory of a path name

Standards

Standards / Extensions	C or C++	Dependencies
XPG4.2 Single UNIX Specification, Version 3	both	

Format

```
#define _XOPEN_SOURCE_EXTENDED 1
#include <libgen.h>

char *dirname(char *path);
```

General description

The dirname() function takes a pointer to a character string that contains a path name, and returns a pointer to a string that is a path name of the parent directory of that file. Trailing '/' characters in the path are not counted as part of the path.

If *path* does not contain a '/' then dirname() returns a pointer to the string ".". If *path* is a NULL pointer or points to an empty string, dirname() returns a pointer to the string ".".

The dirname() function may modify the string pointed to by path.

Examples:

```
Input String      Output String
"/usr/lib"        "/usr"
"/usr/"           "/"
"usr"             "."
"/"               "/"
"."               "."
".."              "."
```

Returned value

If successful, dirname() returns a pointer to a string that is the parent directory of *path*.

If *path* is a NULL pointer or points to an empty string, dirname() returns a pointer to a string ".".

There are no errno values defined.

Related information

- "libgen.h" on page 38
- "basename() — Return the last component of a path name" on page 208

__discarddata() — Release pages backing virtual storage

Standards

Standards / Extensions	C or C++	Dependencies
Language Environment	both	OS/390 V2R10

Format

```
#include <stdlib.h>

int __discarddata(void *addr, size_t size);
```

General description

The __discarddata() function is used to release segments of real storage backing virtual storage. Segments backing virtual storage are released beginning at location *addr* for a length of *size*. For AMODE 31, the *addr* must begin on a page (4K) boundary and size must be a multiple of 4K. For AMODE 64, the *addr* must begin on a segment (1 MB) boundary and size must be a multiple of 1 MB.

Returned value

If successful, __discarddata() returns 0.

If unsuccessful, because *addr* does not begin on a page (4K) boundary or *size* is not a multiple of 4K, __discarddata() returns -1.

There are no errno values defined.

Related information

- "stdlib.h" on page 70

DisconnectServer() — Disconnect from WLM server

Standards

Standards / Extensions	C or C++	Dependencies
z/OS UNIX	both	

Format

```
#include <sys/__wlm.h>

int DisconnectServer(unsigned long *conn_tkn);
```

AMODE 64

```
#include <sys/__wlm.h>

int DisconnectServer(unsigned int *conn_tkn);
```

General description

The DisconnectServer function provides the ability for an application to disconnect from WLM.

**conn_tkn*
Specifies the connect token that represents the WLM connection that is to be disconnected.

Returned value

If successful, DisconnectServer() returns 0.

If unsuccessful, DisconnectServer() returns -1 and sets errno to one of the following values:

Error Code
Description

EFAULT
An argument of this function contained an address that was not accessible to the caller.

EINVAL
An argument of this function contained an incorrect value.

EMVSSAF2ERR
An error occurred in the security product.

EMVSWLMERROR
The WLM disconnect failed. Use __errno2() to obtain the WLM service reason code for the failure.

EPERM
The calling thread's address space is not permitted to the BPX.WLMSERVER Facility class. The caller's address space must be permitted to the BPX.WLMSERVER Facility class if it is defined. If BPX.WLMSERVER is not defined, the calling process is not defined as a superuser (UID=0).

Related information

- “sys/__wlm.h” on page 77
- “CheckSchEnv() — Check WLM scheduling environment” on page 271
- “ConnectServer() — Connect to WLM as a server manager” on page 313
- “ConnectWorkMgr() — Connect to WLM as a work manager” on page 315
- “ContinueWorkUnit() — Continue WLM work unit” on page 323
- “CreateWorkUnit() — Create WLM work unit” on page 344
- “DeleteWorkUnit() — Delete a WLM work unit” on page 378
- “ExtractWorkUnit() — Extract enclave service” on page 457
- “JoinWorkUnit() — Join a WLM work unit” on page 927
- “LeaveWorkUnit() — Leave a WLM work unit” on page 945
- “QueryMetrics() — Query WLM system information” on page 1368
- “QuerySchEnv() — Query WLM scheduling environment” on page 1369

div() — Calculate quotient and remainder

Standards

Standards / Extensions	C or C++	Dependencies
ISO C XPG4 XPG4.2 ISO/ANSI C++ C99 Single UNIX Specification, Version 3 C++ TR1 C99	both	

Format

```
#include <stdlib.h>

div_t div(int numerator, int denominator);
div_t div(long numerator, long denominator);   /* C++ only */
```

C++ TR1 C99:

```
#define _TR1_C99
#include <inttypes.h>
imaxdiv_t div(intmax_t numerator, intmax_t denominator);

#define _TR1_C99
#include <stdlib.h>
lldiv_t div(long long numerator, long long denominator);
```

General description

Calculates the quotient and remainder of the division of *numerator* by *denominator*.

Special behavior for C++: For C++ applications, div() is also overloaded for the type long.

Returned value

Returns a structure of type `div_t`, containing both the quotient `int` *quot* and the remainder `int` *rem*. This structure is defined in stdlib.h. If the returned value cannot be represented, the behavior of div() is undefined. If *denominator* is 0, the same exception will be raised as if you divided by 0. That is, you get the error CEE3209S (Fixed-point divide exception).

Related information

- "stdlib.h" on page 70
- "ldiv() — Compute quotient and remainder of integral division" on page 944

dlclose() — Close a dlopen() object

Standards

Standards / Extensions	C or C++	Dependencies
Single UNIX Specification, Version 3	both	z/OS V1R6

Format

```
#define _UNIX03_SOURCE
#include <dlfcn.h>

int  dlclose(void *handle);
```

General description

Informs the system that the dynamic link library (DLL) referenced by a *handle* returned from a previous dlopen() invocation is no longer needed by the application. Once a DLL has been closed, an application should assume that its symbols and the symbols of any dependent DLLs are no longer available to dlsym().

Returned value

NULL is returned if the referenced DLL was successfully closed. If the DLL could not be closed, or if *handle* does not refer to an open DLL, a non-zero value will be returned.

Usage notes

1. A conforming application should use a *handle* returned from a dlopen() invocation only within a given scope, bracketed by the dlopen() and dlclose() operations. The value of a *handle* must be treated as an opaque object by the application, used only in calls to dlsym() and dlclose().
2. DLLs that are loaded explicitly, that is with dlopen(), and are not freed with a corresponding call to dlclose(), are freed automatically at enclave termination in LIFO sequence.
3. Non-local C++ static destructors defined in a DLL are executed only once, when the DLL program object is deleted from memory.
4. More detailed diagnostic information is available through dlerror(), the _EDC_DLL_DIAG environment variable, and the Language Environment DLL Failure control block (CEEDLLF) chain.
5. This function is not available under SPC, MTF and CSP environments.

Example

The following example illustrates use of dlopen() and dlclose():

```
...
/* Open a dynamic library and then close it ... */

#include <dlfcn.h>

void *mylib;
```

```
int eret;

mylib = dlopen("mylib.so", RTLD_LOCAL | RTLD_LAZY);
...
eret = dlclose(mylib);
...
```

Related information

- "dlerror() — Get diagnostic information"
- "dlopen() — Gain access to a dynamic link library" on page 387
- "dlsym() — Obtain the address of a symbol from a dlopen() object" on page 390
- "dlfcn.h" on page 23

dlerror() — Get diagnostic information

Standards

Standards / Extensions	C or C++	Dependencies
Single UNIX Specification, Version 3	both	z/OS V1R6

Format

```
#define _UNIX03_SOURCE
#include <dlfcn.h>

void  dlerror(void);
```

General description

Returns a null-terminated character string (with no trailing <newline>) that describes the last error that occurred while processing a DLL by dlopen(), dlsym(), or dlclose(). NULL is returned if no errors have occurred since the last invocation of dlerror().

Note: dlerror() is thread safe, so the information returned describes the last error that occurred on that thread

Returned value

A null-terminated character string is returned if successful, otherwise NULL is returned.

Usage notes

1. Messages returned by dlerror() reside in a static buffer that is overwritten on each new call to dlerror() on that thread.
2. Application code should not write to this buffer.
3. Programs wishing to preserve an error message should make their own copies of that message.
4. This function is not available under SPC, MTF and CSP environments.

Example

The following example prints out the last dynamic linking error:

```
...
#include <dlfcn.h>

char *errstr;

errstr = dlerror();
if (errstr != NULL)
printf ("A dynamic linking error occurred: (%s)\n", errstr);
...
```

Related information

- “dlclose() — Close a dlopen() object” on page 385
- “dlopen() — Gain access to a dynamic link library”
- “dlsym() — Obtain the address of a symbol from a dlopen() object” on page 390
- “dlfcn.h” on page 23

dlopen() — Gain access to a dynamic link library

Standards

Standards / Extensions	C or C++	Dependencies
Single UNIX Specification, Version 3	both	z/OS V1R6

Format

```
#define _UNIX03_SOURCE
#include <dlfcn.h>

void  *dlopen(const char *file, int mode);
```

General description

Makes the dynamic link library (DLL) specified by *file* available to the calling program.

If the *file* argument contains a single slash ("/"), it is used as the z/OS UNIX file system path name for the DLL. If the environment variable LIBPATH is set, each directory listed will be searched for the DLL. Otherwise, the current directory will be searched.

Note: Searching for a DLL in the z/OS UNIX file system is case-sensitive.

If the *file* argument begins with two slashes ("//"), then an attempt is made to load the DLL from the caller's MVS load library search order (in order: STEPLIB/JOBLIB, LPA, Link List). The DLL name must be eight characters or less, and is converted to uppercase.

If the *file* argument doesn't begin with one or two slashes ("/" or "//"), and doesn't contain a single slash ("/") anywhere in the name, then it is ambiguous as to where the DLL resides.

- If the POSIX(ON) runtime option is specified, then the z/OS UNIX file system is searched first for the DLL, and if not found, the MVS load library is searched.
- If the POSIX(OFF) runtime option is specified, then the MVS load library is searched first for the DLL, and if not found, the z/OS UNIX file system is searched.

Under the CICS environment, the search sequence for DLL load modules is the same as that used for dynamically loaded CICS modules. Loading DLLs from the z/OS UNIX file system is not supported under CICS.

For more information about how DLLs are loaded and how the search sequence is used, see the topic about loading DLLs in *z/OS Language Environment Programming Guide*.

A successful call returns a handle which the caller may use on subsequent calls to dlsym() and dlclose(). The value of this handle should not be interpreted in any way by the caller.

Only a single copy of a DLL is brought into the address space, even if invoked multiple times for the same DLL, and even if different values of the *file* parameter are used to reference the same DLL.

The *mode* parameter describes how dlopen() operates on a file with respect to the processing of dependent DLLs and the scope of visibility of the symbols provided within *file*. If a *file* is specified in multiple invocations, *mode* is interpreted at each invocation. The *mode* is a bitwise-OR of the values specified.

Mode Values

When a DLL is loaded, it may contain implicit references to symbols in another "dependent" DLL, whose addresses are not known until that DLL is loaded. These implicit references must be relocated before the symbols can be accessed, which means loading the DLL containing the references. The *mode* parameter governs when these relocations (and loads) take place and may have the following values:

Value Description

RTLD_LAZY
: When possible, the loading of dependent DLLs, and resolution of symbols contained therein, may be deferred until the first reference to one of those symbols. This is the default behavior.

  Note: Once *RTLD_NOW* has been specified, all relocations will have been completed causing additional *RTLD_NOW* operations to be redundant and any further *RTLD_LAZY* operations irrelevant.

RTLD_NOW
: Load all dependent DLLs for the DLL being loaded and resolve all symbols before returning. This may include zero or more levels of nested dependent DLLs, all of which are loaded at this time.

RTLD_GLOBAL
: Allows symbols in the DLL being loaded to be visible when resolving symbols through the global symbol object that was opened with dlopen(*NULL*,*0*). All dependent DLLs are always implicitly loaded as if RTLD_GLOBAL had been specified. This is the default behavior.

RTLD_LOCAL
: Prevents symbols in the DLL being loaded to be visible when resolving symbols through the global symbol object that was opened with dlopen(*NULL*,*0*). All dependent DLLs of this DLL continue to be implicitly loaded as if RTLD_GLOBAL had been specified.

  If a subsequent call is made for this same DLL with a *mode* of *RTLD_GLOBAL*, then the DLL will maintain the *RTLD_GLOBAL* status regardless of any previous or future specification of *RTLD_LOCAL*, as long as the DLL remains loaded (see dlclose()).

If the value of *file* is NULL, dlopen() returns a "global symbol object" *handle*. This object will provide access (via dlsym()) to the symbols exported from:

- The main application, and dependent DLLs for the main application that were loaded at program start-up, and
- The set of DLLs loaded using dlopen() with the *RTLD_GLOBAL* flag. This set of DLLs can change dynamically as other DLLs are opened and closed.

Symbols introduced by the call to dlopen() for a DLL, and available through dlsym(), are those which are exported by the DLL. Typically such symbols will be those identified by a #pragma export in C, or with the EXPORTALL compile option. For details on how to specify exported data and functions, for C/C++ as well as other languages, see the topic about building a simple DLL in *z/OS Language Environment Programming Guide*

Returned value

NULL is returned if:

- *file* cannot be found or opened for reading
- *file* is not in correct DLL executable format
- an error occurred during the process of loading *file*, or relocating its symbolic references
-

Usage notes

1. For details on how to create and use DLLs, see *z/OS Language Environment Programming Guide*.
2. The AMODE of the application must be the same as the AMODE of the DLL.
3. Non-local C++ static constructors defined in a DLL are executed only once, when the DLL program object is physically loaded into memory.
4. More detailed diagnostic information is available through dlerror(), the _EDC_DLL_DIAG environment variable, and the Language Environment DLL Failure control block (CEEDLLF) chain.
5. This function is not available under SPC, MTF and CSP environments.

Example

The following example illustrates use of dlopen() and dlclose():

```
...
/* Open a dynamic library and then close it ... */

#include <dlfcn.h>

void *mylib;
int eret;

mylib = dlopen("mylib.so", RTLD_LOCAL | RTLD_LAZY);
...
eret = dlclose(mylib);
...
```

Related information

- "dlclose() — Close a dlopen() object" on page 385
- "dlerror() — Get diagnostic information" on page 386
- "dlsym() — Obtain the address of a symbol from a dlopen() object"
- "dlfcn.h" on page 23

dlsym() — Obtain the address of a symbol from a dlopen() object

Standards

Standards / Extensions	C or C++	Dependencies
Single UNIX Specification, Version 3	both	z/OS V1R6

Format

```
#define _UNIX03_SOURCE
#include <dlfcn.h>

void  *dlsym(void *__restrict__ handle, const char *__restrict__ name);
```

General description

Obtains the address of a symbol defined within a dynamic link library (DLL) made accessible through a dlopen() call. The *handle* argument is the value returned from a call to dlopen() (which has not been released by a call to dlclose()), and *name* is the symbol's name as a character string.

The DLL that was loaded by dlopen() will be searched for the named symbol. If the symbol is not found in that DLL, then the dependent DLLs of that DLL will be searched, followed by any dependents of those, and continuing in a breadth-first manner until the named symbol is found or all dependent DLLs have been searched. This search order determines how duplicate symbols in different DLLs will be found, although the order in which dependent DLLs at the same level are searched is indeterminate.

Also note that a search of dependent DLLs by dlsym() will not result in unloaded dependent DLLs being loaded. Only the dependent DLLs loaded as part of the call to dlopen() will be searched. If the full set of dependent DLLs need to be available to subsequent calls to dlsym(), make sure the DLL is opened with the *RTLD_NOW* load flag. It is indeterminate which dependent DLLs are loaded when *RTLD_LAZY* is specified

The only exception to this is the global symbol object obtained via a dlopen(*NULL*,*0*) call, in which case all DLLs (excluding those opened with *RTLD_LOCAL*) are searched in the order in which they were loaded.

Returned value

NULL is returned:

- If *handle* does not refer to a valid DLL opened by dlopen(),
- or the named symbol (*name*) cannot be found within any of the DLLs associated with *handle*.

Usage notes

1. The named symbol can be either an exported data item or function.
2. DLLs are enclave level resources. See *z/OS XL C/C++ Programming Guide* for more information about the use of DLLs in a multi-threaded environment.
3. C++ symbol names should be passed to dlsym() in mangled form; dlsym() does not perform any name mangling on behalf of the calling application.
4. More detailed diagnostic information is available through dlerror(), the _EDC_DLL_DIAG environment variable, and the Language Environment DLL Failure control block (CEEDLLF) chain.
5. This function is not available under SPC, MTF and CSP environments.

Example

The following example shows how dlopen() and dlsym() can be used to access either function or data objects. For simplicity, error checking has been omitted.

```
void    *handle;
int     *iptr, (*fptr)(int);

/* open the needed object */
handle = dlopen("/usr/home/me/libfoo.so", RTLD_LOCAL | RTLD_LAZY);

/* find the address of function and data objects */
fptr = (int (*)(int))dlsym(handle, "my_function");
iptr = (int *)dlsym(handle, "my_object");

/* invoke function, passing value of integer as a parameter */
(*fptr)(*iptr);
```

Related information

- “dlclose() — Close a dlopen() object” on page 385
- “dlerror() — Get diagnostic information” on page 386
- “dlopen() — Gain access to a dynamic link library” on page 387
- “dlfcn.h” on page 23

dllfree() — Free the supplied dynamic link library

Standards

Standards / Extensions	C or C++	Dependencies
C Library	both	

Format

```
#include <dll.h>

int dllfree(dllhandle *dllHandle);
```

General description

Frees the supplied dynamic link library (DLL). It also deletes the DLL from memory if the handle was the last handle accessing the DLL.

Returned value

dllfree() returns one of the following values and set errno if the return code is not 0:

Value	Meaning
0	Successful
1	The `dllHandle` supplied is NULL or `dllhandle` is inactive.
2	There are no DLLs to be deleted.
3	DLL is not physically deleted because there is another `dllHandle` for this DLL or there is an implicit reference to the DLL.
4	Delete of DLL failed.
5	No match is found for input `dllHandle`.
6	Not supported under this environment.
7	C++ destructors are currently running for this DLL. A dllfree() is already in progress.
8	The handle passed to dllfree() was obtained from dlopen().

Usage notes

1. This function is deprecated; use dlclose() instead.
2. This function is not available under SPC, MTF and CSP environments.
3. If a DLL is loaded implicitly, it cannot be deleted with dllfree(). For more information on the implicit use of DLLs, see *z/OS XL C/C++ Programming Guide*.
4. DLLs that are loaded explicitly, that is with dllload(), and are not freed with a corresponding call to dllfree(), are freed automatically at enclave termination in LIFO sequence.
5. C++ destructors are executed only once, when the DLL load module is physically deleted.
6. More detailed diagnostic information is available through the _EDC_DLL_DIAG environment variable, and the Language Environment DLL Failure control block (CEEDLLF) chain. The default action is to issue an error message to the Language Environment message file.

Example

CELEBDL4

```
/* CELEBDL4

   The following example shows how to use dllfree() to free the
   dllhandle for the DLL stream.

 */
#include <stdio.h>
#include <dll.h>
#include <stdlib.h>

int main() {
    dllhandle *handle;
    char *name="stream";
    int (*fptr1)(int);
    int (*fptr)(int);
    int *ptr_var1;
    int *ptr_var;
    int rc=0;

    handle = dllload(name); /* call to stream DLL */
    if (handle == NULL) {
       perror("failed on call to stream DLL");
       exit(-1);
    }

    fptr1 = (int (*)(int)) dllqueryfn(handle,"f1");
    /* retrieving f1 function */
    if (fptr == NULL) {
       perror("failed on retrieving f1 function");
       exit(-2);
    }

    ptr_var = dllqueryvar(handle,"var1");
    /* retrieving var1 variable */
    if (ptr_var1 == NULL) {
       perror("failed on retrieving var1 variable");
       exit(-3);
    }

    rc = fptr(*ptr_var1); /* execute DLL function f1 */
    *ptr_var++;           /* increment value of var1 */

    rc = dllfree(handle); /* freeing handle to stream DLL */
    if (rc != 0) {
       perror("failed on dllfree call");
    }
    return (0);
}
```

Related information

- "dll.h" on page 23
- "dlclose() — Close a dlopen() object" on page 385
- "dllload() — Load the dynamic link library and connect it to the application"
- "dllqueryfn() — Obtain a pointer to a dynamic link library function" on page 396
- "dllqueryvar() — Obtain a pointer to a dynamic link library variable" on page 397

dllload() — Load the dynamic link library and connect it to the application

Standards

Standards / Extensions	C or C++	Dependencies
C Library	both	

Format

```
#include <dll.h>

dllhandle *dllload(const char *dllName);
```

General description

Note: This function is deprecated; use dlopen() instead.

Loads the dynamic link library (DLL) into memory (if it has not been previously loaded) and connects it to the application. The function that called the DLL receives a handle that uniquely identifies the requested DLL for subsequent explicit requests for that DLL.

A different handle is returned for each successful call to dllload(). A DLL is physically loaded only once, even though there may be many calls to dllload(). C++ constructors are run only once.

The `dllName` identifies the DLL load module to be loaded. It must be a character string terminated with the NULL character. The DLL module must be a member of a PDS or an alias to it.

Note: The `AMODE` of the application must be the same as the `AMODE` of the DLL load module.

This function is not available under SPC, MTF and CSP environments.

The dllName identifies the DLL load to be loaded. It must be a character string, terminated with the NULL character. The DLL module must be a member of a PDS or an alias to it.

If the file argument contains a single slash ('/'), it is used as the z/OS UNIX file system path name for the DLL. If the environment variable LIBPATH is set, each directory listed will be searched for the DLL. Otherwise, the current directory will be searched.

Note: Searching for a DLL in the z/OS UNIX file system is case-sensitive.

If the file argument begins with two slashes ('//'), then an attempt is made to load the DLL from the caller's MVS load library search order (in order: STEPLIB/JOBLIB, LPA, Link List). The DLL name must be eight characters or less, and is converted to uppercase. Note that qualified DLL names are not supported and the MVS load library search order is used (for example, update or use STEPLIB to specify any number of qualifiers to be included in the search).

If the file argument doesn't begin with one or two slashes ('/' or //"), and doesn't contain a single slash ('/') anywhere in the name, then it is ambiguous as to where the DLL resides.

- If the POSIX(ON) runtime option is specified, then the z/OS UNIX file system is searched first for the DLL, and if not found, the MVS load library is searched.
- If the POSIX(OFF) runtime option is specified, then the MVS load library is searched first for the DLL, and if not found, the z/OS UNIX file system is searched.

Under CICS environment, the search sequence for DLL load modules is the same as that used for dynamically loaded CICS modules. Loading DLLs from the z/OS UNIX file system is not supported under CICS.

For more information about how DLLs are loaded and how the search sequence is used, see the topic about loading DLLs in *z/OS Language Environment Programming Guide*.

Returned value

If successful, dllload() returns a unique handle that identifies the DLL.

If unsuccessful, dllload() returns NULL and may set errno to one of the following values:

Error Code
Description

ELEFENCE
The DLL contains a member language not supported on this version of the operating system.

ENOEXEC
The new process image file has the appropriate access permission but is not in the proper format.

Note: Reason codes further qualify the errno. For most of the reason codes, see *z/OS UNIX System Services Messages and Codes*.

For ENOEXEC, the reason codes are:

Reason Code	Explanation
X'xxxx0C27'	The target z/OS UNIX file system file is not in the correct format to be an executable file.
X'xxxx0C31'	The target z/OS UNIX file system file is built at a level that is higher than that supported by the running system.

Usage notes

1. More detailed diagnostic information is available through the _EDC_DLL_DIAG environment variable, and the Language Environment DLL Failure control block (CEEDLLF) chain.

Example

CELEBDL1

```
/* CELEBDL1

   The following example shows how to invoke dllload() functions
   from a simple C application.

 */
#include <stdio.h>
#include <dll.h>

main() {
    dllhandle *handle;
    char *name="stream";

    handle = dllload(name);
    if (handle == NULL) {
       perror("failed on dllload of stream DLL");
       exit(-1);
    }
}
```

Related information

- “dll.h” on page 23
- “dlopen() — Gain access to a dynamic link library” on page 387

- “dllfree() — Free the supplied dynamic link library” on page 391
- “dllqueryfn() — Obtain a pointer to a dynamic link library function”
- “dllqueryvar() — Obtain a pointer to a dynamic link library variable” on page 397

dllqueryfn() — Obtain a pointer to a dynamic link library function

Standards

Standards / Extensions	C or C++	Dependencies
C Library	both	

Format

```
#include <dll.h>

void (*dllqueryfn(dllhandle *dllHandle, const char *funcName))();
```

General description

Note: This function is deprecated; use dlsym() instead.

Obtains a pointer to a dynamic link library (DLL) function (funcName). It uses the dllHandle returned from a previous successful call to dllload() for input. funcName represents the name of an exported function from the DLL. It must be a character string terminated with the NULL character.

This function is not available under the SPC, MTF, and CSP environments.

Returned value

If successful, dllqueryfn() returns a pointer to a function, *funcName*, that can be used to invoke the desired function in a DLL.

If unsuccessful, dllqueryfn() returns NULL and sets errno.

Usage notes

1. More detailed diagnostic information is available through the _EDC_DLL_DIAG environment variable, and the Language Environment DLL Failure control block (CEEDLLF) chain.

Example

CELEBDL2

```
/* CELEBDL2

   The following example shows how to use dllqueryfn() to obtain
   a pointer to a function, f1 that is in DLL load module stream.

 */
#include <stdio.h>
#include <dll.h>

main() {
    dllhandle *handle;
    char *name="stream";
    int (*fptr1)();
    handle = dllload(name);
    if (handle == NULL) {
       perror("failed on dllload of stream DLL");
       exit(-1);
    }

    fptr1 = (int (*)()) dllqueryfn(handle,"f1");
    if (fptr1 == NULL) {
       perror("failed on retrieving f1 function");
       exit (-2);
    }
}
```

Related information

- "dll.h" on page 23
- "dllfree() — Free the supplied dynamic link library" on page 391
- "dllload() — Load the dynamic link library and connect it to the application" on page 393
- "dllqueryvar() — Obtain a pointer to a dynamic link library variable"

dllqueryvar() — Obtain a pointer to a dynamic link library variable

Standards

Standards / Extensions	C or C++	Dependencies
C Library	both	

Format

```
#include <dll.h>

void* dllqueryvar(dllhandle *dllHandle, const char *varName);
```

General description

Obtains a pointer to a dynamic link library (DLL) variable (*varName*). It uses the *dllHandle* returned from a previous successful call to dllload() for input. *varName* represents the name of an exported variable from the DLL. It must be a character string terminated with the NULL character.

This function is not available under SPC, MTF and CSP environments.

Returned value

If successful, dllqueryvar() returns a pointer to a variable in the storage of the DLL.

If unsuccessful, dllqueryvar() returns NULL and sets errno.

Usage notes

1. More detailed diagnostic information is available through the _EDC_DLL_DIAG environment variable, and the Language Environment DLL Failure control block (CEEDLLF) chain.

Example

CELEBDL3

```
/* CELEBDL3

   The following example shows how to use dllqueryvar() to obtain a
   pointer to a variable, var1, that is in DLL load module stream.

 */
#include <stdio.h>
#include <dll.h>

int main() {
    dllhandle *handle;
    char *name="stream";
    int (*fptr1)(int);
    int *ptr_var1;
    int rc=0;

    handle = dllload(name);
    if (handle == NULL) {
       perror("failed on dllload of stream DLL");
       exit(-1);
    }

    fptr1 = (int (*)(int)) dllqueryfn(handle,"f1");
    /* retrieving f1 function */
    if (fptr1 == NULL) {
       perror("failed on retrieving f1 function");
       exit(-2);
    }

    ptr_var1 = dllqueryvar(handle,"var1");
    if (ptr_var1 == NULL) {
       perror("failed on retrieving var1 variable");
       exit(-3);
    }
}
```

Related information

- “dll.h” on page 23
- “dllfree() — Free the supplied dynamic link library” on page 391
- “dllload() — Load the dynamic link library and connect it to the application” on page 393
- “dllqueryfn() — Obtain a pointer to a dynamic link library function” on page 396

dn_comp() — Resolver domain name compression

Standards

Standards / Extensions	C or C++	Dependencies
BSD 4.3	both	OS/390 V2R8

Format

```
#define _XOPEN_SOURCE_EXTENDED 1
#include <sys/types.h>
#include <netinet/in.h>
#include <arpa/nameser.h>
#include <resolv.h>

int dn_comp(const char *exp_dn, u_char *comp_dn, int length,
            u_char **dnptrs, u_char **lastdnptr);
```

General description

The dn_comp() function compresses the domain name *exp_dn* and stores it in *comp_dn*. The size of the compressed name is returned or -1 if there were errors. The size of the array pointed to by *comp_dn* is given by *length*. The compression uses an array of pointers *dnptrs* to previously-compressed names in the current message. The first pointer points to the beginning of the message and the list ends with NULL. The limit to the array is specified by *lastdnptr*.

A side effect of dn_comp() is to update the list of pointers for labels inserted into the message as the name is compressed. If *dnptr* is NULL, names are not compressed. If *lastdnptr* is NULL, the list of labels is not updated.

Note: The dn_comp() function has a dependency on the level of the Enhanced ASCII Extensions. See “Enhanced ASCII support” on page 2113 for details.

Returned value

If successful, dn_comp() returns the size of the compressed name.

If unsuccessful, dn_comp() returns -1 to report the error, when the name to be compressed was not found before the end of the buffer was reached.

There are no documented errno values.

Related information

- “arpa/nameser.h” on page 17
- “netinet/in.h” on page 53
- “resolv.h” on page 62
- “sys/types.h” on page 75
- “dn_expand() — Resolver domain name expansion”
- “dn_find() — Resolver domain name find” on page 400
- “dn_skipname() — Resolver domain name skipping” on page 401
- “res_init() — Domain name resolver initialization” on page 1435
- “res_mkquery() — Make resolver query for domain name servers” on page 1438
- “res_query() — Resolver query for domain name servers” on page 1439
- “res_querydomain() — Build domain name and resolver query” on page 1441
- “res_search() — Resolver query for domain name servers” on page 1442
- “res_send() — Send resolver query for domain name servers” on page 1443

dn_expand() — Resolver domain name expansion

Standards

Standards / Extensions	C or C++	Dependencies
BSD 4.3	both	OS/390 V2R8

Format

```
#define _XOPEN_SOURCE_EXTENDED 1
#include <sys/types.h>
#include <netinet/in.h>
#include <arpa/nameser.h>
#include <resolv.h>

int dn_expand(const u_char *msg, const u_char *eomorig,
              const u_char *comp_dn, char *exp_dn, int length);
```

General description

The dn_expand() function expands the compressed domain name *comp_dn* to a full domain name. The compressed name is contained in a query or reply message; *msg* is a pointer to the beginning of the message. The expanded name is placed in the buffer indicated by *exp_dn* which is of size *length*. The size of the expanded name is returned or -1 if there was an error.

Note: The dn_expand() function has a dependency on the level of the Enhanced ASCII Extensions. See "Enhanced ASCII support" on page 2113 for details.

Returned value

If successful, dn_expand() returns the size of the expanded name.

If unsuccessful, dn_expand() returns -1 to report the error, when the name to be expanded was not found before the end of the buffer was reached.

There are no documented errno values.

Related information

- "arpa/nameser.h" on page 17
- "netinet/in.h" on page 53
- "resolv.h" on page 62
- "sys/types.h" on page 75
- "dn_comp() — Resolver domain name compression" on page 398
- "dn_find() — Resolver domain name find"
- "dn_skipname() — Resolver domain name skipping" on page 401
- "res_init() — Domain name resolver initialization" on page 1435
- "res_mkquery() — Make resolver query for domain name servers" on page 1438
- "res_query() — Resolver query for domain name servers" on page 1439
- "res_querydomain() — Build domain name and resolver query" on page 1441
- "res_search() — Resolver query for domain name servers" on page 1442
- "res_send() — Send resolver query for domain name servers" on page 1443

dn_find() — Resolver domain name find

Standards

Standards / Extensions	C or C++	Dependencies
BSD 4.3	both	OS/390 V2R8

Format

```
#define _XOPEN_SOURCE_EXTENDED 1
#include <sys/types.h>
#include <netinet/in.h>
#include <arpa/nameser.h>
#include <resolv.h>

int dn_find(u_char *exp_dn, u_char *msg, u_char **dnptrs, u_char **lastdnptr);
```

General description

The dn_find() function will search for the expanded name *exp_dn* in the list of previously compressed names *dnptrs*.

dnptrs is the pointer to the first name in the list, not the pointer to the start of the message. The limit to the array is specified by *lastdnptr*.

Note: The dn_find() function has a dependency on the level of the Enhanced ASCII Extensions. See "Enhanced ASCII support" on page 2113 for details.

Returned value

If successful, dn_find() returns the offset of the expanded name *exp_dn* found in the message.

If unsuccessful, dn_find() returns -1 to report the error, when the name was not found before the end of the list was reached.

There are no documented errno values.

Related information

- "arpa/nameser.h" on page 17
- "netinet/in.h" on page 53
- "resolv.h" on page 62
- "sys/types.h" on page 75
- "dn_comp() — Resolver domain name compression" on page 398
- "dn_expand() — Resolver domain name expansion" on page 399
- "dn_skipname() — Resolver domain name skipping"
- "res_init() — Domain name resolver initialization" on page 1435
- "res_mkquery() — Make resolver query for domain name servers" on page 1438
- "res_query() — Resolver query for domain name servers" on page 1439
- "res_querydomain() — Build domain name and resolver query" on page 1441
- "res_search() — Resolver query for domain name servers" on page 1442
- "res_send() — Send resolver query for domain name servers" on page 1443

dn_skipname() — Resolver domain name skipping

Standards

Standards / Extensions	C or C++	Dependencies
BSD 4.3	both	OS/390 V2R8

Format

```
#define _XOPEN_SOURCE_EXTENDED 1
#include <sys/types.h>
#include <netinet/in.h>
#include <arpa/nameser.h>
#include <resolv.h>

int dn_skipname(const u_char *comp_dn, u_char *eom);
```

General description

The dn_skipname() function skips the compressed domain name *comp_dn* and returns the position in the answer buffer that follows the *comp_dn* compressed domain name. If the information supplied in *comp_dn* is not a compressed domain name, -1 is returned to report the error.

Note: The dn_skipname() function has a dependency on the level of the Enhanced ASCII Extensions. See "Enhanced ASCII support" on page 2113 for details.

Returned value

If successful, dn_skipname() returns the position in the answer buffer that follows the *comp_dn* compressed domain name.

If unsuccessful, dn_skipname() returns -1 to report the error, when the name to be skipped was not found before the end of the buffer was reached.

There are no documented errno values.

Related information

- “arpa/nameser.h” on page 17
- “netinet/in.h” on page 53
- “resolv.h” on page 62
- “sys/types.h” on page 75
- “dn_comp() — Resolver domain name compression” on page 398
- “dn_expand() — Resolver domain name expansion” on page 399
- “dn_find() — Resolver domain name find” on page 400
- “res_init() — Domain name resolver initialization” on page 1435
- “res_mkquery() — Make resolver query for domain name servers” on page 1438
- “res_query() — Resolver query for domain name servers” on page 1439
- “res_querydomain() — Build domain name and resolver query” on page 1441
- “res_search() — Resolver query for domain name servers” on page 1442
- “res_send() — Send resolver query for domain name servers” on page 1443

drand48() — Pseudo-random number generator

Standards

Standards / Extensions	C or C++	Dependencies
XPG4 XPG4.2 Single UNIX Specification, Version 3	both	

Format

```
#define _XOPEN_SOURCE
#include <stdlib.h>

double drand48(void);
```

General description

The drand48(), erand48(), jrand48(), lrand48(), mrand48() and nrand48() functions generate uniformly distributed pseudo-random numbers using a linear congruential algorithm and 48-bit integer arithmetic.

The functions drand48() and erand48() return nonnegative, double-precision, floating-point values, uniformly distributed over the interval [0.0,1.0). These functions have been extended so that the returned value will be in the proper floating-point format (hexadecimal or IEEE) based on the floating-point mode of the invoking thread.

The functions lrand48() and nrand48() return nonnegative, long integers, uniformly distributed over the interval [0,2**31).

The functions mrand48() and jrand48() return signed long integers, uniformly distributed over the interval [-2**31,2**31).

The drand48() function generates the next 48-bit integer value in a sequence of 48-bit integer values, X(i), according to the linear congruential formula:

```
X(n+1) = (aX(n) + c)mod(2**48)      n>=0
```

The initial values of X, a, and c are:

```
X(0) = 1
a   = 5deece66d (base 16)
c   = b         (base 16)
```

C/370 provides storage to save the most recent 48-bit integer value of the sequence, X(i). This storage is shared by the drand48(), lrand48() and mrand48() functions. The value, X(n), in this storage may be reinitialized by calling the lcong48(), seed48() or srand48() function. Likewise, the values of a and c, may be changed by calling the lcong48() function. Thereafter, whenever the seed48() or srand48() function is called to change X(n), the initial values of a and c are also reestablished.

Special behavior for z/OS UNIX Services: You can make the drand48() function and other functions in the drand48 family thread-specific by setting the environment variable _RAND48 to the value THREAD before calling any function in the drand48 family.

If you do not request thread-specific behavior for the drand48 family, C/370 serializes access to the storage for X(n), a and c by functions in the drand48 family when they are called by a multithreaded application.

If thread-specific behavior is requested, and the drand48() function is called from thread t, the drand48() function generates the next 48-bit integer value in a sequence of 48-bit integer values, X(t,i), for the thread t. The sequence of values for a thread is generated according to the linear congruential formula:

```
   X(t,n+1) = (a(t)X(t,n) + c(t))mod(2**48)      n>=0
```

The initial values of X(t), a(t) and c(t) for the thread t are:

```
   X(t,0) = 1
   a(t)   = 5deece66d (base 16)
   c(t)   = b         (base 16)
```

C/370 provides storage which is specific to the thread t to save the most recent 48-bit integer value of the sequence, X(t,i), generated by the drand48(), lrand48() or mrand48() function. The value, X(t,n), in this storage may be reinitialized by calling the lcong48(), seed48() or srand48() function from the thread t. Likewise, the values of a(t) and c(t) for thread t may be changed by calling the lcong48() function from the thread. Thereafter, whenever the seed48() or srand48() function is called from the thread t to change X(t,n), the initial values of a(t) and c(t) are also reestablished.

Returned value

drand48() transforms the generated 48-bit value, X(n+1), to a double-precision, floating-point value on the interval [0.0,1.0) and returns this transformed value.

Special behavior for z/OS UNIX Services: If thread-specific behavior is requested for the drand48 family and rand48() is called on thread t, drand48() transforms the generated 48-bit value, X(t,n+1), to a double-precision, floating-point value on the interval [0.0,1.0) and returns this transformed value.

Related information

- "stdlib.h" on page 70
- "erand48() — Pseudo-random number generator" on page 426
- "__isBFP() — Determine application floating-point format" on page 904
- "jrand48() — Pseudo-random number generator" on page 928
- "lcong48() — Pseudo-random number initializer" on page 939
- "lrand48() — Pseudo-random number generator" on page 1010
- "mrand48() — Pseudo-random number generator" on page 1101
- "nrand48() — Pseudo-random number generator" on page 1148
- "seed48() — Pseudo-random number initializer" on page 1467
- "srand48() — Pseudo-random number initializer" on page 1718

dup() — Duplicate an open file descriptor

Standards

Standards / Extensions	C or C++	Dependencies
POSIX.1 XPG4 XPG4.2 Single UNIX Specification, Version 3	both	

Format

```
#define _POSIX_SOURCE
#include <unistd.h>

int dup(int fildes);
```

General description

Returns a new file descriptor that is the lowest numbered available descriptor. The new file descriptor refers to the same open file as *fildes* and shares any locks that may be associated with *fildes*.

The following operations are equivalent:

```
fd = dup(fildes);
fd = fcntl(fildes,F_DUPFD,0);
```

For further information, see "fcntl() — Control open file descriptors" on page 474.

Note: When *fildes* is an XTI endpoint, the lowest numbered available file descriptor must not exceed 65535.

Returned value

If successful, dup() returns a new file descriptor.

If unsuccessful, dup() returns -1 and sets errno to one of the following values:

Error Code
Description

EBADF
fildes is not a valid open file descriptor.

EMFILE
The process has already reached its maximum number of open file descriptors.

Example

CELEBD05

```
/* CELEBD05

   This example duplicates an open file descriptor, using dup().

 */
#define _POSIX_SOURCE
#include <errno.h>
#include <sys/stat.h>
#include <sys/types.h>
#include <unistd.h>
#undef _POSIX_SOURCE
#include <stdio.h>

void print_inode(int fd) {
 struct stat info;
 if (fstat(fd, &info) != 0)
   fprintf(stderr,"fstat() error for fd %d: %s\n",fd,strerror(errno));
 else
   printf("The inode of fd %d is %d\n", fd, (int) info.st_ino);
}

main() {
  int fd;
  if ((fd = dup(0)) < 0)
    perror("&dupf error");
  else {
    print_inode(0);
    print_inode(fd);
    puts("The file descriptors are different but");
    puts("they point to the same file.");
    close(fd);
  }
}
```

Output

```
The inode of fd 0 is 30
The inode of fd 3 is 30
The file descriptors are different but
they point to the same file.
```

Related information

- “unistd.h” on page 82
- “close() — Close a file” on page 288
- “creat() — Create a new file or rewrite an existing one” on page 340
- “dup2() — Duplicate an open file descriptor to another” on page 406
- “exec functions” on page 436
- “fcntl() — Control open file descriptors” on page 474
- “open() — Open a file” on page 1152
- “pipe() — Create an unnamed pipe” on page 1179

dup2() — Duplicate an open file descriptor to another

Standards

Standards / Extensions	C or C++	Dependencies
POSIX.1 Single UNIX Specification, Version 3	both	

Format

```
#define _POSIX_SOURCE
#include <unistd.h>

int dup2(int fd1, int fd2);
```

General description

Returns a file descriptor with the value *fd2*. *fd2* now refers to the same file as *fd1*, and the file that was previously referred to by *fd2* is closed. The following conditions apply:

- If *fd2* is less than 0 or greater than OPEN_MAX, dup2() returns -1 and sets errno to EBADF.
- If *fd1* is a valid file descriptor and is equal to *fd2*, dup2() returns *fd2* without closing it; F_CLOEXEC is not cleared.
- If *fd1* is not a valid file descriptor, dup2() fails and does not close *fd2*.
- If a file descriptor does not already exist, dup2() can be used to create one, a duplicate of *fd1*. F_CLOEXEC is cleared in fd2.

Note: If *fd1* is an XTI endpoint, *fd2* must not exceed 65535.

Returned value

If successful, dup2() returns *fd2*.

If unsuccessful, dup2() returns -1 and sets errno to one of the following values:

Error Code
Description

EBADF
fd1 is not a valid file descriptor, or *fd2* is less than 0 or greater than **OPEN_MAX**.

EINTR
dup2() was interrupted by a signal.

Example

CELEBD06

```
/* CELEBD06

   This example duplicates an open file descriptor, using dup2().

 */

#define _POSIX_SOURCE
#include <errno.h>
#include <fcntl.h>
```

```
#include <sys/stat.h>
#include <sys/types.h>
#include <unistd.h>
#undef _POSIX_SOURCE
#include <stdio.h>

void print_inode(int fd) {
 struct stat info;
 if (fstat(fd, &info) != 0)
   fprintf(stderr,"fstat() error for fd %d: %s\n",fd,strerror(errno));
 else
   printf("The inode of fd %d is %d\n", fd, (int) info.st_ino);
}

main() {
  int fd;
  char fn[]="dup2.file";

  if ((fd = creat(fn, S_IWUSR)) < 0)
    perror("creat() error");
  else {
    print_inode(fd);
    if ((fd = dup2(0, fd)) < 0)
      perror("dup2() error");
    else {
      puts("After dup2()...");
      print_inode(0);
      print_inode(fd);
      puts("The file descriptors are different but they");
      puts("point to the same file which is different than");
      puts("the file that the second fd originally pointed to.");
      close(fd);
    }
    unlink(fn);
  }
}
```

Output

```
The inode of fd 3 is 3031
After dup2()...
The inode of fd 0 is 30
The inode of fd 3 is 30
The file descriptors are different but they
point to the same file which is different than
the file that the second fd originally pointed to.
```

Related information

- “unistd.h” on page 82
- “close() — Close a file” on page 288
- “creat() — Create a new file or rewrite an existing one” on page 340
- “dup() — Duplicate an open file descriptor” on page 404
- “exec functions” on page 436
- “fcntl() — Control open file descriptors” on page 474
- “open() — Open a file” on page 1152
- “pipe() — Create an unnamed pipe” on page 1179

dynalloc() — Allocate a data set

Standards

Standards / Extensions	C or C++	Dependencies
C Library	both	

Format

```
#include <dynit.h>

int dynalloc(__dyn_t *dyn_parms);
```

General description

The dynalloc() function dynamically allocates a MVS data set using the MVS SVC 99 service and by building an SVC 99 parameter list based on parameters specified in *dyn_parms*. The dynalloc() function corresponds to verb code 1 for SVC 99. To use other SVC 99 verb codes, see “svc99() — Access supervisor call” on page 1788.

To avoid infringing on the user's name space, this nonstandard function has two names. One name is prefixed with two underscore characters, and one name is not. The name without the prefix underscore characters is exposed only when you use LANGLVL(EXTENDED).

To use this function, you must either invoke the function using its external entry point name (that is, the name that begins with two underscore characters), or compile your application with LANGLVL(EXTENDED). When you use LANGLVL(EXTENDED). any relevant information in the header is also exposed.

The request block extension and the error message parameter list can be used to process the messages returned by SVC99 when an error occurs. To use this feature you must allocate and initialize these structures using the processes described in *z/OS MVS Programming: Authorized Assembler Services Guide*. You must also make them available to the dynalloc() function by assigning their addresses to __rbx or __emsgparmlist.

Because additional fields have been added to the `__dyn_t` structure, you should recompile existing source code with the latest dynit.h header file to access the new fields.

Some values, such as *ddname* and *dsname*, will be converted to uppercase internally when they are used by the dynalloc() function.

To dynamically allocate an MVS data set, you should:
- Invoke the dyninit() function with a variable of type __dyn_t.
- Assign values to the appropriate fields in the variable that will satisfy the svc99() request.
- Invoke the dynalloc() function with this variable.

Table 21 on page 409 describes the elements that are part of the `__dyn_t` structure, organized as defined in the structure.

Note: For more complete information about the `__dyn_t` structure elements, their usage and restrictions, see the description of SVC 99, the SVC 99 extension block, and the text unit keys and values in *z/OS MVS Programming: Authorized Assembler Services Guide* and *z/OS MVS JCL Reference*.

Table 21. Description of __dyn_t Structure Elements

Element	Text Unit Key	Text Unit Value	Type	Description
__ddname	DALDDNAM	0001	char *	ddname (maximum length of 8)[1]. If 8 question marks (????????) are specified, it means that the request expects a system-generated ddname returned.
__dsname	DALDSNAM	0002	char *	Fully qualified data-set name (maximum length of 44)[1].
__sysout	DALSYSOU	0018	char	The class of the system output data set (for example, SYSOUT=A). Values are: alphabetic character, or the macro __DEF_CLASS, to specify the default class.
__sysoutname	DALSPGNM	0019	char *	Program name for sysout. The __sysout field must be specified with this field (maximum length of 8)[1].
__member	DALMEMBR	0003	char *	Member of a partitioned data set to be allocated (maximum length of 8)[1].
__status	DALSTATS	0004	char	Data set status. Values are: __DISP_OLD, __DISP_NEW, __DISP_MOD, and __DISP_SHR, which are defined in dynit.h.
__normdisp	DALNDISP	0005	char	Specifies the normal disposition of a data set. Values are: __DISP_CATLG, __DISP_UNCATLG, __DISP_DELETE, and __DISP_KEEP, which are defined in dynit.h.
__conddisp	DALCDISP	0006	char	Specifies the conditional disposition of a data set. Values are: __DISP_CATLG, __DISP_UNCATLG, __DISP_DELETE, and __DISP_KEEP, which are defined in dynit.h.
__unit	DALUNIT	0015	char *	Unit name of the device that the data set will (or does, if it already exists) reside on (maximum length of 8)[1].
__volser	DALVLSER	0010	char *	Volume serial number of the device a data set will (or does, if it already exists) reside on (maximum length of 6)[1].

Table 21. Description of __dyn_t Structure Elements (continued)

Element	Text Unit Key	Text Unit Value	Type	Description
__dsorg	DALDSORG	003C	char	Data set organization of a data set. Values are: **__DSORG_unknown** Unknown **__DSORG_VSAM** VSAM **__DSORG_GS** Graphics **__DSORG_PO** Partitioned organization **__DSORG_POU** Partitioned organization unmovable **__DSORG_DA** Direct access **__DSORG_DAU** Direct access unmovable **__DSORG_PS** Physical sequential **__DSORG_PSU** Physical sequential unmovable.
__alcunit	DALCYL, DALTRK	0008, 0007	char	Unit of space allocation for a data set. Values are: __CYL and __TRK. To specify allocation units in blocks, use the field __avgblk.
__primary	DALPRIME	000A	int	Primary space allocation for a data set.
__secondary	DALSECND	000B	int	Secondary space allocation for a data set.
__dirblk	DALDIR	000C	int	Number of directory blocks for a partitioned data set.
__avgblk	DALBLKLN	0009	int	Specifies the unit of space allocation to be blocks and sets the average block length.

Table 21. Description of __dyn_t Structure Elements (continued)

Element	Text Unit Key	Text Unit Value	Type	Description
__recfm		0049	short	Record format of a data set. The following macros in dynit.h can be added together to determine the __recfm value: **_M_** Machine-code printer-control characters **_A_** ASA printer-control characters **_S_** Standard fixed, spanned variable **_B_** Blocked **_D_** Variable ASCII records **_V_** Variable **_F_** Fixed **_U_** Undefined **_FB_** Fixed blocked **_VB_** Variable blocked **_FBS_** Fixed blocked standard **_VBS_** Variable blocked standard. For example, to specify a recfm of FBA, set: _recfm = _FB_ + _A_
__blksize	DALBLKSZ	0030	short	Block size of a data set.
__lrecl	DALLRECL	0042	unsigned short	Record length of a data set.
__volrefds	DALLVLRDS	0014	char *	Fully qualified name of a cataloged data set to be used as a model for obtaining volume serial information (maximum length of 44)[1].
__dcbrefds	DALDCBDS	002C	char *	Fully qualified name of a cataloged data set to be used as a model for obtaining DCB information (maximum length of 44)[1].
__dcbrefdd	DALLDCBDD	002D	char *	ddname of a data set to be used as a model for obtaining DCB information (maximum length of 9)[1]. For more information, see *z/OS MVS Programming: Authorized Assembler Services Guide*.
__misc_flags			unsigned char	Specifies the attributes. See “Example” on page 414 for instructions on how to specify the flags shown below using a logical | (OR).
__CLOSE	DALCLOSE	001C	unsigned char	(Flag) Deallocate data set when file is closed.
__RELEASE	DALRLSE	000D	unsigned char	(Flag) Release unused space when file is closed.
__CONTIG	DALSPRFRM	000E	unsigned char	(Flag) Allocate space contiguously.
__ROUND	DALROUND	000F	unsigned char	(Flag) Allocate space in whole cylinders when blocks are requested.
__TERM	DALTERM	0028	unsigned char	(Flag) Time-sharing terminal is to be used as I/O device.

Table 21. Description of __dyn_t Structure Elements (continued)

Element	Text Unit Key	Text Unit Value	Type	Description
__DUMMY_DSN	DALDUMMY	0024	unsigned char	(Flag) Dummy data set is to be allocated.
__HOLDQ	DALSHOLD	0059	unsigned char	(Flag) Hold queue routing for sysout data set.
__PERM	DALPERMA	0052	unsigned char	(Flag) Set permanent allocation attribute.
__password	DALPASSW	0050	char *	Password for a password-protected data set. The dsname field must be specified with this field (maximum length of 8)[1].
__miscitems			char * __ptr32 * __ptr32	For all other text unit keys not available in __dyn_t, this pointer will let you specify an array of text unit strings. If you specify this field, you must turn the high bit on the last item (as in svc99()). Use the bitwise inclusive-OR (|) operand with the last item and the hexadecimal value 0x80000000.
__infocode			short	Returns the information code returned by the MVS dynamic allocation functions. For more information, see *z/OS MVS Programming: Authorized Assembler Services Guide*.
__errcode			short	Returns the error code returned by the MVS dynamic allocation functions. For more information, see *z/OS MVS Programming: Authorized Assembler Services Guide*.
__storclass	DALSTCL	8004	char *	Specifies the storage class of system managed storage.
__mgntclass	DALMGCL	8005	char *	Specifies the management class of a data set.
__dataclass	DALDACL	8006	char *	Specifies the data class of a data set.
__recorg	DALRECO	800B	char	Specifies the record organization of a VSAM data set. Values are: __KS, __ES, __RR, __LS.
__keyoffset	DALKEYO	800C	short	Specifies the key offset. The position of the first byte of the key in records of the specified VSAM data set.
__keylength	DALKYLEN	0040	short	Specifies the length in bytes of the keys used in the data set.
__refdd	DALREFD	800D	char *	Specifies the name of the JCL DD statement from which the attributes are to be copied. For more information, see *z/OS MVS Programming: Authorized Assembler Services Guide*.
__like	DALLIKE	800F	char *	Specifies the name of the model data set from which the attributes are to be copied.
__dsntype	DALDSNT	8012	char	Specifies the type attributes of a data set. Valid types include __DSNT_BASIC, __DSNT_EXTPREF, __DSNT_EXTREQ, __DSNT_HFS, __DSNT_LARGE, __DSNT_LIBRARY, __DSNT_PDS, and __DSNT_PIPE.

Table 21. Description of __dyn_t Structure Elements (continued)

Element	Text Unit Key	Text Unit Value	Type	Description
__pathname	DALPATH	8017	char *	Pathname (maximum length is 255)[1]. See *z/OS V2R2.0 UNIX System Services User's Guide* for the pathname format.
__pathopts	DALPOPT	8018	int	Specifies file options for the HFS file. Values are: __PATH_OCREAT, __PATH_OAPPEND, __PATH_OEXCL, __PATH_ONOCTTY, __PATH_OTRUNC, __PATH_ONONBLOCK, __PATH_ORDONLY, __PATH_OWRONLY, __PATH_ORDWR. For information about the file options, see *z/OS MVS JCL Reference*. For information about DYNALLOC, see *z/OS MVS Programming: Authorized Assembler Services Guide*.
__pathmode	DALPMDE	8019	int	Specifies the file access attributes for the HFS file. Values are: __PATH_SIRUSR, __PATH_SIWUSR, __PATH_SIXUSR, __PATH_SIRWXU, __PATH_SIRGRP, __PATH_SIWGRP, __PATH_SIXGRP, __PATH_SIRWXG, __PATH_SIROTH, __PATH_SIWOTH, __PATH_SIXOTH, __PATH_SIRWXO, __PATH_SISUID, __PATH_SISGID. For information on the file attributes, refer to *z/OS MVS JCL Reference*. For information on DYNALLOC, refer to *z/OS MVS Programming: Authorized Assembler Services Guide*.
__pathndisp	DALPNDS	801A	char	Specifies the normal HFS file disposition desired. It is either __DISP_KEEP or __DISP_DELETE
__pathcdisp	DALPCDS	801B	char	Specifies the abnormal HFS file disposition desired. It is either __DISP_KEEP or __DISP_DELETE
__rbx			__S99rbx_t * __ptr32	For users who make use of the Request Block Extension.
__emsgparmlist			__S99emparms_t * __ptr32	For users who want to process associated messages with the dynamic allocation.
__rls	DALRLS	801C	char	Specifies the type of record level sharing (RLS) being done for a specific data set. The valid values are __RLS_NRI, __RLS_CR and __RLS_CRE. See *z/OS XL C/C++ Programming Guide* and *z/OS DFSMS Using Data Sets* for a description of these VSAM RLS/TVS access modes.

[1] If an element exceeds its maximum allowable length, it is truncated to that length.

Special behavior for POSIX C: For POSIX C programs, allocations established by the dynalloc() function persist neither after an `exec` nor in the child process after fork().

Special behavior for enhanced ASCII: When compiled ASCII, there is one input element in the `__dyn_t` structure that must contain EBCDIC text strings and there is a consideration to note with respect to retrieval of error messages related to a dynamic allocation failure. On input, any character data provided in __miscitems must be specified in the EBCDIC codeset. The __dynalloc() function does not decode the text units and convert the character data. The text units are passed directly to the system. When __emsgparmlist is specified, indicating intent to retrieve error messages using the IEFDB476 service, it should be noted that all error messages returned by the service will be in the EBCDIC codeset.

Note: The dynalloc() function has a dependency on the level of the enhanced ASCII extensions. See "Enhanced ASCII support" on page 2113 for details.

Special behavior for AMODE 64: The definitions in the `__dyn_t` structure are changed to require three of its pointer elements to be 32 bits wide. This is because the system services that work with these control structures require 31-bit addressable storage. The __miscitems are additional text units that are not already supported by elements of the `__dyn_t` structure. These are propagated by the dynalloc() function directly to into an SVC 99 call. The __rbx is propagated by the dynalloc() function directly into an SVC 99 call. The __emsgparmlist address is designed to be passed as a parameter to the IEFDB476 service, which is an AMODE 31 service, to retrieve messages associated with a dynamic allocation failure. The `__dyn_t` structure itself can be in 64-bit addressable storage. The `__dyn_t` structure must be initialized using the dyninit() macro defined in dyninit.h to ensure the proper "hidden" version indicator is used. Improper initialization of the `__dyn_t` structure will result in undefined behavior.

Returned value

If successful under MVS, the dynalloc() function returns 0.

If SVC 99 is not supported on your system, or if a text string passed to SVC 99 cannot be built from a field in *dyn_parms*, a negative value is returned.

The value -1 is returned if there is not sufficient storage to process all the text units. Otherwise, the return code is the value returned from SVC 99, and the error and information codes are found in those fields in *dyn_parms*.

For example, if you pass NULL to the dynalloc() function, the return code is nonzero.

For more information about return codes, see *z/OS MVS Programming: Authorized Assembler Services Guide*.

Example

CELEBD07

```
/* CELEBD07

   This example dynamically allocates a data set.

 */
#include <dynit.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#define ZERO 0

int main () {
   __dyn_t ip;

   dyninit(&ip);

   ip.__ddname = "mydd";                /* MYDD DD                */
   ip.__dsname = "PLIXXX.MY.DATASET";   /* DSN='PLIXXX.MY.DATASET' */
   ip.__status = __DISP_NEW;              /* DISP=(NEW,CATLG)        */
   ip.__normdisp = __DISP_CATLG;
   ip.__alcunit = __CYL;                /* SPACE=(CYL,(2,1)),      */
   ip.__primary = 2;
   ip.__secondary = 1;
   ip.__dirblk = 1;
   ip.__misc_flags = __RELEASE & __CONTIG;  /*       RLSE,CONTIG)  */
   ip.__dsorg = __DSORG_PO;             /* DCB=(DSORG=PO,          */
   ip.__recfm = _F_ + _B_ + _A_;        /*        RECFM=FBA,   */
   ip.__lrecl = 121;                    /*      LRECL=121,         */
   ip.__blksize = 12100;                /*      BLKSIZE=12100)     */

   if (dynalloc(&ip) != ZERO)
   {
       printf("Dynalloc failed with error code %d, info code %d\n",
               ip.__errcode, ip.__infocode);
   }
}
```

Related information

- “dynit.h” on page 23
- “dynfree() — Deallocate a data set”
- “dyninit() — Initialize __dyn_t structure” on page 416
- “svc99() — Access supervisor call” on page 1788

dynfree() — Deallocate a data set

Standards

Standards / Extensions	C or C++	Dependencies
C Library	both	

Format

```
#include <dynit.h>

int dynfree(__dyn_t *dyn_parms);
```

General description

Dynamically deallocates a z/OS data set in accordance with the attributes defined in *dyn_parms*.

To avoid infringing on the user's name space, this nonstandard function has two names. One name is prefixed with two underscore characters, and one name is not. The name without the prefix underscore characters is exposed only when you use LANGLVL(EXTENDED).

To use this function, you must either invoke the function using its external entry point name (that is, the name that begins with two underscore characters), or compile with LANGLVL(EXTENDED). When you use LANGLVL(EXTENDED) any relevant information in the header is also exposed.

The only fields in __dyn_t that are used by dynfree() are:

```
char *__ddname
char *__dsname
char *__member
char *__pathname
char __normdisp
char __pathndisp
char **__miscitems
```

If any other fields are specified, they will be ignored. For more information on the __dyn_t structure, see Table 21 on page 409

To dynamically deallocate a data set on z/OS, you should:
- Invoke dyninit() with a variable of type __dyn_t
- Assign values to the appropriate fields that will satisfy the svc99() request
- Invoke dynfree() with this variable.

Note: The dynfree() function has a dependency on the level of the Enhanced ASCII Extensions. See "Enhanced ASCII support" on page 2113 for details.

Returned value

If successful under z/OS, dynfree() returns 0.

If unsuccessful, dynfree() returns nonzero. dynfree() returns -1 if there is not sufficient storage to process all the text units.

Example

```
/*
   This example dynamically deallocates a data set.
 */
#include <dynit.h>

int main(void) {
⋮
  __dyn_t ip;
⋮
  dyninit(ip);
  ip.__ddname = "mydd";
⋮
  dynfree(&ip);
}
```

Related information

- "dynit.h" on page 23
- "dynalloc() — Allocate a data set" on page 407
- "dyninit() — Initialize __dyn_t structure"
- "svc99() — Access supervisor call" on page 1788

dyninit() — Initialize __dyn_t structure

Standards

Standards / Extensions	C or C++	Dependencies
C Library	both	

Format

```
#include <dynit.h>

int dyninit(__dyn_t *dyn_parms);
```

General description

Initializes the __dyn_t structure that is used to build the parameter lists that are passed to the dynalloc() function and the dynfree() function. If you do not initialize the __dyn_t structure using dyninit(), undefined behavior may result.

The __dyn_t structure is defined in the dynit.h header file. A description of the elements is found in "dynalloc() — Allocate a data set" on page 407.

Returned value

If successful under MVS, dyninit() returns 0.

If unsuccessful, dyninit() returns nonzero.

Example

CELEBD09

```
/* CELEBD09

   This example initializes a __dyn_t
    structure, called ip.

 */
#include <stdio.h>
#include <string.h>
#include <dynit.h>

main() {
  char dsn[]="USER.TEST.DATASET";
  __dyn_t ip;
  int ret;

  dyninit(&ip);
  ip.__ddname   = "TEST";
  ip.__dsname   = dsn;
  ip.__status   = __DISP_NEW;
  ip.__normdisp = __DISP_DELETE;
  ip.__alcunit  = __TRK;
  ip.__primary  = 1;
  ip.__unit     = "SYSDA  ";

  if ((ret = dynalloc(&ip)) != 0)
    printf("dynalloc() ret=%d, error code %04x, info code %04x\n",
           ret, ip.__errcode, ip.__infocode);

  else {
    dyninit(&ip);
    ip.__ddname   = "TEST";

    if ((ret = dynfree(&ip)) != 0)
      printf("dynfree() ret=%d, error code %04x, info code %04x\n",
             ret, ip.__errcode, ip.__infocode);

    else puts("success!");
  }
}
```

Related information

- "dynit.h" on page 23
- "dynalloc() — Allocate a data set" on page 407
- "dynfree() — Deallocate a data set" on page 415
- "svc99() — Access supervisor call" on page 1788

ecvt() — Convert double to string

Standards

Standards / Extensions	C or C++	Dependencies
XPG4.2 Single UNIX Specification, Version 3	both	

Format

```
#define _XOPEN_SOURCE_EXTENDED 1
#include <stdlib.h>

char *ecvt(double x, int ndigit, int *__restrict__ decpt, int *__restrict__ sign);
```

General description

The ecvt() function converts double floating-point argument values to floating-point output strings. The ecvt() function has been extended to determine the floating-point format (hexadecimal floating-point or IEEE Binary Floating-Point) of double argument values by using __isBFP().

z/OS XL C/C++ formatted output functions, including the ecvt() function, convert IEEE Binary Floating-Point infinity and NaN argument values to special infinity and NaN floating-point number output sequences. See "*fprintf* Family of Formatted Output Functions" on fprintf(), printf(), sprintf() — Format and write data for a description of the special infinity and NaN output sequences.

The ecvt() function converts *x* to a NULL-terminated string of *ndigit* digits (where *ndigit* is reduced to an unspecified limit determined by the precision of a double) and returns a pointer to the string. The high-order digit is nonzero, unless the value is 0. The low-order digit is rounded. The position of the radix character relative to the beginning of the string is stored in the integer pointed to by *decpt* (negative means left of the returned digits). The radix character is not included in the returned string. If the sign of the result is negative, the integer pointed to by *sign* is nonzero, otherwise it is 0.

The function returns a pointer to a buffer used only by the calling thread which may be overwritten by subsequent calls to ecvt(), "fcvt() — Convert double to string" on page 485 and "gcvt() — Convert double to string" on page 683.

If the converted value is out of range or is not representable, the function returns NULL.

Note: This function has been moved to the Legacy Option group in Single UNIX Specification, Version 3 and may be withdrawn in a future version. The sprintf() function is preferred for portability.

Returned value

If successful, ecvt() returns the character equivalent of *x* as specified above.

If unable to allocate the return buffer, or the conversion fails, ecvt() returns NULL.

Related information

- “stdlib.h” on page 70
- “fcvt() — Convert double to string” on page 485
- “gcvt() — Convert double to string” on page 683
- “__isBFP() — Determine application floating-point format” on page 904

encrypt() — Encoding function

Standards

Standards / Extensions	C or C++	Dependencies
XPG4 XPG4.2 Single UNIX Specification, Version 3	both	

Format

```
#define _XOPEN_SOURCE
#include <unistd.h>

void encrypt(char block[64], int edflag);
```

General description

The encrypt() function uses an array of 16 48-bit keys produced by the setkey() function to encode bytes specified by the *block* argument according to the Data Encryption Standard (DES) encryption algorithm or to decode argument bytes according to the DES decryption algorithm.

The *block* argument of encrypt() is an array of length 64 bytes containing only the bytes with numerical value of 0 and 1. The array is modified in place using keys produced by setkey(). If *edflag* is 0, the argument is encoded using the DES encryption algorithm. If *edflag* is 1 the argument is decoded using the DES decryption algorithm.

Special behavior for z/OS UNIX Services: The encrypt() function is thread-specific. Thus, for each thread from which the encrypt() function is called by a threaded application, the setkey() function must first be called from the thread to establish a DES key array for the thread.

Returned value

encrypt() returns no values.

Special behavior for z/OS UNIX Services: encrypt() will set errno to one of the following values:

Error Code
Description

EINVAL
64 byte input array contains bytes with values other than 0x00 or 0x01.

ENOMEM
If setkey() has not been called or failed to produce a DES key array for the thread from which encrypt() is called.

ENOSYS
If DES key array exists for thread from which encrypt() is called to decode data.

Note: Because encrypt() returns no values, applications wishing to check for errors should set errno to 0, call encrypt(), then test errno and, if it is nonzero, assume an error has occurred.

Related information

- "unistd.h" on page 82
- "__cnvblk() — Convert block" on page 294
- "crypt() — String encoding function" on page 345
- "setkey() — Set encoding key" on page 1551

endgrent() — Group database entry functions

Standards

Standards / Extensions	C or C++	Dependencies
XPG4.2 Single UNIX Specification, Version 3	both	

Format

```
#define _XOPEN_SOURCE_EXTENDED 1
#include <grp.h>

void endgrent(void),
struct group *getgrent (void);
void setgrent(void);
```

General description

The getgrent() function returns a pointer to the broken-out fields of a line in the group database, mapped by the **group** structure defined in the <grp.h> header file. Repeated calls to getgrent() return a pointer to the next **group** structure in the database, until End Of File (EOF), at which point a NULL pointer is returned. setgrent() interrupts this sequential search and rewinds the user database to the beginning, such that the next getgrent() returns a pointer to the first **group** structure. Use of setgrent() is optional after an End Of File (EOF), as the next getgrent() after end of file again returns a pointer to the first **group** structure. endgrent() is optionally used to close the user database when searching is complete.

The setgrent() function effectively rewinds the group database to allow repeated searches.

The endgrent() function may be called to close the group database when processing is complete.

Returned value

When first called, getgrent() returns a pointer to the next group structure in the group database. Upon subsequent calls it returns a pointer to a group structure, or it returns a NULL pointer on either End Of File (EOF) or an error. The return value may point to static data that is overwritten by each call.

There are no documented errno values.

Related information

- "grp.h" on page 32
- "getgrgid() — Access the group database by ID" on page 709
- "getgrgid_r() — Get group database entry for a group ID" on page 710
- "getgrnam() — Access the group database by name" on page 711
- "getgrnam_r() — Search group database for a name" on page 713
- "getlogin() — Get the user login name" on page 731
- "getlogin_r() — Get login name" on page 732
- "getpwent() — Get user database entry" on page 760
- "getpwnam() — Access the user database by user name" on page 760
- "getpwnam_r() — Search user database for a name" on page 762
- "getpwuid() — Access the user database by user ID" on page 763
- "getpwuid_r() — Search user database for a user ID" on page 764

endhostent() — Close the host information data set

Standards

Standards / Extensions	C or C++	Dependencies
XPG4.2 Single UNIX Specification, Version 3	both	

Format

X/Open:

```
#define _XOPEN_SOURCE_EXTENDED 1
#include <netdb.h>

void endhostent(void);
```

Berkeley sockets:

```
#define _OE_SOCKETS
#include <netdb.h>

void endhostent();
```

General description

The endhostent() function closes the local host tables, which contains information about known hosts.

You can use the **X_SITE** environment variable to specify different local host tables and override those supplied by the z/OS global resolver during initialization. For more information on these local host tables or the environment variables, see *z/OS Communications Server: IP Configuration Guide*.

Special behavior for C++: To use this function with C++, you must use the _XOPEN_SOURCE_EXTENDED 1 feature test macro.

Related information

- “netdb.h” on page 49
- “gethostbyaddr() — Get a host entry by address” on page 717
- “gethostbyname() — Get a host entry by name” on page 719
- “gethostent() — Get the next host entry” on page 721
- “sethostent() — Open the host information data set” on page 1539

endnetent() — Close network information data sets

Standards

Standards / Extensions	C or C++	Dependencies
XPG4.2 Single UNIX Specification, Version 3	both	

Format

X/Open:

```
#define _XOPEN_SOURCE_EXTENDED 1
#include <netdb.h>

void endnetent(void);
```

Berkeley sockets:

```
#define _OE_SOCKETS
#include <netdb.h>

void endnetent();
```

General description

The endnetent() function closes the *tcpip*.HOSTS.ADDRINFO data set. The *tcpip*.HOSTS.ADDRINFO data set contains information about known networks.

You can use the **X_ADDR** environment variable to specify a data set other than *tcpip*.HOSTS.ADDRINFO. For more information on these data sets and environment variables, see *z/OS Communications Server: IP Configuration Guide, SC27-3650*.

Special behavior for C++: To use this function with C++, you must use the _XOPEN_SOURCE_EXTENDED 1 feature test macro.

Related information

- “netdb.h” on page 49
- “getnetbyaddr() — Get a network entry by address” on page 741
- “getnetbyname() — Get a network entry by name” on page 742
- “getnetent() — Get the next network entry” on page 744
- “setnetent() — Open the network information data set” on page 1562

endprotoent() — Work with a protocol entry

Standards

Standards / Extensions	C or C++	Dependencies
XPG4.2 Single UNIX Specification, Version 3	both	

Format

X/Open:

```
#define _XOPEN_SOURCE_EXTENDED 1
#include <netdb.h>

void endprotoent(void);
```

Berkeley sockets:

```
#define _OE_SOCKETS
#include <netdb.h>

void endprotoent();
```

General description

The endprotoent() function closes the */etc/protocol* or the *tcpip*.ETC.PROTO data set, which contains information about the networking protocols (IP, ICMP, TCP, and UDP).

Special behavior for C++: To use this function with C++, you must use the _XOPEN_SOURCE_EXTENDED 1 feature test macro.

Related information

- "netdb.h" on page 49
- "getprotobyname() — Get a protocol entry by name" on page 756
- "getprotoent() — Get the next protocol entry" on page 759
- "setprotoent() — Open the protocol information data set" on page 1569

endpwent() — User database functions

Standards

Standards / Extensions	C or C++	Dependencies
XPG4.2 Single UNIX Specification, Version 3	both	

Format

```
#define _XOPEN_SOURCE_EXTENDED 1
#include <pwd.h>

void endpwent(void),
struct passwd *getpwent(void);
void setpwent(void);
```

General description

The getpwent() function returns a pointer to the broken-out fields of a line in the user database, mapped by the *passwd* structure defined in the <pwd.h> header file. Repeated calls to getpwent() return a pointer to the next *passwd* structure in the database, until End Of File (EOF), at which point a NULL pointer is returned. setpwent() interrupts this sequential search and rewinds the user database to the beginning, such that the next getpwent() returns a pointer to the first *passwd* structure. Use of setpwent() is optional after an End Of File (EOF), as the next getpwent() after end of file again returns a pointer to the first *passwd* structure. endpwent() is optionally used to close the user database when searching is complete.

The setpwent() function effectively rewinds the user database to allow repeated searches.

The endpwent() function may be called to close the user database when processing is complete.

Returned value

When first called, getpwent() returns a pointer to the next *passwd* structure in the user database. Upon subsequent calls it returns a pointer to a *passwd* structure, or it returns a NULL pointer on either End Of File (EOF) or an error. The return value may point to static data that is overwritten by each call.

There are no documented errno values for endpwent() and setpwent(). For getpwent(), if unsuccessful, getpwemt() sets errno to one of the following values:

Error Code
Description

EMVSSAF2ERR
The system authorization facility (SAF) or RACF Get GMAP service had an error.

EMVSSAFEXTRERR
The SAF or RACF RACROUTE EXTRACT service had an error.

Related information

- "pwd.h" on page 61
- "getgrent() — Get group database entry" on page 709
- "getgrgid() — Access the group database by ID" on page 709
- "getgrnam() — Access the group database by name" on page 711
- "getlogin() — Get the user login name" on page 731
- "getpwent() — Get user database entry" on page 760
- "getpwnam() — Access the user database by user name" on page 760
- "getpwuid() — Access the user database by user ID" on page 763

endservent() — Close network services information data sets

Standards

Standards / Extensions	C or C++	Dependencies
XPG4.2 Single UNIX Specification, Version 3	both	

Format

X/Open:

```
#define _XOPEN_SOURCE_EXTENDED 1
#include <netdb.h>

void endservent(void);
```

Berkeley sockets:

```
#define _OE_SOCKETS
#include <netdb.h>

void endservent();
```

General description

The endservent() function closes the */etc/services* or the *tcpip*.ETC.SERVICES data set, which contains information about network services. Example services are name server, File Transfer Protocol (FTP), and telnet.

Special behavior for C++: To use this function with C++, you must use the _XOPEN_SOURCE_EXTENDED 1 feature test macro.

Related information

- “netdb.h” on page 49
- “getservbyname() — Get a server entry by name” on page 770
- “getservbyport() — Get a service entry by port” on page 772
- “getservent() — Get the next service entry” on page 773
- “setservent() — Open the network services information data set” on page 1576

endutxent() — Close the utmpx database

Standards

Standards / Extensions	C or C++	Dependencies
XPG4.2 Single UNIX Specification, Version 3	both	

Format

```
#define _XOPEN_SOURCE_EXTENDED 1
#include <utmpx.h>

void endutxent(void);
```

General description

The endutxent() function closes the utmpx database for the current thread. The database may be opened by getutxent(), getutxid(), getutxline(), or pututxline().

Because the endutxent() function processes thread-specific data the endutxent() function can be used safely from a multithreaded application. If multiple threads in the same process open the database, then each thread opens the database with a different file descriptor. The thread's database file descriptor is closed when the calling thread terminates or the endutxent() function is called by the calling thread.

Programs must not reference the data passed back by getutxline(), getutxid(), getutxent(), or pututxline() after endutxent() has been called (the storage has been freed.)

After getuxline(), getutxent(), getutxid(), or pututxline(), the utmpx database is open. No other process can do pututxline() to this utmpx database until this process issues endutxent() or __utmpxname() to close the utmpx database, or this process ends. You can cause all z/OS UNIX user logins/logouts to hang if you fail to exit() or issue endutxent() or __utmpxname(), and you have the main /etc/utmpx database open in your process. endutxent() resets the name of the next utmpx file to open back to the default. If you want to do additional utmpx operations using a nonstandard utmpx file name, you must reissue __utmpxname() after closing the utmpx database with endutxent().

Returned value

endutxent() returns no values.

Related information

- "utmpx.h" on page 84
- "getutxent() — Read next entry in utmpx database" on page 793
- "getutxid() — Search by ID utmpx database" on page 795
- "getutxline() — Search by line utmpx database" on page 796
- "pututxline() — Write entry to utmpx database" on page 1354
- "setutxent() — Reset to start of utmpx database" on page 1595
- "__utmpxname() — Change the utmpx database name" on page 1971

erand48() — Pseudo-random number generator

Standards

Standards / Extensions	C or C++	Dependencies
XPG4 XPG4.2 Single UNIX Specification, Version 3	both	

Format

```
#define _XOPEN_SOURCE
#include <stdlib.h>

double erand48(unsigned short int x16v[3]);
```

General description

The drand48(), erand48(), jrand48(), lrand48(), mrand48() and nrand48() functions generate uniformly distributed pseudo-random numbers using a linear congruential algorithm and 48-bit integer arithmetic.

The functions drand48() and erand48() return nonnegative, double-precision, floating-point values, uniformly distributed over the interval [0.0,1.0). These functions have been extended so that the returned value will be in the proper floating-point format (hexadecimal or IEEE) based on the floating-point mode of the invoking thread.

The functions lrand48() and nrand48() return nonnegative, long integers, uniformly distributed over the interval [0,2**31).

The functions mrand48() and jrand48() return signed long integers, uniformly distributed over the interval [-2**31,2**31).

The erand48() function generates the next 48-bit integer value in a sequence of 48-bit integer values, X(i), according to the linear congruential formula:

```
X(n+1) = (aX(n) + c)mod(2**48)     n>=0
```

The erand48() function uses storage provided by the argument array, *x16v*[3], to save the most recent 48-bit integer value in the sequence, X(i). The erand48() function uses *x16v*[0] for the low-order (rightmost) 16 bits, *x16v*[1] for the middle-order 16 bits, and *x16v*[2] for the high-order 16 bits of this value.

The initial values of a, and c are:

```
a   = 5deece66d (base 16)
c   = b         (base 16)
```

The values a and c, may be changed by calling the lcong48() function. The initial values of a and c are restored if either the seed48() or srand48() function is called.

Special behavior for z/OS UNIX Services: You can make the erand48() function and other functions in the drand48 family thread-specific by setting the environment variable _RAND48 to the value THREAD before calling any function in the drand48 family.

If you do not request thread-specific behavior for the drand48 family, C/370 serializes access to the storage for X(n), a and c by functions in the drand48 family when they are called by a multithreaded application.

If thread-specific behavior is requested and the erand48() function is called from thread t, the erand48() function generates the next 48-bit integer value in a sequence of 48-bit integer values, X(t,i), for the thread according to the linear congruential formula:

```
X(t,n+1) = (a(t)X(t,n) + c(t))mod(2**48)     n>=0
```

The erand48() function uses storage provided by the argument array, *x16v*[3], to save the most recent 48-bit integer value in the sequence, X(t,i). The erand48() function uses *x16v*[0] for the low-order (rightmost) 16 bits, *x16v*[1] for the middle-order 16 bits, and *x16v*[2] for the high-order 16 bits of this value.

The initial values of a(t) and c(t) on the thread t are:

```
a(t)   = 5deece66d (base 16)
c(t)   = b         (base 16)
```

The values a(t) and c(t) may be changed by calling the lcong48() function from the thread t. The initial values of a(t) and c(t) are restored if either the seed48() or srand48() function is called from the thread.

Returned value

erand48() saves the generated 48-bit value, X(n+1), in storage provided by the argument array, *x16v*[3]. erand48() transforms the generated 48-bit value to a double-precision, floating-point value on the interval [0.0,1.0) and returns this transformed value.

Special behavior for z/OS UNIX Services: If thread-specific behavior is requested for the drand48 family and erand48() is called on thread `t`, erand48() saves the generated 48-bit value, X(t,n+1), in storage provided by the argument array, *x16v*[3]. erand48() transforms the generated 48-bit value to a double-precision, floating-point value on the interval [0.0,1.0) and returns this transformed value.

Related information

- "stdlib.h" on page 70
- "drand48() — Pseudo-random number generator" on page 402
- "__isBFP() — Determine application floating-point format" on page 904
- "jrand48() — Pseudo-random number generator" on page 928
- "lcong48() — Pseudo-random number initializer" on page 939
- "lrand48() — Pseudo-random number generator" on page 1010
- "mrand48() — Pseudo-random number generator" on page 1101
- "nrand48() — Pseudo-random number generator" on page 1148
- "seed48() — Pseudo-random number initializer" on page 1467
- "srand48() — Pseudo-random number initializer" on page 1718

erf(), erfc(), erff(), erfl(), erfcf(), erfcl() — Calculate error and complementary error functions

Standards

Standards / Extensions	C or C++	Dependencies
SAA XPG4 XPG4.2 C99 Single UNIX Specification, Version 3 C++ TR1 C99	both	

Format

SAA:

```
#include <math.h>

double erf(double x);
double erfc(double x);
```

Compiler option: LANGLVL(EXTENDED), LANGLVL(SAA), or LANGLVL(SAA2)

XPG4:

```
#define _XOPEN_SOURCE
#include <math.h>

double erf(double x);
double erfc(double x);
```

C99:

```
#define _ISOC99_SOURCE
#include <math.h>

float erff(float x);
long double erfl(long double x);
float erfcf(float x);
long double erfcl(long double x)
```

C++ TR1 C99:

```
#define _TR1_C99
#include <math.h>

float erf(float x);
long double erf(long double x);
float erfc(float x);
long double erfc(long double x);
```

General description

Calculates the error and complementary error functions:

$$2\pi^{-1/2} \int_{0}^{x} e^{-t^2} dt$$

Because the erfc() function calculates the value of `1.0 - erf(x)`, it is used in place of erf() for large values of *x*.

Note: The following table shows the viable formats for these functions. See "IEEE binary floating-point" on page 94 for more information about IEEE Binary Floating-Point.

Function	SPC	Hex	IEEE
erf	X	X	X
erff		X	X
erfl		X	X
erfc	X	X	X
erfcf		X	X
erfcl		X	X

Returned value

Both erf() and erfc() return the calculated value.

If the correct value would cause underflow, 0 is returned and the value of the macro ERANGE is stored in errno. A range error is returned if *x* is too large.

Special behavior for IEEE: erf() and erfc() are always successful.

Requirements

This function is exposed by specifying on the compile step either the specific option LANGLVL(LONGLONG) or the general option LANGLVL(EXTENDED).

Example

CELEBE01

```
/* CELEBE01

  This example uses &erf. and &erfc. to compute the error
   function of two numbers.
```

```
 */
#include <stdio.h>
#include <math.h>

double smallx, largex, value;

int main(void)
{
   smallx = 0.1;
   largex = 10.0;

   value = erf(smallx);         /* value = 0.112463 */
   printf("Error value for 0.1: %f\n", value);

   value = erfc(largex);        /* value = 2.088488e-45 */
   printf("Error value for 10.0: %e\n", value);
}
```

Output

```
Error value for 0.1: 0.112463
Error value for 10.0: 2.088488e-45
```

Related information

- “math.h” on page 44
- “gamma() — Calculate gamma function” on page 682
- “j0(), j1(), jn() — Bessel functions of the first kind” on page 930
- “y0(), y1(), yn() — Bessel functions of the second kind” on page 2098

erfd32(), erfd64(), erfd128(), erfcd32(), erfcd64(), erfcd128() - Calculate error and complementary error functions

Standards

Standards / Extensions	C or C++	Dependencies
C/C++ DFP	both	z/OS V1.10

Format

```
#define __STDC_WANT_DEC_FP__
#include <math.h>

_Decimal32  erfd32(_Decimal32 x);
_Decimal64  erfd64(_Decimal64 x);
_Decimal128 erfd128(_Decimal128 x);
_Decimal32  erf(_Decimal32 x);      /* C++ only */
_Decimal64  erf(_Decimal64 x);      /* C++ only */
_Decimal128 erf(_Decimal128 x);     /* C++ only */

_Decimal32  erfcd32(_Decimal32 x);
_Decimal64  erfcd64(_Decimal64 x);
_Decimal128 erfcd128(_Decimal128 x);
_Decimal32  erfc(_Decimal32 x);     /* C++ only */
_Decimal64  erfc(_Decimal64 x);     /* C++ only */
_Decimal128 erfc(_Decimal128 x);    /* C++ only */
```

General description

Calculates the error and complementary error functions:

$$2\pi^{-1/2} \int_{0}^{x} e^{-t^2} dt$$

Because the erfc() function calculates the value of 1.0 - erf(*x*), it is used in place of erf() for large values of *x*.

These functions work in IEEE decimal floating-point format. See “IEEE decimal floating-point” on page 95 for more information.

Note: To use IEEE decimal floating-point, the hardware must have the Decimal Floating-Point Facility installed.

Returned value

erf() and erfc() are always successful.

Example

CELEBE12

```
/* CELEBE12

   This example illustrates the erfd32() and erfcd32() functions.

*/

#define __STDC_WANT_DEC_FP__
#include <stdio.h>
#include <math.h>

_Decimal32 smallx, largex, value;

int main(void)
{
   smallx = 0.1DF;
   largex = 10.0DF;

   value = erfd32(smallx);
   printf("Error value for 0.1: %Hf\n", value);

   value = erfcd32(largex);
   printf("Error value for 10.0: %He\n", value);
}
```

Related information

- “math.h” on page 44

__err2ad() — Return address of reason code of last failure

Standards

Standards / Extensions	C or C++	Dependencies
z/OS UNIX	both	

Format

```
int *__err2ad(void);
```

General description

The __err2ad() function returns the address of the errno2. The errno2 may be set by the z/OS XL C/C++ runtime library, z/OS UNIX callable services or other callable services.

__err2ad() provides assistance in diagnosing problems by allowing an application to reset the errno2 value prior to calling a function.

For more information about __errno2(), see *z/OS XL C/C++ Runtime Library Reference*.

Returned value

__err2ad() is always successful.

Related information

- "__errno2() — Return reason code information"

__errno2() — Return reason code information

Standards

Standards / Extensions	C or C++	Dependencies
z/OS UNIX	both	

Format

```
int __errno2(void);
```

General description

The __errno2() function can be used when diagnosing application problems. This function enables z/OS XL C/C++ application programs to access additional diagnostic information, errno2 (errnojr), associated with errno. The errno2 may be set by the z/OS XL C/C++ runtime library, z/OS UNIX callable services or other callable services. The errno2 is intended for diagnostic display purposes only and it is not a programming interface. The __errno2() function is not portable.

Note: Not all functions set errno2 when errno is set. In the cases where errno2 is not set, the __errno2() function may return a residual value. You may use the __err2ad() function to clear errno2 to reduce the possibility of a residual value being returned.

Returned value

The __errno2() function is always successful. The returned value is intended for diagnostic display purposes only.

The returned value may input to the BPXMTEXT utility to produce detailed information about the reported error if available.

For more information about the return value, see the *z/OS UNIX System Services Command Reference*, *z/OS UNIX System Services Programming: Assembler Callable Services Reference, SA23-2281*, and *z/OS Language Environment Debugging Guide*.

Example

CELEBE02

```
/* CELEBE02

   The following example's output only occurs
   if the buffer is flushed.

 */
#include <errno.h>
#include <stdio.h>
int   main(void) {
   FILE *f;
   f = fopen("notafile","r");
   if (f==NULL) {
      perror("fopen() failed");
      printf("__errno2 = %08x\n", __errno2());
   }
   return(0);
}
```

```
Sample output of routine using __errno2(), CELEBE02:

fopen() failed: EDC5129I No such file or directory. __errno2 = 05620062
```

CELEBE08

```
#pragma runopts(posix(on))
#define _EXT
#include <stdio.h>
#include <errno.h>
#include <stdlib.h>

int main(void){
  FILE *fp;
  /* add errno2 to perror message */
  setenv("_EDC_ADD_ERRNO2", "1", 1);
  fp = fopen("testfile.dat", "r");
  if (fp == NULL)
    perror("fopen() failed");
   return 0;
}

Sample output of routine using _EDC_ADD_ERRNO2, CELEBE08:

fopen() failed: EDC5129I No such file or directory. (errno2=0x05620062)
```

CELEBE09

```
#pragma runopts(posix(on))
#define _EXT
#include <stdio.h>
#include <errno.h>

int main(void){
  FILE *f;
  f = fopen("testfile.dat", "r");
  if (f == NULL){
    perror("fopen() failed");
    printf("__errno2 = %08x\n", __errno2());
  }

  /* reset errno2 to zero */
  *__err2ad() = 0x0;
  printf("__errno2 = %08x\n", __errno2());

  f = fopen("testfile.dat", "r");
  if (f == NULL){
    perror("fopen() failed");
    printf("__errno2 = %08x\n", __errno2());
  }

  return 0;
}
```

For more information about _EDC_ADD_ERRNO2 , see *z/OS XL C/C++ Programming Guide*. For more information about __err2ad() , see *z/OS XL C/C++ Runtime Library Reference*.

Related information

- "errno.h" on page 23
- "__err2ad() — Return address of reason code of last failure" on page 431

__etoa() — EBCDIC to ISO8859-1 string conversion

Standards

Standards / Extensions	**C or C++**	**Dependencies**
z/OS UNIX	both	

Format

```
#include <unistd.h>

int __etoa(char *string);
```

General description

The __etoa() function converts an EBCDIC character string *string* to its ISO8859-1 equivalent. The conversion is performed using the codeset page associated with the current locale. The input character string up to, but not including, the NULL character is changed from the current locale to an ISO8859-1 representation.

The argument *string* points to the EBCDIC character string to be converted to its ISO8859-1 equivalent.

Returned value

If successful, __etoa() converts the input EBCDIC string to its equivalent ISO8859-1 value, and returns the length of the converted string.

If unsuccessful, __etoa() returns -1 and sets errno to one of the following values. (This function internally may call iconv_open() and iconv(). The errnos returned by these functions are propagated without modification.)

Error Code
Description

EINVAL
The current locale does not describe a single-byte character set.

ENOMEM
There is insufficient storage to complete the conversion process.

Related information

- “sys/msg.h” on page 74
- “unistd.h” on page 82
- “iconv() — Code conversion” on page 824
- “iconv_open() — Allocate code conversion descriptor” on page 829

__etoa_l() — EBCDIC to ISO8859-1 conversion operation

Standards

Standards / Extensions	C or C++	Dependencies
z/OS UNIX	both	

Format

```
#include <unistd.h>

int __etoa_l(char *bufferptr, int leng);
```

General description

The __etoa_l() function converts *leng* EBCDIC bytes in the buffer pointed to by *bufferptr* to their ISO8859-1 equivalent. The conversion is performed using the codeset page associated with the current locale.

The argument *bufferptr* points to a buffer containing the EBCDIC bytes to be converted to their ISO8859-1 equivalent. The input buffer is treated as a sequence of bytes, and all bytes in the input buffer are converted, including any imbedded NULLs.

Returned value

If successful, __etoa_l() converts the input EBCDIC bytes to their equivalent ISO8859-1 value, and returns the number of bytes converted.

If unsuccessful, __etoa_l() returns -1 and sets errno to one of the following values. (This function may internally call iconv_open() and iconv(). The errnos returned by these functions are propagated without modification.)

Error Code
  Description

EINVAL
  The current locale does not describe a single-byte character set.

ENOMEM
  There is insufficient storage to complete the conversion process.

Related information

- “sys/msg.h” on page 74
- “unistd.h” on page 82
- “iconv() — Code conversion” on page 824
- “iconv_open() — Allocate code conversion descriptor” on page 829

exec functions

Standards

Standards / Extensions	C or C++	Dependencies
POSIX.1 XPG4 XPG4.2 Single UNIX Specification, Version 3	both	

Format

```
#define _POSIX_SOURCE
#include <unistd.h>
extern char **environ;

int execl(const char *path, const char *arg, ..., NULL);
int execle(const char *path, const char *arg, ..., NULL, char *const envp[]);
int execlp(const char *file, const char *arg, ..., NULL);
int execv(const char *path, char *const argv[]);
int execve(const char *path, char *const argv[], char *const envp[]);
int execvp(const char *file, char *const argv[]);
```

Note: Although POSIX.1 does not require that the unistd.h include file be included, it is recommended that you include it for portability.

General description

All exec functions run a new program by replacing the current process image with a new process image obtained from a file in the HFS (hierarchical file system).

For information on specifying names for MVS data sets and HFS files, see *z/OS XL C/C++ Programming Guide*.

A successful exec function never returns control because the calling process is overwritten with the new process.

The argument *path* is a string giving the absolute or relative path name of a file. This file contains the image of the process to be run.

file is a string that is used in determining the path name of the file containing the image of the process to be run. If *file* contains a slash character (/), it is assumed to be the absolute or relative path name of the file. If *file* does not contain a slash, the system searches for the given file name under the list of directories given by the PATH environment variable. The system checks under directories in the order they appear in the PATH variable, and executes the first file whose name matches the *file* string. The file must reside in the HFS.

The exec functions use the following environment variables:

STEPLIB

Supports the creation and propagation of a STEPLIB environment to the new process image. The following are the accepted values for the STEPLIB environment variable and the actions taken for each value:

- STEPLIB=NONE. No Steplib DD is to be created for the new process image.

- STEPLIB=CURRENT. The TASKLIB, STEPLIB or JOBLIB DD data set allocations that are active for the calling task at the time of the call to exec() are propagated to the new process image, if they are found to be cataloged. Uncataloged data sets are not propagated to the new process image.
- STEPLIB=Dsn1:Dsn2:,...DsnN. The specified data sets, Dsn1:Dsn2:...DsnN, are built into a STEPLIB DD in the new process image.

  Note: The actual name of the DD is not STEPLIB, but is a system-generated name that has the same effect as a STEPLIB DD. The data sets are concatenated in the order specified. The specified data sets must follow standard MVS data set naming conventions. Data sets found to be in violation of this standard are ignored. If the data sets do follow the standard, but:
  - The caller does not have the proper security access to a data set
  - A data set is uncataloged or is not in load library format

  then the data set is ignored. Because the data sets in error are ignored, the executable file may run without the proper STEPLIB environment. If a data set is in error due to improper security access, a X'913' abend is generated. The dump for this abend can be suppressed by your installation.

If the STEPLIB environment variable is not specified, the exec() default behavior is the same as if STEPLIB=CURRENT were specified.

If the program to be invoked is a set-user-ID or set-group-ID file and the user-ID or group-ID of the file is different from that of the current process image, the data sets to be built into the STEPLIB environment for the new process image must be found in the system sanction list for set-user-id and set-group-id programs. Only those data sets that are found in the sanction list are built into the STEPLIB environment for the new process image. For detailed information regarding the sanction list, and for information on STEPLIB performance considerations, see *z/OS UNIX System Services Planning, GA32-0884*.

_BPX_JOBNAME

Used to change the jobname of the new process image. The jobname change is allowed only if the invoker has appropriate privileges and is running in an address space created by fork. If these conditions are not met, the environment variable is ignored. Accepted values are strings of 1–8 alphanumeric characters. Incorrect specifications are ignored.

_BPX_ACCT_DATA

Used to change the account data of the new process image. Rules for specifying account data:
- Up to 142 actual account data characters are allowed, including any commas
- Sub-parameters must be separated by commas.
- There is no restriction on the character set.
- If the account data is greater than 142 characters, the data is ignored.

_BPXK_JOBLOG

The _BPXK_JOBLOG environment variable can be used to specify that WTO messages are to be written to an open HFS job log file. The following are the allowable values:

Value Description

nn Job log messages are to be written to open file descriptor nn.

STDERR
Job log messages are to be written to the standard error file descriptor, 2.

None Job log messages are not to be written. This is the default.

The file that is used to capture messages can be changed at any time by calling the oe_env_np service (BPX1ENV) and specifying _BPXK_JOBLOG with a different file descriptor.

Message capturing is turned off if the specified file descriptor is marked for close on a fork or exec.

Message capturing is process-related. All threads under a given process share the same job log file. Message capturing may be initiated by any thread under that process.

Multiple processes in a single address space can each have different files active as the JOBLOG file; some or all of them can share the same file; and some processes can have message capturing active while others do not.

Only files that can be represented by file descriptors may be used as job log files; MVS data sets are not supported.

Message capturing will be propagated on a fork() or spawn(). In the case where a file descriptor was specified, the physical file must be the same for message capturing to continue in the forked or spawned process. If STDERR was specified, the file descriptor may be re-mapped to a different physical file.

Message capturing may be overridden on exec() or spawn() by specifying the _BPXK_JOBLOG environment variable as a parameter to the exec() or spawn().

Message capturing will only work in forked (BPXAS) address spaces.

Note: This is not true joblog support, messages that would normally go to the JESYSMSG data set are captured, but messages that go to JESMSGLG are not captured.

Special behavior for XPG4: If this file is not a valid executable object, the execlp() and execvp() functions invoke **/bin/sh** with the invoker's path name and the rest of the input arguments. It is similar to invoking:

```
execl("/bin/sh",
      "sh",
      "--",
      fully_expanded_pathname,
      arg1, arg2, ..., argn,
      NULL
     );
```

where arg1, arg2, ..., argn are the caller's arguments to execlp() or execvp(), and fully_expanded_pathname is the path name of the shell script found by searching the directories in the current PATH.

arg, ..., NULL is a series of pointers to NULL-terminated character strings specifying arguments for the process being invoked. If the new process is a main(), these strings are stored in an array, and a pointer to the array is passed in the *argv* parameter. The first argument is required, and it should point to a string containing the name of the file that is associated with the process that exec is starting. A NULL pointer must follow the last argument string pointer.

argv[ ] is a pointer to an array of pointers to NULL-terminated character strings. There must be a NULL pointer after the last character string to mark the end of the array. These strings are used as arguments for the process being invoked. *argv*[0] should point to a string containing the name of a file associated with the process being started by exec. *envp*[] is a pointer to an array of pointers to NULL-terminated character strings. There must be a NULL pointer after the last character string to mark the end of the array. The strings of *envp* provide the environment variables for the new process.

All the forms of exec functions provide a way to locate the file containing the new process you want to run and a collection of arguments that should be passed to the new process. Each form of exec has its own method for specifying this information.

Some exec calls explicitly pass an environment using an *envp* argument. In versions where an environment is not passed explicitly—execl(), execlp(), execv(), and execvp()—the system uses the entire environment of the caller. The caller's environment is assumed to be the *environment variables* that the *external variable **environ* points to.

The variable **ARG_MAX**, obtained from z/OS UNIX services by an invocation of `sysconf(`*`_SC_ARG_MAX`*`)`, specifies the maximum number of bytes that can be used for arguments and environment variables passed to the process being invoked. The number of bytes includes the NULL terminator on each string.

A process started by an exec function has all of the open file descriptors that were present in the caller, except for those files opened with the close-on-exec flag FD_CLOEXEC. See “fcntl() — Control open file descriptors” on page 474 for more information about this flag. In file descriptors that remain open, all attributes remain unchanged (including file locks).

Directory streams that are open in the calling process image are closed in the new process image.

The state of conversion descriptors and message catalog descriptors is undefined.

Signals set to be ignored in the caller, `SIG_IGN`, are set to be ignored in the new process image. Be careful to take care of signals that are being ignored. Although sigaction() specifying a handler is not passed by, `SIG_IGN` is. Blocking of signals is also passed by. All other signals are set to the default action, `SIG_DFL`, in the new process image, no matter how the caller handled such signals.

The real user ID (UID), real group ID (GID), and supplementary group IDs of the new process are the same as those of the caller. If the set-user-ID mode bit of the program file is on, the effective user ID of the new process is set to the file's owner. Similarly, if the set-group-ID mode bit of the program file is on, the effective group ID of the new process is set to the file's group. The effective user ID of the new process image is saved as the saved set-user-ID, and the effective group ID of the new process image is saved as the saved set-group-ID.

Any shared memory segments attached to the calling process image will not be attached to the new process image, see “shmat() — Shared memory attach operation” on page 1598. Any shared memory segments attached to the calling process image will be detached (that is, the value of shm_nattch decremented by one). If this is the last thread attached to the shared memory segment and a shmctl() RMID has been issued, the segment will be removed from the system.

Special behavior for XPG4.2: Interval timers are preserved across an exec.

The new process also inherits the following from the caller:
- Controlling terminal (**XPG4.2**)
- Nice value (see "nice() — Change priority of a process" on page 1145) (**XPG4**)
- semadj values (see "semop() — Semaphore operations" on page 1488) (**XPG4**)
- Process ID
- Parent process ID
- Process group ID
- Resource limits (see "setrlimit() — Control maximum resource consumption" on page 1573 and "ulimit() — Get or set process file size limits" on page 1942) (**XPG4.2**)
- Session membership
- Time left until an alarm clock signal
- Working directory
- Root directory
- File mode creation mask
- File size limit (see "ulimit() — Get or set process file size limits" on page 1942) (**XPG4**)
- Process signal mask
- Pending signals
- tms_utime, tms_stime, tms_cutime, and tms_cstime. See "times() — Get process and child process times" on page 1879 for more about these qualities.

A successful exec function automatically opens the specified program file, and updates the access time st_atime for that file. The program file is closed automatically after the program has been read from the file. The precise time of this close operation is undefined.

Special behavior for z/OS UNIX Services:

1. A prior loaded copy of an HFS program in the same address space is reused under the same circumstances that apply to the reuse of a prior loaded MVS unauthorized program from an unauthorized library by the MVS XCTL service with the following exceptions:
   - If the calling process is in Ptrace debug mode, a prior loaded copy is not reused.
   - If the calling process is not in Ptrace debug mode, but the only prior loaded usable copy found of the HFS program is in storage modifiable by the caller, the prior copy is not reused.
2. If the specified file name represents an external link or a sticky bit file, the program is loaded from the caller's MVS load library search order. For an external link, the external name is only used if the name is eight characters or less, otherwise the caller receives an error from the loadhfs service. For a sticky bit program, the file name is used if it is eight characters or less. Otherwise, the program is loaded from the HFS, and the following restriction exists for sticky bit programs that have the set-user-ID or set-group-ID attribute:

   If the program is found in the MVS program search order, the MVS program name must have a BPX.STICKYSUG.program_name resource profile defined in the RACF FACILITY class. See the*z/OS UNIX System Services Planning* book for details on defining the resource profile. Failure to follow this restriction will cause the abend EC6 with code xxxxE055.

3. If the calling task is in a WLM enclave, the resulting task in the new process image is joined to the same WLM enclave. This allows WLM to manage the old and new process images as one 'business unit of work' entity for system accounting and management purposes.

Note: If you are expecting this function to take advantage of the z/OS UNIX magic number support, the Language Environment runtime option to POSIX(ON) must have been set when the process was initialized. Attempting to use magic number support with a process initialized with POSIX(OFF) may produce undesirable effects. See *z/OS UNIX System Services Planning, GA32-0884* and *z/OS V2R2.0 UNIX System Services User's Guide, SA23-2279* for details and uses of the z/OS UNIX magic number.

Returned value

If successful, an exec function never returns control because the calling process is overwritten with the new process.

If unsuccessful, an exec function returns -1 and sets errno to one of the following values:

Error Code
Description

E2BIG The combined argument list and environment list of the new process has more bytes than the system-defined length. See "sysconf() — Determine system configuration options" on page 1798 for information about the system-defined length.

EACCES
The process did not have appropriate permissions to run the specified file, for one of these reasons:

- The process did not have permission to search a directory named in your *path*.
- The process did not have execute permission for the file to be run.
- The system cannot run files of this type.

EFAULT
A bad address was received as an argument of the call, or the user exit program checked.

Consult Reason Code to determine the exact reason the error occurred. The following reason code can accompany the return code: JRExecParmErr and JRExitRtnError.

EINVAL
The new process image file has the appropriate permission and has a recognized format, but the system does not support execution of a file with this format.

ELOOP
A loop exists in symbolic links. This error is issued if the number of symbolic links detected in the resolution of the *path* or *file* argument is greater than POSIX_SYMLOOP (a value defined in the limits.h header file)

ELEMULTITHREAD
The exec function was invoked from a multithreaded environment.

EMVSSAF2ERR
The executable file is a set-user-ID or set-group-ID file, and the file owner's UID or GID is not defined to RACF.

ENAMETOOLONG
All or part of the file name is too long. This can happen if:

- A *path* or *file* argument exceeds the value of **PATH_MAX**, or an element of your *path* exceeds **PATH_MAX**.
- Any *pathname* component is greater than **NAME_MAX**, and _POSIX_NO_TRUNC is in effect.
- The length of a path name string substituted for a symbolic link in the *path* argument exceeds **PATH_MAX**.

The **PATH_MAX** and **NAME_MAX** values are determined with pathconf().

ENOENT
One or more *pathname* components in *path* or *file* does not exist. This error is also issued if *path* or *file* is a NULL string.

ENOEXEC
The new process image file has the appropriate access permission but has an unrecognized format. This errno can be returned from any one of the exec family of functions, except for execlp() and execvp().

Note: Reason codes further qualify the errno. For most of the reason codes, see *z/OS UNIX System Services Messages and Codes*.

For ENOEXEC, the reason codes are:

Reason Code	Explanation
X'xxxx0C27'	The target HFS file is not in the correct format to be an executable file.
X'xxxx0C31'	The target HFS file is built at a level that is higher than that supported by the running system.

ENOMEM
The new process requires more memory than is permitted by the operating system.

ENOTDIR
A directory component of *path* or *file* is not really a directory.

Example

CELEBE03

```
/* CELEBE03

   This example runs a program, using the execl() function.

 */
#define _POSIX_SOURCE
#include <stdio.h>
#include <sys/wait.h>            /*FIX: used be <wait.h>*/
#include <sys/types.h>
#include <unistd.h>

main() {
  pid_t pid;
  int status;
```

```
   if ((pid = fork()) == 0) {
     execl("/bin/false", NULL);
     perror("The execl() call must have failed");
     exit(255);
   }
   else {
     wait(&status);
     if (WIFEXITED(status))
       printf("child exited with status of %d\n", WEXITSTATUS(status));
     else
       puts("child did not exit successfully\n");
   }
 }
```

Output

```
child exited with status of 1
```

Related information

- "limits.h" on page 39
- "signal.h" on page 63
- "unistd.h" on page 82
- "alarm() — Set an alarm" on page 156
- "chmod() — Change the mode of a file or directory" on page 272
- "_exit() — End a process and bypass the cleanup" on page 445
- "fcntl() — Control open file descriptors" on page 474
- "fork() — Create a new process" on page 571
- "getrlimit() — Get current or maximum resource consumption" on page 765
- "nice() — Change priority of a process" on page 1145
- "putenv() — Change or add an environment variable" on page 1348
- "semop() — Semaphore operations" on page 1488
- "setuid() — Set the effective user ID" on page 1592
- "shmat() — Shared memory attach operation" on page 1598
- "sigaction() — Examine or change a signal action" on page 1611
- "sigpending() — Examine pending signals" on page 1650
- "sigprocmask() — Examine or change a thread" on page 1651
- "stat() — Get file information" on page 1720
- "system() — Execute a command" on page 1805
- "times() — Get process and child process times" on page 1879
- "ulimit() — Get or set process file size limits" on page 1942
- "umask() — Set and retrieve file creation mask" on page 1947

exit() — End program

Standards

Standards / Extensions	C or C++	Dependencies
ISO C POSIX.1 XPG4 XPG4.2 C99 C11 Single UNIX Specification, Version 3	both	

Format

```
#include <stdlib.h>

__noreturn__ void exit(int status);
```

General description

The exit() function:

1. Calls all functions registered with the atexit() function, and destroys C++ objects with static storage duration, all in last-in-first-out (LIFO) order. C++ objects with static storage duration are destroyed in the reverse order of the completion of their constructor. (Automatic objects are not destroyed as a result of calling exit().)

   Functions registered with atexit() are called in the reverse order of their registration. A function registered with atexit(), before an object obj1 of static storage duration is initialized, will not be called until obj1's destruction has completed. A function registered with atexit(), after an object obj2 of static storage duration is initialized, will be called before obj2's destruction starts.
2. Flushes all buffers, and closes all open files.
3. All files opened with tmpfile() are deleted.
4. Returns control to the host environment from the program.

Process termination in _exit() is equivalent to program termination in exit().

The argument *status* can have a value from 0 to 255 inclusive or be one of the macros EXIT_SUCCESS or EXIT_FAILURE. The value of EXIT_SUCCESS is defined in `stdlib.h` as 0; the value of EXIT_FAILURE is 8.

This function is also available to C applications in a stand-alone Systems Programming C (SPC) Environment.

In a POSIX C program, exit() returns control to the kernel with the value of *status*. The kernel then performs normal process termination.

POSIX-level thread cleanup routines are *not* executed. These includes cleanup routines created with pthread_cleanup_push() and destructor routines created with pthread_key_create().

Special behavior for C++: If exit() is called in a z/OS XL C++ program, the program terminates without leaving the current block, and therefore destructors are not called for local (automatic) variables. Destructors for initialized static objects will be called in the reverse order of the completion of their constructors.

Functions registered with atexit() are called in the reverse order of their registration. A function registered with atexit(), before an object obj1 of static storage duration is initialized, will not be called until obj1's destruction has completed. A function registered with atexit(), after an object obj2 of static storage duration is initialized, will be called before obj2's destruction starts.

Returned value

exit() returns no values.

exit() returns control to its host environment, with the returned value `status`.

For example, if program A invokes program B using a call to the system() function, and program B calls the exit() function, then program B returns to its host environment, which is program A.

Example

```
/* This example flushes all buffers, closes any open files, and ends the
   program if it cannot open the file myfile.
 */
#include <stdio.h>
#include <stdlib.h>

FILE *stream;

int main(void)
{
⋮
   if ((stream = fopen("myfile.dat", "r")) == NULL)
   {
      printf("Could not open data file\n");
      exit(EXIT_FAILURE);
      }
}
```

Related information

- “System Programming C (SPC) Facilities” in *z/OS XL C/C++ Programming Guide*
- “Using Runtime User Exits” in *z/OS XL C/C++ Programming Guide*
- “stdlib.h” on page 70
- “abort() — Stop a program” on page 101
- “atexit() — Register program termination function” on page 199
- “_exit() — End a process and bypass the cleanup”
- “_Exit() — Terminate a process” on page 447
- “signal() — Handle interrupts” on page 1643
- “wait() — Wait for a child process to end” on page 1992
- “waitpid() — Wait for a specific child process to end” on page 1997

_exit() — End a process and bypass the cleanup

Standards

Standards / Extensions	C or C++	Dependencies
POSIX.1 XPG4 XPG4.2 Single UNIX Specification, Version 3	both	

Format

```
#define _POSIX_SOURCE
#include <unistd.h>

void _exit(int status);
```

General description

Ends the current process and makes an exit status value for the process available to the system.

The argument *status* specifies a return status for the process that is ending. Ending the process has the following results:

- _exit() closes all open file descriptors and directory streams in the caller.
- If the caller's parent is currently suspended because of wait() or waitpid(), the low-order 8 bits of *status* become available to the parent. For a discussion on accessing those 8 bits, refer to “waitpid() — Wait for a specific child process to end” on page 1997.
- If the caller's parent is not currently suspended because of wait() or waitpid(), _exit() saves the *status* value so that it can be returned to the parent if the parent calls wait() or waitpid().
- A SIGCHILD signal is sent to the parent process.
- If the process calling _exit() is a controlling process, the SIGHUP signal is sent to each process in the foreground process group of the controlling terminal belonging to the caller.
- If the process calling _exit() is a controlling process, _exit() disassociates the associated controlling terminal from the session. A new controlling process can then acquire the terminal.
- Exiting from a process does not end its child processes directly. The SIGHUP signal may end children in some cases. Children that survive when a process ends are assigned a new parent process ID. The new parent process ID is always 1, indicating the root ancestor of all processes.
- If a process ends and orphans a process group and if a member of that group is stopped, each member of the group is sent a SIGHUP signal, followed by a SIGCONT signal.
- All threads are ended, and their resources cleaned up. (*Threads* are MVS tasks that call a z/OS UNIX callable service.) POSIX-level thread cleanup routines are *not* executed. These include cleanup routines created with pthread_cleanup_push() and destructor routines created with pthread_key_create().

These results occur whenever a process ends. _exit() does not cause C runtime library cleanup to be performed; therefore, stream buffers are not necessarily flushed.

Note: If _exit() is issued from a TSO/E address space, it ends the calling task and all its subtasks.

Special behavior for C++: If _exit() is called in a C++ program, the program terminates without leaving the current block, and destructors are not called for local (automatic) variables. In addition, unlike exit(), destructors for global (static) variables are not called.

Returned value

_exit() is always successful and returns no values.

No value is stored in errno for this function.

Example

CELEBE05

```
/* CELEBE05

   This example ends a process.

 */
#define _POSIX_SOURCE
```

```
#include <unistd.h>
#include <stdio.h>

main() {
  puts("Remember that stream buffers are not automatically");
  puts("flushed before _exit()!");
  fflush(NULL);
  _exit(0);
}
```

Output

```
Remember that stream buffers are not automatically
flushed before _exit()!
```

Related information

- "stdlib.h" on page 70
- "unistd.h" on page 82
- "abort() — Stop a program" on page 101
- "atexit() — Register program termination function" on page 199
- "close() — Close a file" on page 288
- "exit() — End program" on page 443
- "_Exit() — Terminate a process"
- "fork() — Create a new process" on page 571
- "sigaction() — Examine or change a signal action" on page 1611
- "signal() — Handle interrupts" on page 1643
- "wait() — Wait for a child process to end" on page 1992

_Exit() — Terminate a process

Standards

Standards / Extensions	C or C++	Dependencies
C99 C11 Single UNIX Specification, Version 3 C++ TR1 C99	both	z/OS V1R7

Format

```
#define _ISOC99_SOURCE
#include <stdlib.h>

__noreturn__ void _Exit(int status);
```

General description

When running POSIX(OFF), the _Exit() function is equivalent to exit() with the exception that it does not run atexit() registered routines or signal handlers registered using signal().

When running POSIX(ON), the _Exit() function is equivalent to _exit().

Returned value

The _Exit() function does not return to its caller.

Related information

- “stdlib.h” on page 70
- “exit() — End program” on page 443
- “_exit() — End a process and bypass the cleanup” on page 445

exp(), expf(), expl() — Calculate exponential function

Standards

Standards / Extensions	C or C++	Dependencies
ISO C POSIX.1 XPG4 XPG4.2 ISO/ANSI C++ C99 Single UNIX Specification, Version 3 C++ TR1 C99	both	

Format

```
#include <math.h>

double exp(double x);
float exp(float x);                   /* C++ only */
long double exp(long double x);       /* C++ only */
float expf(float x);
long double expl(long double x);
```

General description

Calculates the exponent of *x*, defined as *e**x*, where *e* equals 2.17128128....

Note: These functions work in both IEEE Binary Floating-Point and hexadecimal floating-point formats. See “IEEE binary floating-point” on page 94 for more information about IEEE Binary Floating-Point.

Returned value

If successful, the function returns the calculated value.

If an overflow occurs, the function returns HUGE_VAL. If an underflow occurs, it returns 0. Both overflow and underflow set errno to ERANGE.

Example

CELEBE06

```
/* CELEBE06

   This example calculates y as the exponential function of x.

 */
#include <math.h>
#include <stdio.h>

int main(void)
{
   double x, y;
   x = 5.0;
   y = exp(x);

   printf("exp( %f ) = %f\n", x, y);
}
```

Output

```
exp( 5.000000 ) = 148.413159
```

Related information

- "math.h" on page 44
- "log(), logf(), logl() — Calculate natural logarithm" on page 990
- "log10(), log10f(), log10l() — Calculate base 10 logarithm" on page 1000
- "pow(), powf(), powl() — Raise to power" on page 1193

expd32(), expd64(), expd128() — Calculate exponential function

Standards

Standards / Extensions	C or C++	Dependencies
C/C++ DFP	both	z/OS V1.8

Format

```
#define __STDC_WANT_DEC_FP__
#include <math.h>

_Decimal32  expd32(_Decimal32 x);
_Decimal64  expd64(_Decimal64 x);
_Decimal128 expd128(_Decimal128 x);
_Decimal32  exp(_Decimal32 x);     /* C++ only */
_Decimal64  exp(_Decimal64 x);     /* C++ only */
_Decimal128 exp(_Decimal128 x);    /* C++ only */
```

General description

Calculates the exponent of x, defined as e***x*, where e equals 2.17128128....

Notes:

1. To use IEEE decimal floating-point, the hardware must have the Decimal Floating-Point Facility installed.
2. These functions work in IEEE decimal floating-point format. See "IEEE binary floating-point" on page 94 for more information

Returned value

If successful, the function returns the calculated value.

If an overflow occurs, the function returns HUGE_VAL_D32, HUGE_VAL_D64, or HUGE_VAL_D128. If an underflow occurs, it returns 0. Both overflow and underflow set errno to ERANGE.

Example

```
/* CELEBE11

   This example illustrates the expd64() function.

*/
```

```
#define  __STDC_WANT_DEC_FP__
#include <math.h>
#include <stdio.h>

int main(void)
{
   _Decimal64 x, y;

   x = 5.0DD;
   y = expd64(x);

   printf("expd64(%Df) = %Df\n", x, y);
}
```

Related information

- "math.h" on page 44
- "exp(), expf(), expl() — Calculate exponential function" on page 448
- "logd32(), logd64(), logd128() — Calculate natural logarithm" on page 994
- "log10d32(), log10d64(), log10d128() — Calculate base 10 logarithm" on page 1002
- "powd32(), powd64(), powd128() — Raise to power" on page 1194

expm1(), expm1f(), expm1l() — Exponential minus one

Standards

Standards / Extensions	C or C++	Dependencies
XPG4.2 C99 Single UNIX Specification, Version 3 C++ TR1 C99	both	z/OS V1R7

Format

```
#define _XOPEN_SOURCE_EXTENDED 1
#include <math.h>

double expm1(double x);
```

C99:

```
#define _ISOC99_SOURCE
#include <math.h>

float expm1f(float x);
long double expm1l(long double x);
```

C++ TR1 C99:

```
#define _TR1_C99
#include <math.h>

float expm1(float x);
long double expm1(long double x);
```

General description

The expm1() functions calculate the function:

e^x -1.0

Note: The following table shows the viable formats for these functions. See "IEEE binary floating-point" on page 94 for more information about IEEE Binary Floating-Point.

Function	Hex	IEEE
expm1	X	X
expm1f	X	X
expm1l	X	X

Returned value

If successful, expm1() returns the above function calculated on *x*.

If unsuccessful, expm1() may fail as follows:

- If *x* is negative and exceeds an internally defined large value, expm1() functions will return -1.0.
- If the value of the function overflows, expm1() functions will return HUGE_VAL or HUGE_VALF or HUGE_VALL as appropriate, and set errno to ERANGE.

Related information

- "math.h" on page 44
- "exp(), expf(), expl() — Calculate exponential function" on page 448
- "ilogb(), ilogbf(), ilogbl() — Integer unbiased exponent" on page 835
- "log1p(), log1pf(), log1pl() — Natural log of x+1" on page 998

expm1d32(), expm1d64(), expm1d128() — Exponential minus one

Standards

Standards / Extensions	C or C++	Dependencies
C/C++ DFP	both	z/OS V1.11

Format

```
#define __STDC_WANT_DEC_FP__
#include <math.h>

_Decimal32  expm1d32(_Decimal32 x);
_Decimal64  expm1d64(_Decimal64 x);
_Decimal128 expm1d128(_Decimal128 x);

_Decimal32  expm1(_Decimal32 x);      /* C++ only */
_Decimal64  expm1(_Decimal64 x);      /* C++ only */
_Decimal128 expm1(_Decimal128 x);     /* C++ only */
```

General description

The expm1() functions calculate the function:

e^x -1.0

Notes:

1. These functions work in IEEE decimal floating-point format. See “IEEE decimal floating-point” on page 95IEEE Decimal Floating-Point for more information.
2. To use IEEE decimal floating-point, the hardware must have the Decimal Floating-Point Facility installed.

Returned value

If successful, expm1() returns the above function calculated on *x*.

If unsuccessful, expm1() may fail as follows:

- If x is negative and exceeds an internally defined large value, expm1() functions will return -1.0.
- If the value of the function overflows, expm1() functions will return HUGE_VAL_D32 or HUGE_VAL_D64 or HUGE_VAL_D128 as appropriate, and set errno to ERANGE.

Example

```
/* CELEBE13

   This example illustrates the expm1d32() function.

*/

#define  __STDC_WANT_DEC_FP__
#include <math.h>
#include <stdio.h>

void main(void)
{
   _Decimal32 x, y;

   x = 2.5DF;
   y = expm1d32(x);

   printf("expm1d32(%Hf) = %Hf\n", x, y);
}
```

Related information

- “math.h” on page 44
- “expd32(), expd64(), expd128() — Calculate exponential function” on page 449
- “ilogbd32(), ilogbd64(), ilogbd128() — Integer unbiased exponent” on page 837
- “log1pd32(), log1pd64(), log1pd128() — Natural log of x+1” on page 999

ExportWorkUnit() — WLM export service

Standards

Standards / Extensions	C or C++	Dependencies
z/OS UNIX	both	OS/390 V2R9

Format

```
#include <sys/__wlm.h>

int ExportWorkUnit(wlmetok_t *enclavetoken,
                   wlmxtok_t *exporttoken,
                   unsigned long *conntoken);
```

AMODE 64:

```
#include <sys/__wlm.h>

int ExportWorkUnit(wlmetok_t *enclavetoken,
                   wlmxtok_t *exporttoken,
                   unsigned int *conntoken);
```

General description

Exports an enclave to all systems in a parallel sysplex, enabling dispatchable units on other systems to join the enclave.

The ExportWorkUnit() function uses the following parameters:

**enclavetoken*
: Points to a work unit enclave token that was returned from a call to CreateWorkUnit() or ContinueWorkUnit().

**exporttoken*
: Points to a data field of type wlmxtok_t where the ExportWorkUnit() function is to return the WLM work unit export token.

**conntoken*
: Specifies the connect token that represents the connection to WLM.

Returned value

If successful, ExportWorkUnit() returns 0.

If unsuccessful, ExportWorkUnit() returns -1 and sets errno to one of the following values:

Error Code
: **Description**

EFAULT
: An argument of this function contained an address that was not accessible to the caller.

EINVAL
: An argument of this function contained a value that is not correct.

EMVSSAF2ERR
: An error occurred in the security product.

EMVSWLMERROR
: A WLM service failed. Use __errno2() to obtain the WLM service reason code for the failure.

EPERM
: The calling thread's address space is not permitted to the BPX.WLMSERVER Facility class. The caller's address space must be permitted to the BPX.WLMSERVER Facility class, if the BPX.WLMSERVER class is defined. If BPX.WLMSERVER is not defined, the calling process is not defined as a superuser (UID=0).

Related information

- "sys/__wlm.h" on page 77
- "UnDoExportWorkUnit() — WLM undo export service" on page 1955
- For more information, see *z/OS MVS Programming: Workload Management Services, SC34-2663*.

exp2(), exp2f(), exp2l() — Calculate the base-2 exponential

Standards

Standards / Extensions	C or C++	Dependencies
C99 Single UNIX Specification, Version 3 C++ TR1 C99	both	z/OS V1R7

Format

```
#define _ISOC99_SOURCE
#include <math.h>

double exp2(double x);
float exp2f(float x);
long double exp2l(long double x);
```

C++ TR1 C99:

```
#define _TR1_C99
#include <math.h>

float exp2(float x);
long double exp2(long double x);
```

General description

The exp2 functions compute the base-2 exponential of *x*.

Note: The following table shows the viable formats for these functions. See “IEEE binary floating-point” on page 94 for more information about IEEE Binary Floating-Point.

Function	Hex	IEEE
exp2	X	X
exp2f	X	X
exp2l	X	X

Returned value

The exp2 functions return 2 to the power *x*.

Related information

- “math.h” on page 44

exp2d32(), exp2d64(), exp2d128() — Calculate the base-2 exponential

Standards

Standards / Extensions	C or C++	Dependencies
C/C++ DFP	both	z/OS V1.11

Format

```
#define __STDC_WANT_DEC_FP__
#include <math.h>

_Decimal32  exp2d32(_Decimal32 x);
_Decimal64  exp2d64(_Decimal64 x);
_Decimal128 exp2d128(_Decimal128 x);

_Decimal32  exp2(_Decimal32 x);      /* C++ only */
_Decimal64  exp2(_Decimal64 x);      /* C++ only */
_Decimal128 exp2(_Decimal128 x);     /* C++ only */
```

General description

The exp2() functions compute the base-2 exponential of *x*.

Notes:

1. These functions work in IEEE decimal floating-point format. See "IEEE decimal floating-point" on page 95IEEE Decimal Floating-Point for more information.
2. To use IEEE decimal floating-point, the hardware must have the Decimal Floating-Point Facility installed.

Returned value

The exp2() functions return 2 to the power *x*.

Example

```
/* CELEBE14

   This example illustrates the exp2d128() function.

*/

#define  __STDC_WANT_DEC_FP__
#include <math.h>
#include <stdio.h>

void main(void)
{
   _Decimal128 x, y;

   x = 4.785DL;
   y = exp2d128(x);

   printf("exp2d128(%DDf) = %DDf\n", x, y);
}
```

Related information

- "math.h" on page 44

extlink_np() — Create an external symbolic link

Standards

Standards / Extensions	C or C++	Dependencies
z/OS UNIX	both	

Format

```
#include <unistd.h>

int extlink_np(const char *ename, const char *elink);
```

General description

Creates the external symbolic link file named by *elink* with the object specified by *ename*. The *ename* is not resolved, and refers to an object outside the HFS (hierarchical file system). The variable *elink* is the name of the external symbolic link file created, and *ename* is the name of the object contained within that file.

Returned value

If successful, extlink_np() returns 0.

If unsuccessful, extlink_np() returns -1, does not affect any file it names, and sets errno to one of the following values:

Error Code
Description

EACCES
A component of the *elink* path prefix denies search permission.

EEXIST
The file named by *elink* already exists.

EINVAL
elink has a slash as its last component, which indicates that the preceding component will be a directory. An external link cannot be a directory.

ELOOP
A loop exists in symbolic links. This error is issued if the number of symbolic links encountered during resolution of the *elink* argument is greater than POSIX_SYMLOOP.

ENAMETOOLONG
pathname is longer than **PATH_MAX** characters, or some component of *pathname* is longer than **NAME_MAX** characters while _POSIX_NO_TRUNC is in effect. For symbolic links, the length of the *pathname* string substituted for a symbolic link exceeds **PATH_MAX**. The **PATH_MAX** and **NAME_MAX** values can be determined with pathconf().

ENOTDIR
A component of the path prefix of *elink* is not a directory.

ENOSPC
The new external link cannot be created because there is no space left on the file system to contain it.

EROFS
The file named by *elink* cannot be created on a read-only file system.

Example

CELEBE07

```
/* CELEBE07

   This example creates an external symbolic link.
```

```
 */
#include <stdio.h>
#include <fcntl.h>
#include <errno.h>
#include <unistd.h>
#include <stdlib.h>
#include <sys/stat.h>
#include <sys/types.h>

main( argc, argv )
  int   argc ;
  char  *argv ;
{
  int   i_rc ;
  int   i_fd ;
  char  ac_mvsds[] = "SYS1.LINKLIB" ;
  char  ac_mvsdsextsymlnk[] = "sys1.linklib.extsymlink" ;

  i_rc = unlink( ac_mvsdsextsymlnk ) ;

  if (( i_rc == -1 ) && ( errno == ENOENT )) {
  }
  else
  {
    perror( "unlink() error" ) ;
    return( -1 ) ;
  }

  printf( "Before extlink_np() call ...\n" ) ;
  system( "ls -il sys1.*" ) ;

  i_rc = extlink_np( ac_mvsds, ac_mvsdsextsymlnk ) ;

  if ( i_rc == -1 )
  {
    perror( "extlink_np() error" ) ;
    return( -1 ) ;
  }

  printf( "After extlink_np() call ...\n" ) ;
  system( "ls -il sys1.*" ) ;

  i_rc = unlink( ac_mvsdsextsymlnk ) ;
}
```

Related information

- "unistd.h" on page 82
- "link() — Create a link to a file" on page 970
- "lstat() — Get status of file or symbolic link" on page 1022
- "readlink() — Read the value of a symbolic link" on page 1387
- "symlink() — Create a symbolic link to a path name" on page 1795
- "unlink() — Remove a directory entry" on page 1962

ExtractWorkUnit() — Extract enclave service

Standards

Standards / Extensions	C or C++	Dependencies
z/OS UNIX	both	OS/390 V2R8

Format

```
#include <sys/__wlm.h>

int ExtractWorkUnit(wlmetok_t *enclavetoken);
```

General description

The ExtractWorkUnit() function will allow the task to retrieve the enclaves token for the purpose of performance management.

The ExtractWorkUnit() function uses the following parameter:

**enclavetoken*
: Points to a data field of type wlmetok_t where the ExtractWorkUnit() function is to return the WLM work unit token to which the current process is joined.

Returned value

If successful, ExtractWorkUnit() returns 0.

If unsuccessful, ExtractWorkUnit() returns -1 and sets errno to one of the following values:

Error Code
: **Description**

EFAULT
: An argument of this service contained an address that was not accessible to the caller.

EINVAL
: The Functioncode parm contains a value that is not correct or the function parmlist data is incorrect.

EMVSSAF2ERR
: An error occurred in the security product. Consult the reason code, which can be retrieved using the __errno2() function.

EMVSWLMERROR
: A WLM service failed. Consult the reason code, which can be retrieved using the __errno2() function.

EPERM
: Do not have appropriate permissions and privilege.

ESRCH
: A WLM_EXTRACT_WORKUNIT request was issued but the WLM enclave token was not returned.

Related information

- "sys/__wlm.h" on page 77

__e2a_l() — Convert characters from EBCDIC to ASCII

Standards

Standards / Extensions	C or C++	Dependencies
	both	z/OS V1R2

Format

```
#include <unistd.h>

size_t  __e2a_l(char  *bufptr, size_t  szLen)
```

General description

The __e2a_l() function converts *szLen* characters in *bufptr* between IBM-1047 and ISO8859-1, returning the number of characters converted if successful or -1 if not. Conversion occurs in place in the buffer. __e2a_l() is not sensitive to the locale, and only converts between ISO8859-1 and IBM-1047.

Note: This function is valid for applications compiled XPLINK only.

Returned value

If successful, __e2a_l() returns the number of characters converted.

If unsuccessful, __e2a_l() returns -1 and sets errno to the following value:

Error Code
Description

EINVAL
The pointer to *bufptr* is NULL or *szLen* is a negative value.

Related information

- “unistd.h” on page 82
- “__a2e_l() — Convert characters from ASCII to EBCDIC” on page 206
- “__a2e_s() — Convert string from ASCII to EBCDIC” on page 206
- “__e2a_s() — Convert string from EBCDIC to ASCII”

__e2a_s() — Convert string from EBCDIC to ASCII

Standards

Standards / Extensions	C or C++	Dependencies
	both	z/OS V1R2

Format

```
#include <unistd.h>

size_t  __e2a_s(char  *string)
```

General description

The __e2a_s() function converts a string between IBM-1047 and ISO8859-1, returning the string length if successful or -1 if not. Conversion occurs in place in the string. __e2a_s() is not sensitive to the locale, and only converts between ISO8859-1 and IBM-1047.

Note: This function is valid for applications compiled XPLINK only.

Returned value

If successful, __e2a_s() returns the string length.

If unsuccessful, __e2a_s() returns -1 and sets errno to the following value:

Error Code
Description

EINVAL
The pointer to string is NULL.

Related information

- "unistd.h" on page 82
- "__a2e_l() — Convert characters from ASCII to EBCDIC" on page 206
- "__a2e_s() — Convert string from ASCII to EBCDIC" on page 206
- "__e2a_l() — Convert characters from EBCDIC to ASCII" on page 458

fabs(), fabsf(), fabsl() — Calculate floating-point absolute value

Standards

Standards / Extensions	C or C++	Dependencies
ISO C POSIX.1 XPG4 XPG4.2 ISO/ANSI C++ C99 Single UNIX Specification, Version 3 C++ TR1 C99	both	

Format

```
#include <math.h>

double fabs(double x);
float fabs(float x);                  /* C++ only */
long double fabs(long double x);      /* C++ only */
float fabsf(float x);
long double fabsl(long double x);
```

General description

The fabs() functions calculate the absolute value of a floating-point argument.

Note: These functions work in both IEEE Binary Floating-Point and hexadecimal floating-point formats. See "IEEE binary floating-point" on page 94 for more information about IEEE Binary Floating-Point.

Returned value

Returns the absolute value of the float input.

Example

```
/* This example calculates y as the absolute value of x. */
#include <math.h>

int main(void)
```

```
{
   double x, y;

   x = -5.6798;
   y = fabs(x);

   printf("fabs( %f ) = %f\n", x, y);
}
```

Output

```
fabs( -5.679800 ) = 5.679800
```

Related information

- “math.h” on page 44
- “abs(), absf(), absl() — Calculate integer absolute value” on page 103
- “labs() — Calculate long absolute value” on page 935

fabsd32(), fabsd64(), fabsd128() — Calculate floating-point absolute value

Standards

Standards / Extensions	C or C++	Dependencies
C/C++ DFP	both	z/OS V1.8

Format

```
#define __STDC_WANT_DEC_FP__
#include <math.h>

_Decimal32  fabsd32(_Decimal32 x);
_Decimal64  fabsd64(_Decimal64 x);
_Decimal128 fabsd128(_Decimal128 x);
_Decimal32  fabs(_Decimal32 x);    /* C++ only */
_Decimal64  fabs(_Decimal64 x);    /* C++ only */
_Decimal128 fabs(_Decimal128 x);   /* C++ only */
```

General description

The fabs() functions calculate the absolute value of a decimal floating-point argument.

Notes:

1. To use IEEE decimal floating-point, the hardware must have the Decimal Floating-Point Facility installed.
2. These functions work in IEEE decimal floating-point format. See “IEEE binary floating-point” on page 94 for more information

Returned value

Returns the absolute value of the decimal floating-point input.

Example

```
/* CELEBF75

   This example illustrates the fabsd128() function.

*/
```

```
#define  __STDC_WANT_DEC_FP__
#include <math.h>
#include <stdio.h>

int main(void)
{
  _Decimal128 x, y;

  x = -5.6798DL;
  y = fabsd128(x);

  printf("fabsd128(%DDf) = %DDf\n", x, y);
}
```

Related information

- “math.h” on page 44
- “fabs(), fabsf(), fabsl() — Calculate floating-point absolute value” on page 460

fattach() — Attach a STREAMS-based file descriptor to a file in the file system name space

Standards

Standards / Extensions	C or C++	Dependencies
XPG4.2 Single UNIX Specification, Version 3	both	

Format

```
#define _XOPEN_SOURCE_EXTENDED 1
#include <stropts.h>

int fattach(int fildes, const char *path);
```

General description

The fattach() function attaches a STREAMS-based file descriptor to a file, effectively associating a path name with *fildes*. The *fildes* argument must be a valid open file descriptor associated with a STREAMS file. The *path* argument points to a path name of an existing file. The process must have appropriate privileges, or must be the owner of the file named by *path* and have write permission. A successful call to fattach() causes all path names that name the file named by *path* to name the STREAMS file associated with *fildes*, until the STREAMS file is detached from the file. A STREAMS file can be attached to more than one file and can have several path names associated with it.

The attributes of the named STREAMS file are initialized as follows: the permissions, user ID, group ID, and times are set to those of the file named by *path*, the number of links is set to 1, and the size and device identifier are set to those of the STREAMS file associated with *fildes*. If any attributes of the named STREAMS file are subsequently changed (for example, by chmod()), neither the attributes of the underlying file nor the attributes of the STREAMS file to which *fildes* refers are affected.

File descriptors referring to the underlying file, opened before an fattach() call, continue to refer to the underlying file.

Returned value

If successful, fattach() returns 0.

If unsuccessful, fattach() returns -1 and sets errno to one of the following values.

Note: z/OS UNIX services do not supply any STREAMS devices or pseudodevices. It is impossible for fattach() to attach a STREAMS-based file descriptor to a file. It will always return -1 with errno set to indicate the failure. See “open() — Open a file” on page 1152 for more information.

Error Code
Description

EACCES
Search permission is denied for a component of the path prefix, or the process is the owner of *path* but does not have write permissions on the file named by *path*.

EBADF
The *fildes* argument is not a valid open file descriptor.

EBUSY
The file named by *path* is currently a mount point or has a STREAMS file attached to it.

EINVAL
The *fildes* argument does not refer to a STREAMS file.

ELOOP
Too many symbolic links were encountered in resolving *path*.

ENAMETOOLONG
The size of *path* exceeds **PATH_MAX**, or a component of *path* is longer than **NAME_MAX**, or path name resolution of a symbolic link produced an intermediate result whose length exceeds **PATH_MAX**.

ENOENT
A component of *path* does not name an existing file or *path* is an empty string.

ENOTDIR
A component of the path prefix is not a directory.

EPERM
The effective user ID of the process is not the owner of the file named by *path* and the process does not have appropriate privilege.

Related information

- “stropts.h” on page 72
- “fdetach() — Detach a name from a STREAMS-based file descriptor” on page 488
- “isastream() — Test a file descriptor” on page 902

__fbufsize() — Retrieve the buffer size of an open stream

Standards

Standards / Extensions	C or C++	Dependencies
Language Environment	both	

Format

```
#include <stdio.h>
#include <stdio_ext.h>

size_t __fbufsize(FILE *stream);
```

General description

The __fbufsize() function retrieves the buffer size, in bytes, of the specified stream.

Returned value

The __fbufsize() function returns the size of the buffer in bytes. Otherwise, the __fbufsize() function returns 0. If an error has occurred, __fbufsize() returns 0 and sets errno to nonzero.

An application wishing to check for error situations should set errno to 0, then call __fbufsize(), and then check errno. If errno is nonzero, assume that an error has occurred.

Error Code
    Description

EBADF
    The stream specified by *stream* is not valid.

Example

CELEBF87

```
/* CELEBF87

   This example determines the size of the I/O buffer
   for an open stream.

*/

#include <stdio.h>
#include <stdio_ext.h>

void main() {
   FILE *f;
   int bufsize = 0;
   char filename[FILENAME_MAX] = "myfile.dat";

   f = fopen(filename,"wb");
   if (f == NULL) {
      perror("fopen failed\n");
      return;
   }

   bufsize = __fbufsize(f);

   printf("The buffer size for %s is %d\n",filename,bufsize);

   return;
}
```

Output

```
The buffer size for myfile.dat is 4096
```

Related information

- “stdio.h” on page 68
- “stdio_ext.h” on page 70
- “setbuf() — Control buffering” on page 1522
- “setvbuf() — Control buffering” on page 1596

__fchattr() — Change the attributes of a file or directory by file descriptor

Standards

Standards / Extensions	C or C++	Dependencies
z/OS UNIX	both	z/OS V1R2

Format

```
#define _OPEN_SYS_FILE_EXT 1
#include <sys/stat.h>

int __fchattr(int filedes, attrib_t *attributes, int attributes_len);
```

General description

The __fchattr() function modifies the attributes that are associated with a file. It can be used to change the mode, owner, access time, modification time, change time, reference time, audit flags, general attribute flags, file tag, and file format and size. The file to be impacted is defined by its file descriptor with the *filedes* argument.

The *attributes* argument is the address of an attrib_t structure which is used to identify the attributes to be modified and the new values desired. The attrib_t type is an f_attributes structure as defined in <sys/stat.h> for use with the __fchattr() function. For proper behavior, the user should ensure that this structure has been initialized to zeros before it is populated. The f_attributes structure is defined as indicated in Table 20 on page 261.

The f_attributes structure is defined in <sys/stat.h> for use with the __fchattr() function.

Returned value

If successful, __fchattr() returns 0.

If unsuccessful, __fchattr() returns -1 and sets errno to one of the following values:

Error Code
Description

EACCES
The calling process did not have appropriate permissions. Possible reasons include:

- The calling process was attempting to set access time or modification time to current time, and the effective UID of the calling process does not match the owner of the file; the process does not have write permission for the file; or the process does not have appropriate privileges.

- The calling process was attempting to truncate the file, and it does not have write permission for the file.

EBADF
The *filedes* parameter is not a valid file descriptor.

ECICS
An attempt was made to change file tag attributes under non-OTE CICS and file tagging is not supported in that environment.

EFBIG
The calling process was attempting to change the size of a file but the specified length is greater than the maximum file size limit for the process.

EINVAL
The attributes structure containing the requested changes is not valid.

EPERM
The operation is not permitted for one of the following reasons:

- The calling process was attempting to change the mode or the file format but the effective UID of the calling process does not match the owner of the file, and the calling process does not have appropriate privileges.
- The calling process was attempting to change the owner but it does not have appropriate privileges.
- The calling process was attempting to change the general attribute bits but it does not have write permission for the file.
- The calling process was attempting to set a time value (not current time) but the effective UID does not match the owner of the file, and it does not have appropriate privileges.
- The calling process was attempting to set the change time or reference time to current time but it does not have write permission for the file.
- The calling process was attempting to change auditing flags but the effective UID of the calling process does not match the owner of the file, and the calling process does not have appropriate privileges.
- The calling process was attempting to change the Security Auditor's auditing flags but the user does not have auditor authority.

EROFS
pathname specifies a file that is on a read-only file system.

Related information

- “sys/stat.h” on page 75
- “__chattr() — Change the attributes of a file or directory” on page 261
- “__lchattr() — Change the attributes of a file or directory when they point to a symbolic or external link” on page 936

fchaudit() — Change audit flags for a file by descriptor

Standards

Standards / Extensions	C or C++	Dependencies
z/OS UNIX	both	

Format

```
#include <sys/stat.h>

int fchaudit(int fildes, unsigned int flags, unsigned int option);
```

General description

Changes the audit flags of a file. The parameter *fildes* is the file descriptor for the open file whose audit flags are to be changed. *flags* specifies what the audit flags should be changed to:

AUDTREADFAIL
Audit failing read requests.

AUDTREADSUCC
Audit successful read requests.

AUDTWRITEFAIL
Audit failing write requests.

AUDTWRITESUCC
Audit successful write requests.

AUDTEXECFAIL
Audit failing execute or search requests.

AUDTEXECSUCC
Audit successful execute or search requests. The bitwise inclusive-OR of any or all of these can be used to set more than one type of auditing.

The parameter *option* specifies whether the user audit flags or the security auditor audit flags should be changed:

AUDT_USER (0)
User audit flags are changed. The user must be the file owner or have appropriate authority to change the user audit flags for a file.

AUDT_AUDITOR (1)
Security-auditor audit flags are changed. The user must have security auditor authority to change the security auditor audit flags for a file.

Returned value

If successful, fchaudit() returns 0.

If unsuccessful, fchaudit() returns -1 and sets errno to one of the following values:

Error Code
Description

EBADF
fildes is not a valid open file descriptor.

EINVAL
option does not contain a 0 or 1.

EPERM
The effective user ID (UID) of the calling process does not match the owner of the file, and the calling process does not have appropriate privileges.

EROFS
fildes is associated with a file that is on a read-only file system.

Example

CELEBF02

```
/* CELEBF02

   The following program changes the audit flags of a file.

 */
#define _OPEN_SYS
#include <fcntl.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <unistd.h>
#include <stdio.h>

main() {
  int fd;
  char fn[]="fchaudit.file";

  if ((fd = creat(fn, S_IRUSR|S_IWUSR)) < 0)
    perror("creat() error");
  else {
    if (fchaudit(fd, AUDTREADSUCC, AUDT_USER) != 0)
      perror("fchaudit() error");
    close(fd);
    unlink(fn);
  }
}
```

Related information

- "sys/stat.h" on page 75
- "access() — Determine whether a file can be accessed" on page 111
- "chaudit() — Change audit flags for a file by path" on page 265
- "fchmod() — Change the mode of a file or directory by descriptor" on page 469
- "fchown() — Change the owner or group by file descriptor" on page 471
- "fstat() — Get status information about a file" on page 651

fchdir() — Change working directory

Standards

Standards / Extensions	C or C++	Dependencies
XPG4.2 Single UNIX Specification, Version 3	both	

Format

```
#define _XOPEN_SOURCE_EXTENDED 1
#include <unistd.h>

int fchdir(int fildes);
```

General description

The fchdir() function has the same effect as *chdir*() except that the directory that is to be new current working directory is specified by the file descriptor *fildes*.

Returned value

If successful, fchdir() changes the working directory and returns 0.

If unsuccessful, fchdir() does not change the working directory, returns -1, and sets errno to one of the following values:

Error Code
Description

EACCES
Search permission is denied for the directory referenced by *fildes*.

EBADF
The *fildes* arguments is not an open file descriptor.

ENOTDIR
The open file descriptor *fildes* does not refer to a directory.

Related information

- "unistd.h" on page 82
- "chdir() — Change the working directory" on page 267
- "chroot() — Change root directory" on page 279

fchmod() — Change the mode of a file or directory by descriptor

Standards

Standards / Extensions	C or C++	Dependencies
POSIX.1a XPG4.2 Single UNIX Specification, Version 3	both	

Format

```
#define _POSIX1_SOURCE 2
#include <sys/stat.h>

int fchmod(int fildes, mode_t mode);
```

General description

Sets the S_ISUID, S_ISGID, and file permission bits of the open file identified by *fildes*, its file descriptor.

The *mode* argument is created with one of the symbols defined in the sys/stat.h header file. For more information on these symbols, refer to "chmod() — Change the mode of a file or directory" on page 272.

Returned value

If successful, fchmod() marks for update the `st_ctime` field of the file and returns 0.

If unsuccessful, fchmod() returns -1 and sets errno to one of the following values:

Error Code
Description

EBADF
fildes is not a valid open file descriptor.

EPERM
The effective user ID (UID) does not match the owner of the file, and the calling process does not have appropriate privileges.

EROFS
The file resides on a read-only file system.

Example

CELEBF03

```
/* CELEBF03

   This example changes a file permission.

 */
#define _POSIX_SOURCE
#include <fcntl.h>
#include <sys/stat.h>
#include <sys/types.h>
#include <unistd.h>
#undef _POSIX_SOURCE
#include <stdio.h>

main() {
  char fn[]="temp.file";
  int  fd;
  struct stat info;

  if ((fd = creat(fn, S_IWUSR)) < 0)
    perror("creat() error");
  else {
    stat(fn, &info);
    printf("original permissions were: %08x\n", info.st_mode);
    if (fchmod(fd, S_IRWXU|S_IRWXG) != 0)
      perror("fchmod() error");
    else {
      stat(fn, &info);
      printf("after fchmod(), permissions are: %08x\n", info.st_mode);
    }
    close(fd);
    unlink(fn);
  }
}
```

Output

```
original permissions were: 03000080
after fchmod(), permissions are: 030001f8
```

Related information

- “sys/stat.h” on page 75
- “chmod() — Change the mode of a file or directory” on page 272
- “chown() — Change the owner or group of a file or directory” on page 275
- “fchown() — Change the owner or group by file descriptor” on page 471
- “mkdir() — Make a directory” on page 1068
- “mkfifo() — Make a FIFO special file” on page 1071
- “open() — Open a file” on page 1152
- “stat() — Get file information” on page 1720

fchown() — Change the owner or group by file descriptor

Standards

Standards / Extensions	C or C++	Dependencies
POSIX.1a XPG4.2 Single UNIX Specification, Version 3	both	

Format

```
#define _POSIX1_SOURCE 2
#include <unistd.h>

int fchown(int fildes, uid_t owner, gid_t group);
```

General description

Changes the owner or group (or both) of a file. *fildes* is the file descriptor for the file. *owner* is the user ID (UID) of the new owner of the file. *group* is the group ID of the new group for the file.

If _POSIX_CHOWN_RESTRICTED is defined in the unistd.h header file, a process can change the group of a file only if one of the following conditions is true:

1. The process has appropriate privileges.

   Or
2. All of the following are true:
   1. The effective user ID of the process is equal to the user ID of the file owner.
   2. The *owner* argument is equal to the user ID of the file owner or `(uid_t)-1`,
   3. The *group* argument is either the effective group ID or a supplementary group ID of the calling process.

If *fildes* points to a regular file and one or more of the S_IXUSR, S_IXGRP, or S_IXOTH bits of the file mode are set when fchown() returns successfully, it clears the set-user-ID (S_ISUID) and set-group-ID (S_ISGID) bits of the file mode.

If the file referred to by *fildes* is not a regular file and one or more of the S_IXUSR, S_IXGRP, or S_IXOTH bits of the file mode are set, the set-user-ID (S_ISUID) and set-group-ID (S_ISGID) bits of the file are cleared.

When fchown() completes successfully, it marks the `st_ctime` field of the file to be updated.

Returned value

If successful, fchown() updates the change time for the file and returns 0.

If unsuccessful, fchown() returns -1 and sets errno to one of the following values:

Error Code
Description

EBADF
fildes is not a valid open file descriptor.

EPERM
Either the effective user ID does not match the owner of the file, or the calling process does not have appropriate privileges, and POSIX_CHOWN_RESTRICTED indicates that such privilege is required.

EROFS
The file resides on a read-only system.

Example

CELEBF04

```
/* CELEBF04

   This example changes the owner ID and group ID.

 */
#define _POSIX_SOURCE
#include <fcntl.h>
#include <unistd.h>
#include <sys/stat.h>
#include <sys/types.h>
#undef _POSIX_SOURCE
#include <stdio.h>

main() {
  char fn[]="temp.file";
  FILE *stream;
  int  fd;
  struct stat info;

  if ((fd = creat(fn, S_IWUSR)) < 0)
    perror("creat() error");
  else {
    stat(fn, &info);
    printf("original owner was %d and group was %d\n", info.st_uid,
           info.st_gid);
    if (fchown(fd, 25, 0) != 0)
      perror("fchown() error");
    else {
      stat(fn, &info);
      printf("after fchown(), owner is %d and group is %d\n",
             info.st_uid, info.st_gid);
    }
    close(fd);
    unlink(fn);
  }
}
```

Output

```
original owner was 0 and group was 500
after fchown(), owner is 25 and group is 0
```

Related information

- "unistd.h" on page 82
- "chown() — Change the owner or group of a file or directory" on page 275
- "chmod() — Change the mode of a file or directory" on page 272
- "fchmod() — Change the mode of a file or directory by descriptor" on page 469
- "mkdir() — Make a directory" on page 1068
- "mkfifo() — Make a FIFO special file" on page 1071
- "open() — Open a file" on page 1152
- "stat() — Get file information" on page 1720

fclose() — Close file

Standards

Standards / Extensions	C or C++	Dependencies
ISO C POSIX.1 XPG4 XPG4.2 C99 Single UNIX Specification, Version 3	both	

Format

```
#include <stdio.h>

int fclose(FILE *stream);
```

General description

Flushes a stream, and then closes the file associated with that stream. Afterwards, the function releases any buffers associated with the stream. To *flush* means that unwritten buffered data is written to the file, and unread buffered data is discarded.

A pointer to a closed file *cannot* be used as an input value to the freopen() function.

Notes:

1. The storage pointed to by the FILE pointer is freed by the fclose() function. An attempt to use the FILE pointer to a closed file is not valid. This restriction is true even when fclose() fails.
2. If an application has locked a (FILE *) object (with flockfile() or ftrylockfile()), it is responsible for relinquishing the locked (FILE *) object (with funlockfile()) before calling fclose(). Failure to relinquish a locked (FILE *) object may cause deadlock (or looping).

Returned value

If successful closing the stream, fclose() returns 0.

If a failure occurs in flushing buffers or in outputting data, fclose() returns EOF. An attempt will still be made to close the file.

Special behavior for XPG4

fclose() sets errno to one of the following values:

Error Code
Description

EAGAIN
The O_NONBLOCK flag is set and output cannot be written immediately.

EBADF
The underlying file descriptor is not valid.

EFBIG
Writing to the output file would exceed the maximum file size or the process's file size supported by the implementation.

EINTR
The fclose() function was interrupted by a signal before it had written any output.

EIO The process is in a background process group and is attempting to write to its controlling terminal, but TOSTOP (defined in the termio.h include file) is set, the process is neither ignoring nor blocking SIGTTOU signals, and the process group of the process is orphaned.

ENOSPC
There is no free space left on the output device

ENXIO
A request was made of a nonexistent device, or the request was outside the device.

EPIPE fclose() is trying to write to a pipe or FIFO that is not open for reading by any process. This error also generates a SIGPIPE signal.

Example

```
/* This example opens a file myfile.dat for reading as a stream and then
   closes the file.
 */
#include <stdio.h>

int main(void)
{
   FILE *stream;

   stream = fopen("myfile.dat", "r");
⋮
   if (fclose(stream))   /* Close the stream. */
      printf("fclose error\n");
}
```

Related information

- See the topics about closing files and opening files in *z/OS XL C/C++ Programming Guide*
- “stdio.h” on page 68
- “fopen() — Open a file” on page 565
- “freopen() — Redirect an open file” on page 620

fcntl() — Control open file descriptors

Standards

Standards / Extensions	C or C++	Dependencies
POSIX.1 XPG4 XPG4.2 Single UNIX Specification, Version 3	both	

Format

```
#define _POSIX_SOURCE
#include <fcntl.h>

int fcntl(int fildes, int action, ...);
```

X/Open:

```
#define _XOPEN_SOURCE_EXTENDED 1
#include <sys/types.h>
#include <unistd.h>
#include <fcntl.h>

int fcntl(int socket, int cmd, ...);
```

Berkeley sockets:

```
#define _OE_SOCKETS
#include <sys/types.h>
#include <unistd.h>
#include <fcntl.h>

int fcntl(int socket, int cmd, ...);
```

General description

The fcntl() function performs various actions on open file descriptors.

The argument *fildes* is a file descriptor for the file you want to manipulate. *action* is a symbol indicating the action you want to perform on *fildes*. These symbols are defined in the <fcntl.h> header file. If needed, "`...`" indicates a third argument. The type of the third argument depends on *action*, and some actions do not need an additional argument.

Behavior for sockets: The operating characteristics of sockets can be controlled with the fcntl() call. The operations to be controlled are determined by *cmd*. The *arg* parameter is a variable with a meaning that depends on the value of the *cmd* parameter.

Parameter	**Description**
socket	The socket descriptor.
cmd	The command to perform.
arg	The data associated with *cmd*.

The *action* argument can be one of the following symbols:

F_CLOSFD
Closes a range of file descriptors. A third `int` argument must be specified to indicate the upper limit for the range of the file descriptors to be closed, while *fildes* specifies the lower limit. If -1 is specified for the third argument, all file descriptors greater than or equal to the lower limit are closed.

F_DUPFD
Duplicates the file descriptor. A third `int` argument must be specified. fcntl() returns the lowest file descriptor greater than or equal to this third argument that is not already associated with an open file. This file descriptor refers to the same file as *fildes* and shares any locks. The flags

FD_CLOEXEC and FD_CLOFORK are turned off in the new file descriptor, so that the file is kept open if an exec function is called.

Note: If *fildes* is an XTI endpoint, there must be at least one available file descriptor greater than or equal to the third argument and less than 65536.

F_DUPFD2

Duplicates the file descriptor. A third `int` argument must be specified to indicate which file descriptor to use as the duplicate. This file descriptor is closed if already open and then used as the new file descriptor. The new file descriptor refers to the same file as *fildes* and shares any locks. The flag FD_CLOEXEC is turned off in the new file descriptor, so that the file is kept open if an exec function is called.

Note:

1. If the third argument refers to the same file descriptor as *fildes*, the file descriptor is not closed and is FD_CLOEXEC is not turned off.
2. If *fildes* is an XTI endpoint, the third argument must not exceed the limit of 65535.

F_GETFD

Obtains the file descriptor flags for *fildes*. fcntl() returns these flags as its result. For a list of supported file descriptor flags, see "File flags" on page 480.

F_SETFD

Sets the file descriptor flags for *fildes*. You must specify a third `int` argument, giving the new file descriptor flag settings. fcntl() returns 0 if it successfully sets the flags.

F_GETFL

Obtains the file status flags and file access mode flags for *fildes*. fcntl() returns these flags as its result. For a list of supported file status and file access mode flags, see "File flags" on page 480.

Usage of the F_GETFL action will return the setting of O_LARGEFILE status flag when fcntl() has been enabled to operate on large files

Behavior for sockets: This command gets the status flags of socket descriptor *socket*. With the `_OPEN_SYS` feature test macro you can query the FNDELAY flag. With the `_XOPEN_SOURCE_EXTENDED` 1 feature test macro you can query the O_NDELAY flag. The FNDELAY and O_NDELAY flags mark *socket* as being in nonblocking mode. If data is not present on calls that can block, such as read(), readv(), and recv(), the call returns with -1, and the error code is set to EWOULDBLOCK.

F_SETFL

Sets the file status flags for *fildes*. You must specify a third `int` argument, giving the new file descriptor flag settings. fcntl() does not change the file access mode, and file access bits in the third argument are ignored. fcntl() returns 0 if it successfully sets the flags.

Behavior for sockets: This command sets the status flags of socket descriptor *socket*. With the `_OPEN_SYS` feature test macro you can set the FNDELAY flag. With the `_XOPEN_SOURCE_EXTENDED` 1 feature test macro you can set the O_NDELAY flag.

F_GETLK

Obtains locking information for a file. See "File locking" on page 481

F_SETLK

Sets or clears a file segment lock. See "File locking" on page 481

F_SETLKW

Sets or clears a file segment lock; but if a shared or exclusive lock is blocked by other locks, fcntl() waits until the request can be satisfied. See "File locking" on page 481

F_GETOWN

Behavior for sockets: Obtains the PID for the filedes and returns this value. The value returned will be either the process ID or the process group ID that is associated with the socket. If it is a positive integer, it specifies a process ID. If it is a negative integer (other than -1), it specifies a process group ID.

F_SETOWN

Behavior for sockets: Sets either the process ID or the process group ID that is to receive either the SIGIO or SIGURG signals for the socket associated with filedes. The SIGURG signal is generated as a result of receiving out-of-band data. Refer to send(), sendto(), sendmsg(), and recv(), recvfrom() and recvmsg() for more information on sending and receiving out-of-band data.

You must specify a third int argument, giving the PID requested. This value can be either a positive integer, specifying a process ID, or a negative integer (other than -1), specifying a process group ID. The difference between specifying a process ID or a process group ID is that in the first case only a single process will receive the signal, while in the second case all processes in the process group will receive the signal.

F_SETTAG

Sets the file tag for the file referred to by file descriptor *fildes*.

The third argument *ftag* is the address of a populated file_tag structure.

If the *ftag* argument supplied to fcntl(F_SETTAG) does not have the ft_deferred bit set ON, fcntl() will immediately set the file's File Tag with the provided *ftag*'s ft_ccsid and ft_txtflag values.

If the *ftag* argument supplied to fcntl(F_SETTAG) has the ft_deferred bit set ON, fcntl() will not set the file's File Tag until first write to the file. The CCSID used to tag the file will be the current Program CCSID at the time of first write, regardless of the *ftag* ft_ccsid value, however the ft_txtflag value will be used.

If the ft_ccsid of the specified file_tag differs from the Program CCSID, automatic file conversion will occur, provided:

- The ft_txtflag is set to ON.
- Environment variable _BPXK_AUTOCVT is ON or ALL; or if _BPXK_AUTOCVT is unset, the BPXPRMxx member AUTOCVT is either ON or ALL.

Restriction: When _BPXK_AUTOCVT is ON, automatic conversion can only take place between IBM-1047 and ISO8859-1 code sets. Other CCSID pairs are not supported for automatic text conversion. To request automatic conversion for any CCSID pairs that Unicode service supports, set _BPXK_AUTOCVT to ALL.

If AUTOCVT(OFF) and _BPXK_AUTOCVT=OFF, the file will be tagged with the specified file_tag's ft_ccsid and ft_txtflag values, but automatic conversion will not occur. See the "Using Environment Variables" chapter in *z/OS XL C/C++ Programming Guide* for more information about the _BPXK_AUTOCVT environment variable.

If the *ftag* argument supplied to fcntl(F_SETTAG) has the ft_deferred bit set ON, pipes and FIFOs are tagged from the write end with the Program CCSID of the first writer.

F_CONTROL_CVT

Controls or queries the conversion status of the open file referred to by file descriptor *fildes*. Conversion control is generally used to provide CCSID information for untagged files or untagged programs.

Character set conversion between a program and a file, pipe or other I/O stream can be enabled or changed with F_CONTROL_CVT. A pair of CCSID's is specified or defaulted, one for the program and one for the data. As the program reads and writes data, the system will convert from one CCSID to the other.

The third f_cnvrt argument is the required address of an f_cnvrt structure. This structure is defined in <fcntl.h> and includes the following members:

Table 22. Struct f_cnvrt Element Descriptions

Element	Data Type	Description
pccsid	short	The Program CCSID - This is output from query and input to setting conversion ON. A value of 0 on input indicates to use the previously set value or the current Program CCSID.
fccsid	short	The File CCSID - This is output from query and input to setting conversion ON. A value of 0 on input indicates to use the CCSID from the File Tag as stored in the file, specified on mount, or set by a prior call.

Table 22. Struct f_cnvrt Element Descriptions (continued)

Element	Data Type	Description
cvtcmd	int	Conversion Control Command. The following conversion controls are available: • Query Conversion - Returns whether or not conversion is in effect and the Program and File CCSIDs being used. On input, *cvtcmd* is set to QUERYCVT, and on output, it is changed to either SETCVTON, SETCVTOFF, or SETCVTALL to indicate that conversion is currently ON, OFF, or ALL respectively. The current CCSIDs are also returned in their respective positions in the f_cnvrt structure. • Set Conversion OFF - Turns OFF any conversion that may be in effect. On input, *cvtcmd* is set to SETCVTOFF and the rest of the f_cnvrt is ignored. There is no output. A program can use this to override an automatic conversion that might be established by the environment within which it is invoked. If conversion is currently in effect, the CCSIDs being used will be remembered while conversion is turned OFF, so that the prior conversion may be resumed without the program having to remember what the prior CCSIDs were. • Set Conversion ON - Turns on enhanced ASCII conversion environment and optionally specifies the CCSIDs to use in place of the Program or File CCSIDs that are currently in effect. On input, *cvtcmd* is set either to SETCVTON to unconditionally turn on Enhanced ASCII, or to SETAUTOCVTON to turn on it only if _BPXK_AUTOCVT=ON or AUTOCVT(ON) was specified in BPXPRMxx. In this case, automatic conversion can only take place between IBM-1047 and ISO8859-1 code sets. Other CCSID pairs are not supported for automatic text conversion. A value of 0 for the Program CCSID indicates that the current Program CCSID be used. A value of 0 for the file CCSID indicates that no change should be made to the File CCSID. This does not affect the stored File Tag or the current Program CCSID. It only changes the values being used to control conversion on this data stream. • Set Conversion ALL - Turns on Unicode conversion environment and optionally specifies the CCSIDs to use in place of the Program or File CCSIDs that are currently in effect. On input, *cvtcmd* can be set either to SETCVTALL to unconditionally turn on Unicode conversion, or to SETAUTOCVTALL to turn on it only if _BPXK_AUTOCVT=ALL or AUTOCVT(ALL) was specified in BPXPRMxx. SETCVTALL or SETAUTOCVTALL has no effect after the first read or write of the file. In this case, automatic conversion can take place between any code sets that Unicode service supports. A value of 0 for the Program CCSID indicates that the current Program CCSID be used. A value of 0 for the file CCSID indicates that no change should be made to the File CCSID. This does not affect the stored File Tag or the current Program CCSID. It only changes the values being used to control conversion on this data stream.

The call fails if a conversion table is not installed for the resulting CCSID pair.

Attention: Flipping the autoconversion mode off and on, or changing the CCSID values when file conversion and/or tagging takes place, or setting the CCSIDs to values that are not compatible with the program or file, can be quite unpredictable.

Large file support for z/OS UNIX files: Large z/OS UNIX files are supported automatically for AMODE 64 C/C++ applications. AMODE 31 C/C++ applications must be compiled with the option LANGLVL(LONGLONG) and define the _LARGE_FILES feature test macro before any headers are included to enable this function to operate on z/OS UNIX files that are larger than 2 GB in size. File size and offset fields are enlarged to 63 bits in width. Therefore, any other function operating on the file is required to define the _LARGE_FILES feature test macro as well.

File flags

There are several types of flags associated with each open file. Flags for a file are represented by symbols defined in the <fcntl.h> header file.

The following *file descriptor* flags can be associated with a file:

FD_CLOEXEC
If this flag is 1, the file descriptor is closed if the process executes one of the exec function calls. If it is 0, the file remains open.

FD_CLOFORK
If this flag is 1 when a fork occurs, the file descriptor will be closed for the child process. If it is 0, the file remains open for the child.

The following *file status* flags can be associated with a file:

O_APPEND
Append mode. If this flag is 1, every write operation on the file begins at the end of the file.

O_ASYNCSIG
If this flag is 1, then asynchronous I/O will be used for the file.

O_NONBLOCK
No blocking. If this flag is 1, read and write operations on the file return with an error status if they cannot perform their I/O immediately. If this flag is 0, read and write operations on the file wait (or "block") until the file is ready for I/O. For more details, see "read() — Read from a file or socket" on page 1376 and "write() — Write data on a file or socket" on page 2085.

O_SYNC
Force synchronous update. If the flag is 1, every write() operation on the file is written to permanent storage. That is, the file system buffers are forced to permanent storage. (See "fsync() — Write changes to direct-access storage" on page 655.) If this flag is 0, update operations on the file will not be completed until the data has been written to permanent storage. On return from a function that performs a synchronous update, the program is assured that all data for the file has been written to permanent storage.

The following *file access mode* flags can be associated with a file:

O_RDONLY
The file is opened for reading only.

O_RDWR
The file is opened for reading and writing.

O_WRONLY
The file is opened for writing only.

Two masks can be used to extract flags:

O_ACCMODE
Extracts file access mode flags.

O_GETFL
Extracts file status flags and file access mode flags.

File locking

A process can use fcntl() to lock out other processes from a part of a file, so that the process can read or write to that part of the file without interference from others. File locking can ensure data integrity when several processes have a file accessed concurrently. File locking can only be performed on file descriptors that refer to regular files. Locking is not permitted on file descriptors that refer to directories, FIFOs, pipes, character special files, or any other type of files.

A structure that has the type `struct flock` (defined in the <fcntl.h> header file) controls locking operations. This structure has the following members:

`short l_type`
Indicates the type of lock, using one of the following symbols (defined in the <fcntl.h> header file):

F_RDLCK
Indicates a *read lock*, also called a *shared lock*. The process can read the locked part of the file, and other processes cannot obtain write locks for that part of the file in the meantime. More than one process can have a read lock on the same part of a file simultaneously.

To establish a read lock, a process must have the file accessed for reading.

F_WRLCK
Indicates a *write lock*, also called an *exclusive lock*. The process can write on the locked part of the file, and no other process can establish a read lock or write lock on that same part or on an overlapping part of the file. A process cannot put a write lock on part of a file if there is already a read lock on an overlapping part of the file. To establish a write lock, a process must have accessed the file for writing.

F_UNLCK
Unlocks a lock that was set previously. An unlock (F_UNLCK) request in which l_len is non-zero and the offset of the last byte of the requested segment is the maximum value for an object of type off_t, when the process has an existing lock in which l_len is 0 and which includes the last byte of the requested segment, is treated as a request to unlock from the start of the requested segment with an l_len equal to 0. Otherwise, an unlock (F_UNLCK) request attempts to unlock only the requested segment.

`short l_whence`
One of three symbols used to determine the part of the file that is affected by this lock. These symbols are defined in the <unistd.h> header file and are the same as symbols used by lseek():

SEEK_CUR
The current file offset in the file

SEEK_END
The end of the file

SEEK_SET
The start of the file.

`off_t l_start`
Gives the byte offset used to identify the part of the file that is affected by this lock. The part of the file affected by the lock begins at this offset from

the location given by l_whence. For example, if l_whence is SEEK_SET and l_start is 10, the locked part of the file begins at an offset of 10 bytes from the beginning of the file.

off_t l_len
Gives the size of the locked part of the file in bytes. If l_len is 0, the locked part of the file begins at the position specified by l_whence and l_start, and extends to the end of the file. If l_len is positive, the area affected starts at l_start and end at l_start+ l_len-1. If l_len is negative, the area affected starts at l_start+ l_len and end at l_start-1. Locks may start and extend beyond the current end of a file, but cannot extend before the beginning of the file. A lock can be set to extend to the largest possible value of the file offset for that file by setting l_len to 0. If such a lock also has l_start set to 0 and l_whence is set to SEEK_SET, the whole file is locked.

pid_t l_pid
Specifies the process ID of the process that holds the lock. This is an output field used only with F_GETLK actions.

You can set locks by specifying F_SETLK as the *action* argument for fcntl(). Such a function call requires a third argument pointing to a struct flock structure, as in this example:

```
struct flock lock_it;
lock_it.l_type = F_RDLCK;
lock_it.l_whence = SEEK_SET;
lock_it.l_start = 0;
lock_it.l_len = 100;
fcntl(fildes,F_SETLK,&lock_it);
```

This example sets up an flock structure describing a read lock on the first 100 bytes of a file, and then calls fcntl() to establish the lock. You can unlock this lock by setting l_type to F_UNLCK and making the same call. If an F_SETLK operation cannot set a lock, it returns immediately with an error saying that the lock cannot be set.

The F_SETLKW operation is similar to F_SETLK, except that it waits until the lock can be set. For example, if you want to establish an exclusive lock and some other process already has a lock established on an overlapping part of the file, fcntl() waits until the other process has removed its lock. If fcntl() is waiting in an F_SETLKW operation when a signal is received, fcntl() is interrupted. After handling the signal, fcntl() returns -1 and sets errno to EINTR.

F_SETLKW operations can encounter *deadlocks* when process A is waiting for process B to unlock a region, and B is waiting for A to unlock a different region. If the system detects that an F_SETLKW might cause a deadlock, fcntl() fails with errno set to EDEADLK.

A process can determine locking information about a file by using F_GETLK as the *action* argument for fcntl(). In this case, the call to fcntl() should specify a third argument pointing to an flock structure. The structure should describe the lock operation you want. When fcntl() returns, the structure indicated by the flock pointer is changed to show the first lock that would prevent the proposed lock operation from taking place. The returned structure shows the type of lock that is set, the part of the file that is locked, and the process ID of the process that holds the lock. In the returned structure:
- l_whence is always SEEK_SET.
- l_start gives the offset of the locked portion from the beginning of the file.

- `l_len` is the length of the locked portion.

If there are no locks that prevent the proposed lock operation, the returned structure has F_UNLCK in `l_type`, and is otherwise unchanged.

A process can have several locks on a file simultaneously but only one type of lock set on a given byte. Therefore, if a process puts a new lock on part of a file that it had locked previously, the process has only one lock on that part of the file: the type of the lock is the one specified in the most recent locking operation.

All of a process's locks on a file are removed when the process closes any file descriptor that refers to the locked file. Locks are not inherited by child processes created with fork().

All locks are advisory only. Processes can use locks to inform each other that they want to protect parts of a file, but locks do not prevent I/O on the locked parts. If a process has appropriate permissions on a file, it can perform whatever I/O it chooses, regardless of what locks are set. Therefore, file locking is only a convention, and it works only when all processes respect the convention.

Returned value

If successful, the value fcntl() returns will depend on the *action* that was specified.

If unsuccessful, fcntl() returns -1 and sets errno to one of the following values:

Error Code
Description

EAGAIN
The process tried to set a lock with F_SETLK, but the lock is in conflict with a lock already set by some other process on an overlapping part of the file.

EBADF
fildes is not a valid open file descriptor; or the process tried to set a read lock on a file descriptor open for writing only; or the process tried to set a write lock on a file descriptor open for reading only; or the *socket* parameter is not a valid socket descriptor.

In an F_DUPFD2 operation, the third argument is negative, or greater than or equal to **OPEN_MAX**, which is the highest file descriptor value allowed for the process.

EDEADLK
The system detected the potential for deadlock in a F_SETLKW operation.

EINTR
fcntl() was interrupted by a signal during a F_SETLKW operation.

EINVAL
In an F_DUPFD operation, the third argument is negative or greater than or equal to **OPEN_MAX**, the highest file descriptor value allowed for the process. The **OPEN_MAX** value can be determined using pathconf().

In a locking operation, *fildes* refers to a file with a type that does not support locking, or the `struct flock` pointed to by the third argument has an incorrect form.

If an F_CLOSFD operation, the third argument, which specifies the upper limit, is less than *filedes* but is not equal to -1.

Behavior for sockets: The *arg* parameter is not a valid flag, or the *cmd* parameter is not a valid command.

EMFILE
In an F_DUPFD operation, the process has already reached its maximum number of file descriptors, or there are no available file descriptors greater than the specified third argument.

ENOLCK
In an F_SETLK or F_SETLKW operation, the specified file has already reached the maximum number of locked regions allowed by the system.

EOVERFLOW
One of the values to be returned cannot be represented correctly.

The cmd argument is F_GETLK, F_SETLK or F_SETLKW and the smallest or, if l_len is nonzero, the largest offset of any byte in the requested segment cannot be represented correctly in an object of type `off_t`.

EPERM
The operation was F_CLOSFD, but all the requested file descriptors were not closed.

Examples

CELEBF06

```
/* CELEBF06

   This example illustrates one use of fcntl().
   The example will compile only with C/MVS.

 */
#define _POSIX_SOURCE
#include <fcntl.h>
#include <sys/types.h>
#include <unistd.h>
#undef _POSIX_SOURCE
#include <signal.h>
#include <stdio.h>

void catcher(int signum) {
  puts("inside catcher...");
}

main() {
  int p[2], flags;
  struct sigaction sact;
  char c;

  if (pipe(p) != 0)
    perror("pipe() error");
  else {
    sigemptyset(&sact.sa_mask);
    sact.sa_flags = 0;
    sact.sa_handler = catcher;
    sigaction(SIGALRM, &sact, NULL);

    alarm(10);

    if (read(p[0], &c, 1) == -1)
      perror("first read() failed");

    if ((flags = fcntl(p[0], F_GETFL)) == -1)
      perror("first fcntl() failed");
    else if (fcntl(p[0], F_SETFL, flags | O_NONBLOCK) == -1)
```

```
      perror("second fcntl() failed");
    else {
      alarm(10);

      if (read(p[0], &c, 1) == -1)
        perror("second read() failed");

      alarm(0);
    }
    close(p[0]);
    close(p[1]);
  }
}
```

Output

```
inside catcher...
first read() failed: Interrupted function call
second read() failed: Resource temporarily unavailable
```

Sockets example:

```
#define _OPEN_SYS
int s;
int rc;
int flags;
⋮
/* Place the socket into nonblocking mode */
rc = fcntl(s, F_SETFL, FNDELAY);

/* See if asynchronous notification is set */
flags = fcntl(s, F_GETFL, 0);
if (flags & FNDELAY)
   /* it is set */
else
   /* it is not */
```

Related information

- "fcntl.h" on page 27
- "sys/types.h" on page 75
- "unistd.h" on page 82
- "close() — Close a file" on page 288
- "dup() — Duplicate an open file descriptor" on page 404
- "dup2() — Duplicate an open file descriptor to another" on page 406
- "exec functions" on page 436
- "fsync() — Write changes to direct-access storage" on page 655
- "getsockopt() — Get the options associated with a socket" on page 777
- "ioctl() — Control device" on page 869
- "lseek() — Change the offset of a file" on page 1020
- "open() — Open a file" on page 1152
- "setsockopt() — Set options associated with a socket" on page 1578
- "socket() — Create a socket" on page 1688

fcvt() — Convert double to string

Standards

Standards / Extensions	C or C++	Dependencies
XPG4.2 Single UNIX Specification, Version 3	both	

Format

```
#define _XOPEN_SOURCE_EXTENDED 1
#include <stdlib.h>

char *fcvt(double x, int ndigit, int *__restrict__ decpt, int *__restrict__ sign);
```

General description

The fcvt() function converts double floating-point argument values to floating-point output strings. The fcvt() function has been extended to determine the floating-point format (hexadecimal floating-point or IEEE Binary Floating-Point) of double argument values by using __isBFP().

z/OS XL C/C++ formatted output functions, including the fcvt() function, convert IEEE Binary Floating-Point infinity and NaN argument values to special infinity and NaN floating-point number output sequences. See "*fprintf* Family of Formatted Output Functions" on fprintf(), printf(), sprintf() — Format and write data for a description of the special infinity and NaN output sequences.

The fcvt() function converts *x* to a NULL-terminated string which has *ndigit* digits to the right of the radix point (where the total number of digits in the output string is restricted by the precision of a double) and returns a pointer to the string. The function behaves identically to "ecvt() — Convert double to string" on page 418 in all respects other than the number of digits in the return value.

Note: This function has been moved to the Legacy Option group in Single UNIX Specification, Version 3 and may be withdrawn in a future version. The sprintf() function is preferred for portability.

Returned value

If successful, fcvt() returns the character equivalent of *x* as specified above.

If unable to allocate the return buffer, or the conversion fails, fcvt() returns NULL.

Related information

- "stdlib.h" on page 70
- "ecvt() — Convert double to string" on page 418
- "gcvt() — Convert double to string" on page 683
- "__isBFP() — Determine application floating-point format" on page 904

fdelrec() — Delete a VSAM record

Standards

Standards / Extensions	C or C++	Dependencies
Language Environment	both	

Format

```
#include <stdio.h>

int fdelrec(FILE *stream);
```

```
#define _OPEN_SYS_UNLOCKED_EXT 1
#include <stdio.h>

int fdelrec_unlocked(FILE *stream);
```

General description

Removes the record previously read by fread() from the VSAM file associated with *stream*. The fdelrec() function can only be used after an fread() call has been performed and before any other operation on that file pointer. For example, if you need to acquire the file position using ftell() or fgetpos(), you can do it either before the fread() or after the fdelrec(). An fread() after an fdelrec() will retrieve the next record.

To avoid infringing on the user's name space, this nonstandard function has two names. One name is prefixed with two underscore characters, and one name is not. The name without the prefix underscore characters is exposed only when you use LANGLVL(EXTENDED).

To use this function, you must either invoke the function using its external entry point name (that is, the name that begins with two underscore characters), or compile with LANGLVL(EXTENDED). When you use LANGLVL(EXTENDED) any relevant information in the header is also exposed.

The fdelrec() function can be used with *key sequenced data sets* (KSDS), KSDS PATHs, and *relative record data set* (RRDS) opened in an update mode (that is, `rb+/r+b`, `wb+/w+b`, or `ab+/a+b`), with `type=record`.

VSAM does not support deletions from ESDSs.

fdelrec_unlocked() is functionally equivalent to fdelrec() with the exception that it is not thread-safe. This function can safely be used in a multithreaded application if and only if it is called while the invoking thread owns the (FILE*) object, as is the case after a successful call to either the flockfile() or ftrylockfile() function.

Returned value

If successful, fdelrec() returns 0.

If unsuccessful, fdelrec() returns nonzero.

Example

```
/* This example shows how a VSAM record is deleted using the fdelrec()
   function.
 */
#include <stdio.h>
   FILE *stream;
   char buf[80];
   int num_read;
   int rc;
   stream = fopen("DD:MYCLUS", "rb+,type=record");
⋮
   num_read = fread(buf, 1, sizeof(buf), stream);
   rc = fdelrec(stream);
⋮
```

Related information

- “Performing VSAM I/O Operations” in *z/OS XL C/C++ Programming Guide*
- “stdio.h” on page 68
- “flocate() — Locate a VSAM record” on page 544
- “fupdate() — Update a VSAM record” on page 669

fdetach() — Detach a name from a STREAMS-based file descriptor

Standards

Standards / Extensions	C or C++	Dependencies
XPG4.2 Single UNIX Specification, Version 3	both	

Format

```
#define _XOPEN_SOURCE_EXTENDED 1
#include <stropts.h>

int fdetach(const char *path);
```

General description

The fdetach() function detaches a STREAMS-based file from the file to which it was attached by a previous call to fattach(). The *path* argument points to the path name of the attached STREAMS file. The process must have appropriate privileges or be the owner of the file. A successful call to fdetach() causes all path names that named the attached STREAMS file to again name the file to which the STREAMS file was attached. All subsequent operations on *path* will operate on the underlying file and not on the STREAMS file.

All open file descriptions established while the STREAMS file was attached to the file referenced by *path*, will still refer to the STREAMS file after the fdetach() has taken effect.

If there are no open file descriptors or other references to the STREAMS file, then a successful call to fdetach() has the same effect as performing the last close() on the attached file.

Returned value

If successful, fdetach() returns 0.

If unsuccessful, fdetach() returns -1 and sets errno to one of the following values.

Note: z/OS UNIX services do not supply any STREAMS devices or pseudodevices. It is impossible for fdetach() to detach a file from a STREAMS-based file descriptor. See “open() — Open a file” on page 1152 for more information.

Error Code
Description

EACCES
Search permission is denied on a component of the path prefix.

EINVAL
The *path* argument names a file that is not currently attached.

ELOOP
Too many symbolic links were encountered in resolving *path*.

ENAMETOOLONG
The size of a path name exceeds **PATH_MAX**, or a path name component is longer than **NAME_MAX**, or path name resolution of a symbolic link produced an intermediate result whose length exceeds **PATH_MAX**.

ENOENT
A component of *path* does not name an existing file or *path* is an empty string.

ENOTDIR
A component of the path prefix is not a directory.

EPERM
The effective user ID is not the owner of *path* and the process does not have appropriate privileges.

Related information

- "stropts.h" on page 72
- "fattach() — Attach a STREAMS-based file descriptor to a file in the file system name space" on page 462

fdim(), fdimf(), fdiml() — Calculate the positive difference

Standards

Standards / Extensions	C or C++	Dependencies
C99 Single UNIX Specification, Version 3 C++ TR1 C99	both	z/OS V1R5

Format

```
#define _ISOC99_SOURCE
#include <math.h>

double fdim(double x, double y);
float fdimf(float x, float y);
long double fdiml(long double x, long  double y);
```

C++ TR1 C99:

```
#define _TR1_C99
#include <math.h>

float fdim(float x, float y);
long double fdim(long double x, long double y);
```

General description

The fdim functions compute the positive difference between *x* and *y*.

Note: The following table shows the viable formats for these functions. See "IEEE binary floating-point" on page 94 for more information about IEEE Binary Floating-Point.

Function	Hex	IEEE
fdim	X	X
fdimf	X	X
fdiml	X	X

Restriction: The fdimf() function does not support the _FP_MODE_VARIABLE feature test macro.

Returned value

The fdim functions return the positive difference between *x* and *y*.

Related information

- "math.h" on page 44

fdimd32(), fdimd64(), fdimd128() — Calculate the positive difference

Standards

Standards / Extensions	C or C++	Dependencies
C/C++ DFP	both	z/OS V1.8

Format

```
#define __STDC_WANT_DEC_FP__
#include <math.h>

_Decimal32  fdimd32(_Decimal32 x, _Decimal32 y);
_Decimal64  fdimd64(_Decimal64 x, _Decimal64 y);
_Decimal128 fdimd128(_Decimal128 x, _Decimal128 y);
_Decimal32  fdim(_Decimal32 x, _Decimal32 y);     /* C++ only */
_Decimal64  fdim(_Decimal64 x, _Decimal64 y);     /* C++ only */
_Decimal128 fdim(_Decimal128 x, _Decimal128 y);   /* C++ only */
```

General description

The fdim functions compute the positive difference between *x* and *y*.

Notes:

1. To use IEEE decimal floating-point, the hardware must have the Decimal Floating-Point Facility installed.
2. These functions work in IEEE decimal floating-point format. See "IEEE binary floating-point" on page 94 for more information

Returned value

The fdim functions return the positive difference between *x* and *y*.

Example

```
/* CELEBF76

   This example illustrates the fdimd32() function.

*/

#define  __STDC_WANT_DEC_FP__
#include <math.h>
#include <stdio.h>

int main(void)
{
   _Decimal32 x = 56789.70DF, y = 56790.00DF, z;

   z = fdimd32(y, x);

   printf("The result of fdim32(%Hf, %Hf)\n is %Hf\n", y, x, z);
   return 0;
}
```

Related information

- “math.h” on page 44
- “fdim(), fdimf(), fdiml() — Calculate the positive difference” on page 489

fdopen() — Associate a stream with an open file descriptor

Standards

Standards / Extensions	C or C++	Dependencies
POSIX.1 XPG4 XPG4.2 Single UNIX Specification, Version 3	both	

Format

```
#define _POSIX_SOURCE
#include <stdio.h>

FILE *fdopen(int fildes, const char *options);
```

General description

Associates a stream with an open file descriptor. A *stream* is a pointer to a FILE structure that contains information about a file. A stream permits user-controllable buffering and formatted input and output. For a discussion of the z/OS UNIX services implementation of buffering, see *z/OS XL C/C++ Programming Guide*.

The specified *options* must be permitted by the current mode of the file descriptor. For example, if the file descriptor is open-read-only (O_RDONLY), the corresponding stream cannot be opened write-only (w).

These options are the same as for an fopen() operation.

Special behavior for XPG4.2: The values for options are changed to include binary streams.

Mode Description

r or rb
Open for reading.

w or wb
Open for writing.

a or ab
Open for appending.

r+ or rb+ or r+b
Open for update (reading and writing).

w+ or wb+ or w+b
Open for update (reading and writing).

a+ or ab+ or a+b
Open for update at End Of File (EOF) (reading and writing).

All these options have the same behavior as the corresponding fopen() options, except that w, wb, w+, wb+ and w+b do not truncate the file.

The file position indicator of the new stream is the file offset associated with the file descriptor. The error indicator and end of file (EOF) indicator for the stream are cleared.

Returned value

If successful, fdopen() returns a FILE pointer to the control block for the new stream.

If unsuccessful, fdopen() returns NULL and sets errno to one of the following values:

Error Code
Description

EBADF
fildes is not a valid open file descriptor.

EINVAL
The specified mode is incorrect or does not match the mode of the open file descriptor.

Example

CELEBF08

```
/* CELEBF08

   This example associates stream with the file descriptor fd which is
   open for the file fdopen.file.
   The association is made in write mode.

 */
#define _POSIX_SOURCE
#include <stdio.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <unistd.h>

main() {
  char fn[]="fdopen.file";
  FILE *stream;
  int  fd;
  if ((fd = creat(fn, S_IWUSR)) < 0)
    perror("creat() error");
  else {
    if ((stream = fdopen(fd, "w")) == NULL) {
      perror("fdopen() error");
      close(fd);
    }
    else {
      fputs("This is a test", stream);
      fclose(stream);
    }}}
```

Related information

- “stdio.h” on page 68
- “fileno() — Get the file descriptor from an open stream” on page 536
- “fopen() — Open a file” on page 565
- “open() — Open a file” on page 1152

feclearexcept() — Clear the floating-point exceptions

Standards

Standards / Extensions	C or C++	Dependencies
C99 Single UNIX Specification, Version 3 C++ TR1 C99	both	z/OS V1R7

Format

```
#define _ISOC99_SOURCE
#include <fenv.h>

int feclearexcept (int excepts);
```

General description

feclearexcept() clears the supported floating-point exceptions represented by *excepts*.

Note: The following table shows the viable formats for these functions. See “IEEE binary floating-point” on page 94 for more information about IEEE Binary Floating-Point.

Function	Hex	IEEE
feclearexcept		X

Returned value

If successful, feclearexcept() returns 0 if the argument passed is 0 or if all the exceptions are successfully cleared.

Related information

-

fe_dec_getround() — Get the current rounding mode

Standards

Standards / Extensions	C or C++	Dependencies
C/C++ DFP	both	z/OS V1.8

Format

```
#define __STDC_WANT_DEC_FP__
#include <fenv.h>

int fe_dec_getround(void);
```

General description

The fe_dec_getround function gets the current rounding mode for decimal floating-point operations.

The following rounding modes are defined for decimal floating-point, and are located in fenv.h:

FE_DEC_DOWNWARD
rounds towards minus infinity

FE_DEC_TONEAREST
rounds to nearest

FE_DEC_TOWARDZERO
rounds toward zero

FE_DEC_UPWARD
rounds toward plus infinity

FE_DEC_TONEARESTFROMZERO
rounds to nearest, ties away from zero

_FE_DEC_AWAYFROMZERO
rounds away from zero

_FE_DEC_TONEARESTTOWARDZERO
rounds to nearest, ties toward zero

_FE_DEC_PREPAREFORSHORTER
rounds to prepare for shorter precision

Notes:

1. To use IEEE decimal floating-point, the hardware must have the Decimal Floating-Point Facility installed.
2. This function works in IEEE decimal floating-point format. See "IEEE Decimal Floating-Point" for more information.

Returned value

If successful, returns the value of the current rounding mode for decimal floating-point operations.

If there is no such rounding mode or the current rounding mode can't be determined returns -1.

Example

```
/* CELEBF77

   sample program that issues fe_dec_getround()/setround()

   This program calls fe_dec_getround() to get the DFP rounding mode.
   Then it will compare the returned value with FE_DEC_TONEAREST
   rounding mode. If not the same, it will call fe_dec_setround() to
   set the rounding mode to the desired value FE_DEC_TONEAREST.
 */

#define __STDC_WANT_DEC_FP__
#include <fenv.h>
#include <stdlib.h>
#include <stdio.h>

int main(int argc, char *argv[]){
   int r;
   if ((r = fe_dec_getround()) == -1){
     perror("fe_dec_getround");
     exit (-1);
   }
   printf("The Decimal floating point rounding mode is %d\n", r);
   if (r != FE_DEC_TONEAREST){
     printf("The DFP rounding mode is not FE_DEC_TONEAREST.\n");
     if (fe_dec_setround(FE_DEC_TONEAREST) == -1){
       perror("fe_dec_setround");
       exit (-1);
     }
   }
   printf("The DFP rounding mode has been set to FE_DEC_TONEAREST.\n");
   return 0;
}
```

Related information

- “fenv.h” on page 28
- “fe_dec_setround() — Set the current rounding mode”

fe_dec_setround() — Set the current rounding mode

Standards

Standards / Extensions	C or C++	Dependencies
C/C++ DFP	both	z/OS V1.8

Format

```
#define __STDC_WANT_DEC_FP__
#include <fenv.h>

int fe_dec_setround(int round);
```

General description

The fe_dec_setround function establishes the rounding mode for decimal floating-point operations represented by its argument *round*. If the argument is not equal to the value of a valid rounding mode, the rounding mode is not changed.

The following rouding modes are defined for decimal floating-point, and are located in fenv.h:

FE_DEC_DOWNWARD
rounds towards minus infinity

FE_DEC_TONEAREST
rounds to nearest

FE_DEC_TOWARDZERO
rounds toward zero

FE_DEC_UPWARD
rounds toward plus infinity

FE_DEC_TONEARESTFROMZERO
rounds to nearest, ties away from zero

_FE_DEC_AWAYFROMZERO
rounds away from zero

_FE_DEC_TONEARESTTOWARDZERO
rounds to nearest, ties toward zero

_FE_DEC_PREPAREFORSHORTER
rounds to prepare for shorter precision

Notes:

1. To use IEEE decimal floating-point, the hardware must have the Decimal Floating-Point Facility installed.
2. This function works in IEEE decimal floating-point format. See "IEEE Decimal Floating-Point" for more information.

Returned value

If successful, returns a zero value.

If the argument is not equal to a valid rounding mode, returns -1. The following errnos are defined:

Error Code
Definition

EINVAL
The rounding mode specified is not a valid Decimal Floating Point rounding mode.

EMVSERR
The function was unable to set the specified rounding mode due to an internal error.

Example

```
/* CELEBF77

   sample program that issues fe_dec_getround()/setround()

   This program calls fe_dec_getround() to get the DFP rounding mode.
   Then it will compare the returned value with FE_DEC_TONEAREST
   rounding mode. If not the same, it will call fe_dec_setround() to
   set the rounding mode to the desired value FE_DEC_TONEAREST.
 */

#define __STDC_WANT_DEC_FP__
#include <fenv.h>
#include <stdlib.h>
```

```
#include <stdio.h>

int main(int argc, char *argv[]){
   int r;
   if ((r = fe_dec_getround()) == -1){
     perror("fe_dec_getround");
     exit (-1);
   }
   printf("The Decimal floating point rounding mode is %d\n", r);
   if (r != FE_DEC_TONEAREST){
     printf("The DFP rounding mode is not FE_DEC_TONEAREST.\n");
     if (fe_dec_setround(FE_DEC_TONEAREST) == -1){
       perror("fe_dec_setround");
       exit (-1);
     }
   }
   printf("The DFP rounding mode has been set to FE_DEC_TONEAREST.\n");
   return 0;
}
```

Related information

- "fenv.h" on page 28
- "fe_dec_getround() — Get the current rounding mode" on page 494

fegetenv() — Store the current floating-point environment

Standards

Standards / Extensions	C or C++	Dependencies
C99 Single UNIX Specification, Version 3 C/C++ DFP C++ TR1 C99	both	z/OS V1R8

Format

```
#define _ISOC99_SOURCE
#include <fenv.h>

int fegetenv(fenv_t *envp);
```

General description

fegetenv() stores the current floating-point environment in the object pointed to by *envp*.

Note: The following table shows the viable formats for these functions. See "IEEE binary floating-point" on page 94 for more information about IEEE Binary Floating-Point.

Function	Hex	IEEE
fegetenv		X

Notes:

1. To use IEEE decimal floating-point, the hardware must have the Decimal Floating-Point Facility installed.
2. If the hardware has the Decimal Floating-Point Facility installed, this function will store the decimal floating-point rounding mode.

3. This function works in IEEE decimal floating-point format. See "IEEE Decimal Floating-Point" for more information.

Returned value

If successful, fegetenv() returns 0 upon completion of the store.

Related information

- “fenv.h” on page 28
- “fesetenv() — Set the floating-point environment” on page 504
- “feholdexcept() — Save the current floating-point environment” on page 499
- “feupdateenv() — Save the currently raised floating-point exceptions” on page 522

fegetexceptflag() — Store the states of floating-point status flags

Standards

Standards / Extensions	C or C++	Dependencies
C99 Single UNIX Specification, Version 3 C++ TR1 C99	both	z/OS V1R7

Format

```
#define _ISOC99_SOURCE
#include <fenv.h>

int fegetexceptflag(fexcept_t *flagp, int excepts);
```

General description

fegetexceptflag() stores an implementation defined representation of the states of floating-point status flags indicated by *excepts* in the object pointed to by *flagp*.

Note: The following table shows the viable formats for these functions. See “IEEE binary floating-point” on page 94 for more information about IEEE Binary Floating-Point.

Function	Hex	IEEE
fegetexceptflag		X

Returned value

If successful, fegetexceptflag() returns 0 upon completion of the store.

Related information

-

fegetround() — Get the current rounding mode

Standards

Standards / Extensions	C or C++	Dependencies
C99 Single UNIX Specification, Version 3 C++ TR1 C99	both	z/OS V1R7

Format

```
#define _ISOC99_SOURCE
#include <fenv.h>

int fegetround(void);
```

General description

fegetround() gets the current rounding mode.

Note: The following table shows the viable formats for these functions. See “IEEE binary floating-point” on page 94 for more information about IEEE Binary Floating-Point.

Function	Hex	IEEE
fegetround		X

Note: This function will not return or update decimal floating-point rounding mode bits. The *fe_dec_getround()* and *fe_dec_setround()* functions can be used to get and set the current rounding mode for decimal floating-point operations.

Returned value

If successful, fegetround() returns the value of the rounding mode macro representing the current rounding mode. Otherwise, returns a negative value if there is no such rounding mode macro or the current rounding mode is not determinable.

Related information

- “fe_dec_getround() — Get the current rounding mode” on page 494
- “fe_dec_setround() — Set the current rounding mode” on page 495

feholdexcept() — Save the current floating-point environment

Standards

Standards / Extensions	C or C++	Dependencies
C99 Single UNIX Specification, Version 3 C/C++ DFP C++ TR1 C99	both	z/OS V1R8

Format

```
#define _ISOC99_SOURCE
#include <fenv.h>

int feholdexcept(fenv_t *envp);
```

General description

feholdexcept() saves the current floating-point environment in the object pointed to by *envp*, clears the floating-point status flags, and then installs a non-stop (continue on floating-point exceptions) mode, if available, for all floating-point exceptions.

Note: The following table shows the viable formats for these functions. See "IEEE binary floating-point" on page 94 for more information about IEEE Binary Floating-Point.

Function	Hex	IEEE
feholdexcept		X

Notes:

1. To use IEEE decimal floating-point, the hardware must have the Decimal Floating-Point Facility installed.
2. If the hardware has the Decimal Floating-Point Facility installed, this function will save the decimal floating-point rounding mode.
3. This function works in IEEE decimal floating-point format. See "IEEE Decimal Floating-Point" for more information.

Returned value

If successful, feholdexcept() returns 0 when the non-stop floating-point exception handling was successfully installed.

Related information

- "fenv.h" on page 28
- "fegetenv() — Store the current floating-point environment" on page 497
- "fesetenv() — Set the floating-point environment" on page 504
- "feupdateenv() — Save the currently raised floating-point exceptions" on page 522

feof() — Test end of file (EOF) indicator

Standards

Standards / Extensions	C or C++	Dependencies
ISO C POSIX.1 XPG4 XPG4.2 C99 Single UNIX Specification, Version 3 Language Environment	both	

Format

```
#include <stdio.h>

int feof(FILE *stream);
```

```
#define _OPEN_SYS_UNLOCKED_EXT 1
#include <stdio.h>

int feof_unlocked (FILE *stream);
```

General description

Indicates whether the EOF flag is set for the given stream pointed to by *stream*.

The EOF flag is set when the user attempts to read past the EOF. Thus, a read of the last character in the file does not turn the flag on. A subsequent read attempt reaches the EOF.

For HFS files, a simultaneous reader cannot see the extensions automatically. Use clearerr() is required to reset the EOF flag.

If the file has a simultaneous writer that extends the file, the flag can be turned on by the reader before the file is extended. After the extension becomes visible to the reader, a subsequent read will get the new data and set the flag appropriately (see fflush()). For example, if the read does not read past the EOF, the flag is turned off. If a file does not have a simultaneous writer that is extending the file, it is not possible to read past EOF.

A successful repositioning in a file (with fsetpos(), rewind(), fseek()) or a call to clearerr() resets the EOF flag. For a terminal file, when the EOF flag is set, subsequent reads will continue to deliver no data until the EOF flag is cleared. This can be accomplished by calling clearerr() or rewind().

The terminal can only read past the EOF after the rewind() function or the clearerr() function is called. The EOF flag is cleared by calling rewind(), fsetpos(), fseek(), or clearerr() for this stream.

feof_unlocked() is functionally equivalent to feof() with the exception that it is not thread-safe. This function can safely be used in a multithreaded application if and only if it is called while the invoking thread owns the (FILE*) object, as is the case after a successful call to either the flockfile() or ftrylockfile() function.

Returned value

If and only if the EOF flag is set for *stream*, feof() returns nonzero.

Otherwise, feof() returns 0.

Example

CELEBF09

```
/* CELEBF09

   This example scans the input stream until it reads an EOF character.

 */
#include <stdio.h>
#include <stdlib.h>
```

```
main() {

      FILE *stream;
      int rc;
      stream = fopen("myfile.dat","r");

/* myfile.dat contains 3 characters "abc" */
      while (1) {
         rc = fgetc(stream);
         if (rc == EOF) {
            if (feof(stream)) {
               printf("at EOF\n");
               break;
            }
            else {
               printf("error\n");
               break;
            }
         }
         else
            printf("read %c\n",rc);
      }
}
```

Output

```
read a
read b
read c
at EOF
```

Related information

- “stdio.h” on page 68
- “clearerr() — Reset error and end of file (EOF)” on page 284
- “fseek() — Change file position” on page 639
- “fsetpos() — Set file position” on page 649
- “rewind() — Set file position to beginning of file” on page 1445

feraiseexcept() — Raise the supported floating-point exceptions

Standards

Standards / Extensions	C or C++	Dependencies
C99 Single UNIX Specification, Version 3 C++ TR1 C99	both	z/OS V1R7

Format

```
#define _ISOC99_SOURCE
#include <fenv.h>

int feraiseexcept(int excepts);
```

General description

feraiseexcept() raises the supported floating-point exceptions represented by *excepts*. The order in which these floating-point exceptions are raised is unspecified.

Note: The following table shows the viable formats for these functions. See "IEEE binary floating-point" on page 94 for more information about IEEE Binary Floating-Point.

Function	Hex	IEEE
feraiseexcept		X

Returned value

If successful, feraiseexcept() returns 0 if the argument passed is 0 or if all the exceptions are successfully raised.

Related information

-

ferror() — Test for read and write errors

Standards

Standards / Extensions	C or C++	Dependencies
ISO C POSIX.1 XPG4 XPG4.2 C99 Single UNIX Specification, Version 3 Language Environment	both	

Format

```
#include <stdio.h>

int ferror(FILE *stream);

#define _OPEN_SYS_UNLOCKED_EXT 1
#include <stdio.h>

int ferror_unlocked (FILE *stream);
```

General description

Tests for an error in reading from or writing to the specified *stream*. If an error occurs, the error indicator for the *stream* remains set until you close the *stream*, call rewind(), or call clearerr().

If a non-valid parameter is given to an I/O function, z/OS XL C/C++ does not turn the error flag on. This case differs from one where parameters are not valid in context with one another.

ferror_unlocked() is functionally equivalent to ferror() with the exception that it is not thread-safe. This function can safely be used in a multithreaded application if and only if it is called while the invoking thread owns the (FILE*) object, as is the case after a successful call to either the flockfile() or ftrylockfile() function.

Returned value

If successful, ferror() returns a nonzero value to indicate an error for the stream pointed to by *stream*.

If unsuccessful, ferror() returns 0.

Example

CELEBF10

```
/* CELEBF10

   This example puts data out to a stream and then checks that
   a write error has not occurred.

 */
#include <stdio.h>

int main(void)
{
   FILE *stream;
   char *string = "Important information";
   stream = fopen("myfile.dat","w");

   fprintf(stream, "%s\n", string);
   if (ferror(stream))
   {
      printf("write error\n");
      clearerr(stream);
   }
   if (fclose(stream))
      printf("fclose error\n");
}
```

Related information

- “stdio.h” on page 68
- “clearerr() — Reset error and end of file (EOF)” on page 284
- “rewind() — Set file position to beginning of file” on page 1445

fesetenv() — Set the floating-point environment

Standards

Standards / Extensions	C or C++	Dependencies
C99 Single UNIX Specification, Version 3 C/C++ DFP C++ TR1 C99	both	z/OS V1R8pd

Format

```
#define _ISOC99_SOURCE
#include <fenv.h>

int fesetenv(const fenv_t *envp);
```

General description

fesetenv() establishes the floating-point environment represented by the object pointed to by *envp*. The argument *envp* points to an object set by a call to fegetenv() or feholdexcept(), or equal to a floating-point environment macro. fesetenv() merely installs the state of the floating-point status flags represented through *envp* and does not raise these floating-point exceptions.

Note: The following table shows the viable formats for these functions. See "IEEE binary floating-point" on page 94 for more information about IEEE Binary Floating-Point.

Function	**Hex**	**IEEE**
fesetenv		X

Notes:

1. If the hardware has the Decimal Floating-Point Facility installed, this function can be used to set the decimal floating-point rounding mode.
2. This function works in IEEE decimal floating-point format. See "IEEE Decimal Floating-Point" for more information.

Returned value

If successful, fesetenv() returns 0 if the settings are restored.

If unsuccessful, fesetenv() returns -1 and sets one of the following errno values:

Error Code
Description

EINVAL
The rounding mode specified is not a valid Decimal Floating Point rounding mode.

EMVSERR
The function was unable to set the specified rounding mode due to an internal error.

Related information

- "fenv.h" on page 28
- "fegetenv() — Store the current floating-point environment" on page 497
- "feholdexcept() — Save the current floating-point environment" on page 499
- "feupdateenv() — Save the currently raised floating-point exceptions" on page 522

fesetexceptflag() — Set the floating-point status flags

Standards

Standards / Extensions	**C or C++**	**Dependencies**
C99 Single UNIX Specification, Version 3 C++ TR1 C99	both	z/OS V1R7

Format

```
#define _ISOC99_SOURCE
#include <fenv.h>

int fesetexceptflag(const fexcept_t *flagp, int excepts);
```

General description

fesetexceptflag() sets the floating-point status flags indicated by *excepts* to the states stored in the object pointed to by *flagp*. The value of *flagp* should be set by fegetexceptflag(), whose second argument represents the floating-point exceptions indicated by the argument *excepts*. This function does not raise floating-point exceptions, but only sets the state of the flags.

Note: The following table shows the viable formats for these functions. See "IEEE binary floating-point" on page 94 for more information about IEEE Binary Floating-Point.

Function	Hex	IEEE
fesetexceptflag		X

Returned value

If successful, fesetexceptflag() returns 0 if *excepts* is 0 or if all selected exceptions are successfully set.

Related information

-

fesetround() — Set the current rounding mode

Standards

Standards / Extensions	C or C++	Dependencies
C99 Single UNIX Specification, Version 3 C++ TR1 C99	both	z/OS V1R7

Format

```
#define _ISOC99_SOURCE
#include <fenv.h>

int fesetround(int round);
```

General description

fesetround() establishes the rounding mode represented by *round*. If the argument is not equal to the value of a rounding mode macro, the rounding mode is not changed.

Note: The following table shows the viable formats for these functions. See "IEEE binary floating-point" on page 94 for more information about IEEE Binary Floating-Point.

Function	Hex	IEEE
fesetround		X

Note: This function will not return or update decimal floating-point rounding mode bits. The *fe_dec_getround()* and *fe_dec_setround()* functions can be used to get and set the current rounding mode for decimal floating-point operations.

Returned value

If successful, fesetround() returns 0 when *round* is set to a rounding mode.

Related information

- "fe_dec_getround() — Get the current rounding mode" on page 494
- "fe_dec_setround() — Set the current rounding mode" on page 495

fetch() — Get a load module

Standards

Standards / Extensions	C or C++	Dependencies
Language Environment	C	

Format

```
#include <stdlib.h>

void (*fetch(const char *name))();
```

General description

Dynamically loads the load module specified by *name* into memory. The load module can then be invoked from a z/OS XL C program. The name or the alias by which the fetchable load module is identified in the load module library must appear in a fetch() library function call.

To avoid infringing on the user's name space, this nonstandard function has two names. One name, the external entry point name, is prefixed with two underscore characters, and the other name is not. The name without the prefix underscore characters is exposed only when you use LANGLVL(EXTENDED).

To use this function, you must either invoke the function using its external entry point name (that is, the name that begins with two underscore characters __fetch()) or compile with LANGLVL(EXTENDED). When you use LANGLVL(EXTENDED), any relevant information in the header is also exposed.

You cannot fetch a module that contains a main(). If you do, fetch() will not return a usable pointer. Use of the pointer will result in undefined behavior. To call these types of modules, use the system() library function. Alternatively, when creating the module, you can reset the entry point so that the linkage is provided by z/OS XL C.

When non-reentrant modules have been fetched multiple times, you should release them in the reverse order; otherwise the load modules may not be deleted immediately.

You can fetch modules written in C and C++. For C modules, the source of the fetched module must, in general, contain `#pragma linkage(..., fetchable)` (the exception is described below). To fetch a C++ module, the routine must be declared extern "C", and must be declared in a **`#pragma linkage`**`(..., fetchable)` directive. See also "fetchep() — Share writable static" on page 519 for more information about the need for `#pragma linkage`.

Note: For C or C++ modules that are compiled with the XPLINK option and are to be fetched, #pragma linkage(..., fetchable) is required. They cannot use the technique of resetting the entry point. If an application tries to fetch() an XPLINK routine that did not specify FETCHABLE, then an error will be returned.

XPLINK programs can fetch non-XPLINK programs, and vice versa. The function descriptor returned by fetch() will contain glue code to support a stack swap and parameter list conversion if necessary. Calls to a fetched routine that do not cross an XPLINK linkage boundary will not incur any glue code overhead.

If the fetched module is compiled as a DLL it can import variables and functions from other DLL modules, but it cannot export variables or functions.

Nested fetching is supported. That is, a fetched module can also invoke the fetch() library function to dynamically load a separate fetchable module.

Multiple fetching is also supported. Fetching a module more than once will result in separate fetch pointers. If the module is marked "reentrant", multiple fetches will not reload the module into storage. Under MVS, you can place the reentrant module into the Extended Link Pack Area or the Link Pack Area (ELPA/LPA) to save time on the load. Although multiple copies of the reentrant module are not brought into storage, each fetch returns a separate pointer. If a module is not reentrant, multiple fetches cause multiple loads into storage. Be aware that if you fetch() a non-reentrant module multiple times, the module may not get deleted by release() until all fetch instances have been released. Also, you should keep in mind that multiple loads of a non-reentrant module can be costly in terms of storage.

Writable statics are, in general, process-scoped. The exception is that, when a thread calls a fetched module, the writable statics are changed for that thread only, that is, thread-scoped.

Under MVS, fetchable (or dynamically loaded) modules must be link-edited and accessible using the standard system search. MVS supports fetching of non-reentrant, serially reusable, and reentrant modules.

Under POSIX, however, the fetchable, dynamically loaded modules cannot be in the HFS (Hierarchical File System). Note also, that the POSIX and XPG4 external variables are propagated. Refer to *z/OS XL C/C++ Programming Guide* for more information on external variables.

Unless your program is naturally reentrant, each reentrant module has a different copy of writable static. Follow these steps to allow the fetching of your reentrant module that has writable static:

1. Compile the module to be fetched with the RENT compile-time option.
2. Using the object module created in step 1, generate a fetchable member. You must specify the entry point as the function you are fetching unless you have included a **`#pragma linkage(...,`** `FETCHABLE)` directive.

See Figure 2 on page 509 for the program flow of a fetchable module. (FECB refers to a Fetch Control Block, which is a z/OS XL C internal control block used by fetch().)

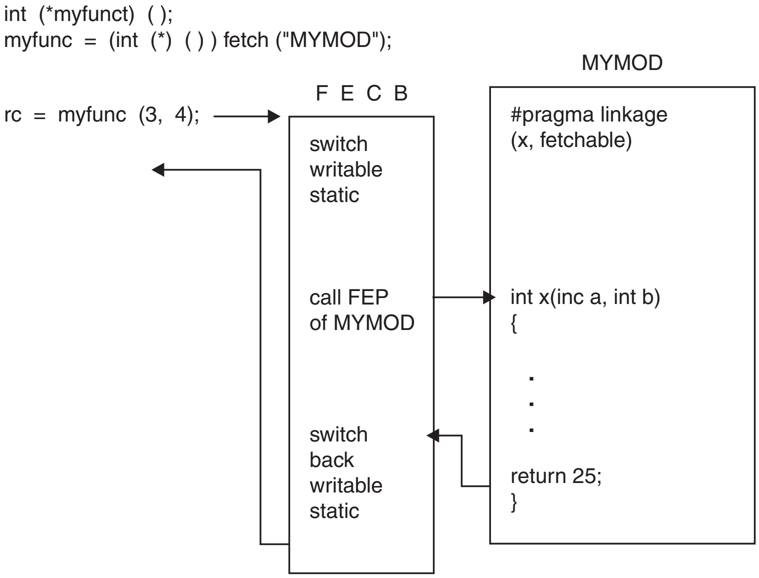


Figure 2. Program Flow of a Fetchable Module

To dynamically fetch a set of functions with shared writable static, you can use the library function fetchep(). See “fetchep() — Share writable static” on page 519 for more details.

Both the module being fetched and the module invoking the fetch() library function can be reentrant.

You can fetch modules without specifying the directive, `#pragma linkage(..., FETCHABLE)`, in the fetched module. If you do, then using the fetch pointer will result in calling the entry point for that module. When you link the module, you must reset the entry point. In addition, you cannot have any writable static.

It follows that fetching a reentrant C module containing writable statics requires that you use the `#pragma linkage(... FETCHABLE)` preprocessor directive in the fetched module.

If the entry point linkage is not a C linkage, you must use a `pragma linkage` with a function pointer defined by a *typedef*. The following sample excerpt would set up a COBOL linkage for a COBOL routine.

```
typedef int COBOL_FUNC ();
#pragma linkage (COBOL_FUNC, COBOL)
⋮
COBOL_FUNC * fetch_ptr;
fetch_ptr = (COBOL_FUNC *) fetch(module);    /* loads fetched module */
fetch_ptr(args);         /* sets up the proper linkage for the call */
```

Once the module is fetched, calling the fetched function is similar to making an interlanguage call.

fetch() also supports `AMODE` switching: when the function call is made, `AMODE` will be switched; upon return, the `AMODE` will be restored. Beware of calling fetched modules with `AMODE=24` that try to access variables or the library above the line.

Notes:

1. You cannot call functions through a function pointer that crosses load module boundaries, except through fetchep(). (See “fetchep() — Share writable static” on page 519 for more information.) For example, you cannot pass the address of a function to a fetched routine and invoke it from the fetched routine because the z/OS C writable static will not be swapped.
2. If you need to access code that has to run in a restricted addressing mode (such as a AMODE 24), you can package the code into a module to be fetched. The module can then be linked using the restricted addressing mode, but fetched from a program with an unrestricted addressing mode.
3. A program that invokes fetch() many times without releasing all of the load modules may run out of memory.

Link considerations: When linking the function to be fetched, you must link in the necessary libraries and specify the entry point as the function you are fetching unless you have included this directive: `#pragma linkage(..., FETCHABLE)`.

When linking the main() z/OS C function, you must specify the necessary libraries to use the functions you are fetching. For example, if you are fetching a COBOL function, specify the COBOL library. This requirement does *not* apply to Language Environment.

When running main(), specify the runtime libraries you will need for main(), as well as the functions you will fetch. This requirement does *not* apply to Language Environment.

Special behavior for C++: A z/OS XL C++ program cannot call fetch(). If you attempt to call fetch() from a z/OS XL C++ program, the compiler will issue an error message. There are three alternatives to fetch() under z/OS XL C++:

- You can replace fetch() with DLL (dynamic link library) calls.
- You can provide a C DLL module to fetch modules, as shown in examples CELEBF52 and CELEBF53. See “Examples.”
- A z/OS XL C++ program may statically call a z/OS XL C function that, in turn, fetches another module.

Returned value

If successful, fetch() returns a pointer to a stub that will call the entry point to the fetched load module.

If the load fails, fetch() returns NULL and may set errno to one of the following values:

Error Code
  Definition

ELEFENCE
  The DLL contains a member language not supported on this version of the operating system.

Examples

Examples of using the fetch() function with C: The following example demonstrates how to compile, link, and run a program that fetches a function in another object module that contains the directive:
`pragma linkage(...., FETCHABLE)`.

Begin with the main program.

```
#include <stdio.h>
#include <stdlib.h>

typedef int (*funcPtr)();  /* pointer to a function returning an int */

int main(void)
{
    int result;
    funcPtr add;

    printf("fetch module\n");
    add = (funcPtr) fetch("f1a");            /* load module F1A      */

    if (add == NULL) {
        printf("ERROR: fetch failed\n");
    }
    else {
        printf("execute fetched module\n");
        result = (*add)(1,2);                /* execute module F1A   */

        printf("1 + 2 == %d\n", result);
    }
}
```

Then the fetched function:

```
#pragma linkage(func1, fetchable)

int func1(int a, int b)
{
   printf("in fetched module\n");

   return(a+b);
}
```

Next, JCL to compile, link, and run under MVS:

```
>
//F1A      EXEC EDCC,INFILE='userid.TEST.SOURCE(F1A)'
//         OUTFILE='userid.TEST.OBJ(F1A),DISP=SHR',
//         CPARM='NOSEQ,NOMARGIN,RENT'
//F1B      EXEC EDCPL,INFILE='userid.TEST.OBJ(F1A)'
//         OUTFILE='userid.TEST.LOAD(F1A),DISP=SHR'
//F1       EXEC EDCCLG,INFILE='userid.TEST.SOURCE(F1)'
//GO.STEPLIB DD
//           DD  DSN=userid.TEST.LOAD,DISP=SHR
```

This example demonstrates the use of fetch() with COBOL and how to compile, link, and run the program.

```
/* cob1 */
#include <stdlib.h>
#include <stdio.h>

typedef void funcV();           /* function returning void            */
#pragma linkage(funcV, COBOL)   /* establish Cobol linkage            */

int main(void)
{
    int var1 = 1;
    int var2 = 2;
    funcV *add;

    printf("fetch module\n");
    add = (funcV *) fetch("cob1a");           /* load module COB1A    */
    if (add == NULL)
    {
        printf("ERROR: fetch failed\n");
    }
    else
    {
        printf("execute fetched module\n");
        (*add)(&var1, &var2);                 /* execute module COB1A */

        printf("1 + 2 == %d\n", var1);
    }
}
```

Here is the fetched COBOL subroutine COB1A.

```
 IDENTIFICATION DIVISION.
 PROGRAM-ID.  COB1A.
****************************************************************
*   This subroutine receives 2 integer parameters.             *
*   The first is added to the second and the result is stored  *
*   back into the first.                                       *
****************************************************************
 ENVIRONMENT DIVISION.
 DATA DIVISION.

 WORKING-STORAGE SECTION.

 LINKAGE SECTION.
 01  VAR1                    PIC S9(9) COMP.
 01  VAR2                    PIC S9(9) COMP.
****************************************************************
*                  PROCEDURE DIVISION                          *
****************************************************************

 PROCEDURE DIVISION USING VAR1 VAR2.

*
*    ADD VAR2 TO VAR1 PLACING THE RESULT IN VAR1.
*

     COMPUTE VAR1 = VAR1 + VAR2.
           GOBACK.
```

Finally, compile, link, and run under MVS:

```
//*==================================================================
//COBCL  PROC CREGSIZ='2048K',
//            INFILE=,
// OUTFILE='&&GSET(GO),DISP=(MOD,PASS),UNIT=VIO,SPACE=(512,(50,20,1))'
//*
//*------------------------------------------------------------------
//* COBOL Compile Step
//*------------------------------------------------------------------
//COBCOMP EXEC PGM=IGYCRCTL,REGION=&CREGSIZ;
//STEPLIB  DD  DSNAME=IGY.V1R3M0.SIGYCOMP,DISP=SHR
//SYSPRINT DD  SYSOUT=*
//SYSIN    DD  DSNAME=&INFILE,DISP=SHR
//SYSLIN   DD  DSNAME=&&LOADSET,UNIT=SYSDA,
//             DISP=(MOD,PASS),SPACE=(TRK,(3,3)),
//             DCB=(BLKSIZE=3200)
//SYSUT1   DD  UNIT=SYSDA,SPACE=(CYL,(1,1))
//SYSUT2   DD  UNIT=SYSDA,SPACE=(CYL,(1,1))
//SYSUT3   DD  UNIT=SYSDA,SPACE=(CYL,(1,1))
//SYSUT4   DD  UNIT=SYSDA,SPACE=(CYL,(1,1))
//SYSUT5   DD  UNIT=SYSDA,SPACE=(CYL,(1,1))
//SYSUT6   DD  UNIT=SYSDA,SPACE=(CYL,(1,1))
//SYSUT7   DD  UNIT=SYSDA,SPACE=(CYL,(1,1))
//*
```

```
//*------------------------------------------------------------------
//* COBOL Link-Edit Step
//*------------------------------------------------------------------
//COBLINK EXEC PGM=HEWL,COND=(8,LT,COBCOMP),REGION=1024K
//SYSLIB   DD  DSNAME=CEE.V1R3M0.SCEELKED,DISP=SHR
//SYSPRINT DD  SYSOUT=*
//SYSLIN   DD  DSNAME=&&LOADSET,DISP=(OLD,DELETE)
//         DD  DDNAME=SYSIN
//SYSLMOD  DD  DSNAME=&OUTFILE;
//SYSUT1   DD  UNIT=SYSDA,SPACE=(TRK,(10,10))
// PEND
//*
//*==================================================================
//* Compile and Link-Edit COBOL program COB1A
//*------------------------------------------------------------------
//COB1A   EXEC COBCL,
//         INFILE='userid.TEST.SOURCE(COB1A)',
//         OUTFILE='userid.TEST.LOAD(COB1A),DISP=SHR'
//COBLINK.SYSIN DD *
 ENTRY COB1A
/*
//*
//*------------------------------------------------------------------
//* Compile and Link-Edit C program COB1
//*------------------------------------------------------------------
//COB1    EXEC EDCCLG,
//         INFILE='userid.TEST.SOURCE(COB1)',
//         CPARM='OPT(0) NOSEQ NOMAR'
//GO.STEPLIB DD
//           DD DSNAME=userid.TEST.LOAD,DISP=SHR
```

Examples of alternatives to fetch() under C++: This example shows how to use DLL as an alternative to fetch(). Here, myfunc() is the function to be dynamically loaded using DLL, and main() invokes DLL.

CELEBF52

```
// CELEBF52-part 1 of 2-other file is CELEBF53.

// This example shows how to use DLL as an alternative to fetch().

// C++ program that invokes myfunc using DLL

#include <stdlib.h>
#include <stdio.h>
#include <dll.h>

extern "C" {           // needed to indicate C linkage
  typedef int (*funcPtr)(); // pointer to a function returning an int
}

int main (void)
{
  dllhandle *dllh;
  funcPtr   fptr;

  if ((dllh = dllload( "celebf53" )) == NULL) {
     perror( "failed to load celebf53" );
     exit( -1 );
  }
  if ((fptr = (funcPtr) dllqueryfn( dllh, "myfunc" )) == NULL) {
     perror( "failed to retrieve myfunc" );
     exit( -2 );
  }
  if ( fptr() != 0 ) {
     perror( "failed to execute myfunc" );
```

```
      exit( -3 );
   }
   if ( dllfree( dllh ) != 0 ) {
      perror( "failed to free celebf53" );
      exit( -4 );
   }
   return( 0 );
}
```

CELEBF53

```
/* CELEBF53-part 2 of 2-other file is CELEBF52.

   This example shows how to use DLL as an alternative to fetch().

 */

/*

  C function dynamically loaded using DLL

 */

#include <stdio.h>

int myfunc (void)
{
  printf( "Hello world\n" );
  return( 0 );
}
```

The following example shows how a C++ program can dynamically call a function in a C DLL module, to fetch other C modules.

CELEBF54

```
// CELEBF54-part 1 of 3-other files are CELEBF55, CELEBF56.
// This example shows how a C++ program can dynamically call a function
// in a C DLL module, to fetch other C modules

// C++ program that dynamically calls a function in a C DLL module

#include <stdio.h>
#include <stdlib.h>
#include <dll.h>
#include <iostream.h>

extern "C" {           // needed to indicate C linkage
  typedef int (*funcPtr)(); // pointer to a function returning an int
}

int main (void)
{
  dllhandle *dllh;
  funcPtr   fptr;

  if ((dllh = dllload( "mydll" )) == NULL) {
     perror( "failed to load mydll" );
     exit( -1 );
  }
  if ((fptr = (funcPtr) dllqueryfn( dllh, "fwrap" )) == NULL) {
     perror( "failed to retrieve fwrap" );
     exit( -2 );
  }
  if ( fptr() != 0 ) {
     perror( "failed to execute fwrap" );
     exit( -3 );
  }
  if ( dllfree( dllh ) != 0 ) {
     perror( "failed to free mydll" );
     exit( -4 );
  }
  return( 0 );
}
```

CELEBF55

```
/* CELEBF55-part 2 of 3-other files are CELEBF54, CELEBF56.
   This example shows how a C++ program can dynamically call a function
   in a C DLL module, to fetch other C modules

   fwrap function used in a DLL module-it fetches mymod, which
   contains myfunc
 */
#include <stdio.h>
#include <stdlib.h>

typedef int (*funcPtr)();   /* pointer to a function returning an int */

int fwrap (void)
{
  funcPtr   fptr;

  if ((fptr = (funcPtr) fetch( "mymod" )) == NULL) {
     perror( "failed to fetch mymod" );
     return( -1 );
  }
  else
     return(fptr());
}
```

CELEBF56

```
/* CELEBF56-part 3 of 3-other files are CELEBF54, CELEBF55.
   This example shows how a C++ program can dynamically call a function
   in a C DLL module, to fetch other C modules
 */

/*    C function to be fetched   */

#include <stdio.h>
#pragma linkage(myfunc, fetchable)

int myfunc (void)
{
  printf( "in fetched module\n" );
  return( 0 );
}
```

The following example shows how to statically call a C function that in turn fetches other functions. Here, myfunc() is the function to be fetched, fetcher() is a C function that fetches myfunc(), and main() is a function that statically calls fetcher().

CELEBF57

```
// CELEBF57-part 1 of 3-other files are CELEBF58, CELEBF59.
// This example shows how to statically call a C function that
// fetches other functions.

// C++ statically calling a C program that uses fetch()

#include <iostream.h>
```

```
extern "C" {          //  needed to indicate C linkage
  int fetcher (void);
}

int main (void)
{
  cout << "The fetcher says: ";
  fetcher();
  cout << "and returns";
  return( 0 );
}
```

CELEBF58

```
/* CELEBF58-part 2 of 3-other files are CELEBF57, CELEBF59.
   This example shows how to statically call a C function that fetches
   other functions.
 */

/*
   C function that fetches mymod which contains myfunc
 */
#include <stdio.h>
#include <stdlib.h>

typedef int (*funcPtr)();   /* pointer to a function returning an int */

int fetcher (void)
{
  funcPtr fptr;

  if ((fptr = (funcPtr) fetch( "mymod" )) == NULL) {
     perror( "failed to fetch mymod" );
     return( -1 );
  }
  else {
     fptr();               /* invoke fetched function */
     return( 0 );
  }
}
```

CELEBF59

```
/* CELEBF59-part 3 of 3-other files are CELEBF57, CELEBF58.
   This example shows how to statically call a C function that fetches
   other functions.
 */

/*    C function to be fetched    */

#include <stdio.h>
#pragma linkage(myfunc, fetchable)

int myfunc (void)
{
  printf( "Hello world " );
  return( 0 );
}
```

Although fetching and using DLL are functionally comparable, there is one subtle difference. Fresh copies of static and global variables are allocated each time a module is fetched, but not each time a DLL load of the same module is done.

The following example shows that, when a module is fetched multiple times, fresh copies of static and global variables are allocated.

CELEBF60

```
/* CELEBF60-part 1 of 2-other file is CELEBF61.
   This example shows how copies of variables are allocated when multiple
   fetches are done.
 */

/*
   C program fetching mymod multiple times--mymod contains myfunc.
 */
#include <stdio.h>
#include <stdlib.h>

typedef int (*funcPtr)(int); /*pointer to a function returning an int*/

int main (void)
{
  funcPtr fptr1, fptr2;

  if ((fptr1 = (funcPtr) fetch( "mymod" )) == NULL) {
     perror( "failed to fetch mymod" );
     return( -1 );
  }
  if ( fptr1(100) != 0 ) {
     perror( "failed to execute myfunc" );
     exit( -2 );
  }
  if ((fptr2 = (funcPtr) fetch( "mymod" )) == NULL) {
     perror( "failed to fetch mymod" );
     return( -3 );
  }
  if ( fptr2(100) != 0 ) {
     perror( "failed to execute myfunc" );
     exit( -4 );
  }
  return( 0 );
}
```

CELEBF61

```
/* CELEBF61-part 2 of 2-other file is CELEBF60.
   This example shows how copies of variables are allocated when multiple
   fetches are done.
 */

/*     C module mymod    */
#include <stdio.h>
#pragma linkage(myfunc, fetchable)

int globvar = 5;

int myfunc (int x)
{
  globvar += x;
  printf( "%d\n", globvar );
  return( 0 );
}
```

Running this example would produce the following results:

```
105
105
```

The following example shows that fresh copies of static and global variables are not allocated for multiple DLL loads of the same module.

CELEBF62

```
// CELEBF62-part 1 of 2-other file is CELEBF63.
// This example shows how copies of variables are allocated when
// multiple DLL loads are done.

// C++ program doing multiple DLL loads of the same module

#include <stdlib.h>
#include <stdio.h>
#include <dll.h>

extern "C" {                    //needed to indicate C linkage
  typedef int (*funcPtr)(int); //pointer to a function returning an int
}

int main (void)
{
  dllhandle *dllh1, *dllh2;
  funcPtr fptr;

  if ((dllh1 = dllload( "mydll" )) == NULL) {
     perror( "failed to load mydll" );
     exit( -1 );
  }
  if ((fptr = (funcPtr) dllqueryfn( dllh1, "myfunc" )) == NULL) {
     perror( "failed to retrieve myfunc" );
     exit( -2 );
  }
  if ( fptr(100) != 0 ) {
     perror( "failed to execute myfunc" );
     exit( -3 );
  }
  if ((dllh2 = dllload( "mydll" )) == NULL) {
     perror( "failed to load mydll" );
     exit( -4 );
  }
  if ((fptr = (funcPtr) dllqueryfn( dllh2, "myfunc" )) == NULL) {
     perror( "failed to retrieve myfunc" );
     exit( -5 );
  }
  if ( fptr(100) != 0 ) {
     perror( "failed to execute myfunc" );
     exit( -6 );
  }
  if ( dllfree( dllh1 ) != 0 ) {
     perror( "failed to free mydll" );
     exit( -7 );
  }
  return( 0 );
}
```

CELEBF63

```
/* CELEBF63-part 2 of 2-other file is CELEBF62.
   This example shows how copies of variables are allocated when multiple
   DLL loads are done.
 */

/* C function invoked using DLL */

#include <stdio.h>
#include <stdlib.h>

int globvar = 5;
int myfunc (int);

int myfunc (int x)
{
  globvar += x;
  printf( "%d\n", globvar );
  return( 0 );
}
```

Running this example would produce the following results:

```
105
205
```

Related information

- See the topic about processing a program with multithreading in *z/OS XL C/C++ Programming Guide*.
- "stdlib.h" on page 70
- "fetchep() — Share writable static"
- "release() — Delete a load module" on page 1424

fetchep() — Share writable static

Standards

Standards / Extensions	C or C++	Dependencies
Language Environment	C only	

Format

```
#include <stdlib.h>

void ( *fetchep( void ( *entry_point)()))();
```

General description

Dynamically fetches a set of functions with shared writable static variables. fetchep() is used to register an entry point. It returns a pointer that may be passed across the fetch boundary and used as if it were the original entry point. Therefore, you can create more than one entry point from a fetched module. A call to the new entry point will use the same writable static as the original fetch pointer uses on each invocation.

fetchep() is called within a fetched module but not from the same level as the fetch() call. If fetchep() is called in the root program that is not a fetched module, fetchep() returns a fetch pointer that will use the root program's writable static (if any exists).

If the *entry_point* given as input to fetchep() is a function address external to the current module or is an non-valid function address, use of the resulting pointer returned from the call will result in undefined behavior.

If writable static is required, then this directive must be used:

```
#pragma linkage(entry_point, FETCHABLE)
```

In addition, the steps for fetching a reentrant module must be followed as described in "fetch() — Get a load module" on page 507. If writable static is *not* required, the C module using fetchep() need not contain the directive: #pragma linkage(..., FETCHABLE).

You can release the new fetch pointer without any effect on the original or any other fetch pointer created from the original fetch pointer. If the original fetched function is released, however, all the fetch pointers created using the fetchep() function will also be released. Trying to use a fetch pointer once it has been released or its origin has been released will result in undefined behavior.

To avoid infringing on the user's name space, this nonstandard function has two names. One name, the external entry point name, is prefixed with two underscore characters, and the other name is not. The name without the prefix underscore characters is exposed only when you use LANGLVL(EXTENDED).

Note: The external entry point name for fetchep() is __ftchep(), **NOT** __fetchep().

To use this function, you must either invoke the function using its external entry point name (that is, the name that begins with two underscore characters __ftchep(), or compile with LANGLVL(EXTENDED). When you use LANGLVL(EXTENDED) any relevant information in the header is also exposed.

Examples

These examples and diagram demonstrate the program flow of a call to fetch() and subsequent calls to fetchep().

```
/* The module that calls fetch()    */
#include <stdlib.h>
typedef int (*FUNC_T)();

int main(void) {
FUNC_T (*myfunc)();
FUNC_T myfunc1;
FUNC_T myfunc2;
FUNC_T myfunc3;

myfunc = (FUNC_T (*)())fetch("MYMOD");
myfunc1 = myfunc(0);
myfunc2 = myfunc(1);
myfunc3 = myfunc(2);
}
```

```
/*
   The following code is the fetched module.
   Please see fetch() for information on how to compile and link the
   above.
 */

/*   inside MYMOD   */
#include <stdlib.h>
typedef int (*FUNC_T)();
int k;              /* global variable to share within MYMOD */
#pragma linkage(x, fetchable)
FUNC_T x(int a)
{
  switch(a)
    {
    case 0:
       return (FUNC_T)fetchep((void(*)())func1);
    case 1:
       return (FUNC_T)fetchep((void(*)())func2);
    case 2:
       return (FUNC_T)fetchep((void(*)())func3);
    }
{

int func1(int a, int b)
```

```
{
k = 6;
⋮
}

int func2(int a, int b)
{
k = 4;
⋮
}

int func3(int a, int b)
{
k = 5;
⋮
}
```

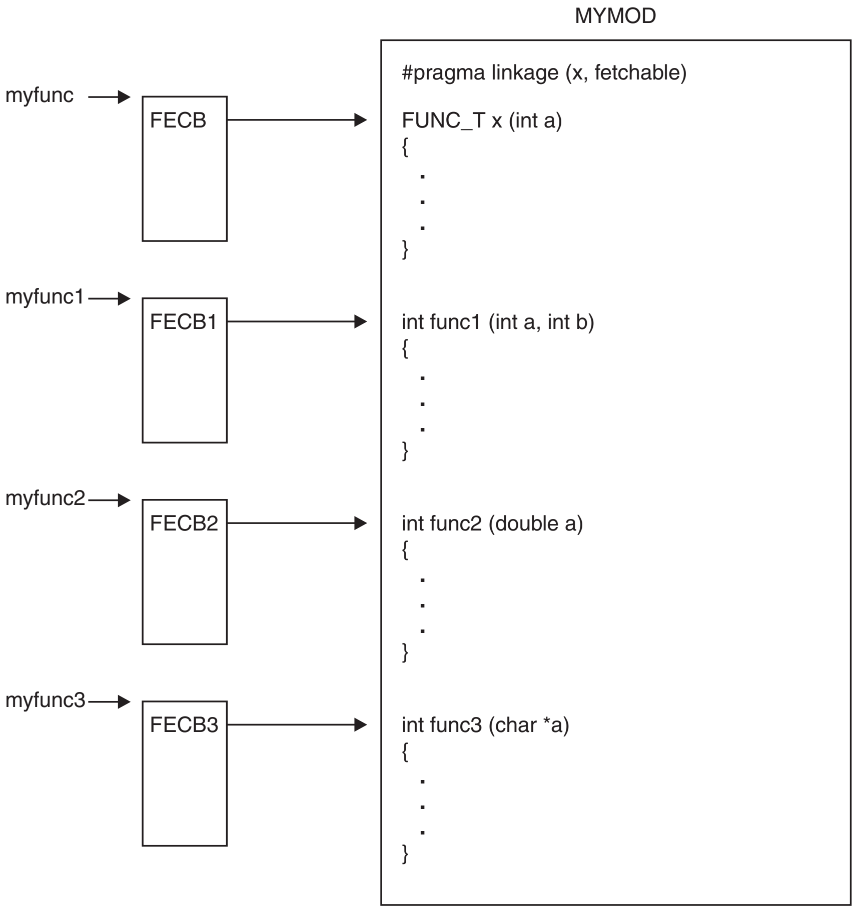


Figure 3. Program Flow of fetchep()

Related information

- “stdlib.h” on page 70
- “fetch() — Get a load module” on page 507
- “release() — Delete a load module” on page 1424

fetestexcept() — Test the floating-point status flags

Standards

Standards / Extensions	C or C++	Dependencies
C99 Single UNIX Specification, Version 3 C++ TR1 C99	both	z/OS V1R7

Format

```
#define _ISOC99_SOURCE
#include <fenv.h>

int fetestexcept(int excepts);
```

General description

fetestexcept() determines which of a specified subset of floating-point exception flags are currently set. *excepts* specifies the floating-point status flags to be queried.

Note: The following table shows the viable formats for these functions. See “IEEE binary floating-point” on page 94 for more information about IEEE Binary Floating-Point.

Function	Hex	IEEE
fetestexcept		X

Returned value

If successful, fetestexcept() returns the value of the bitwise OR for the floating-point exception macros corresponding to the currently set floating-point exceptions included in *excepts*.

feupdateenv() — Save the currently raised floating-point exceptions

Standards

Standards / Extensions	C or C++	Dependencies
C99 Single UNIX Specification, Version 3 C/C++ DFP C++ TR1 C99	both	z/OS V1R8

Format

```
#define _ISOC99_SOURCE
#include <fenv.h>

int feupdateenv(const fenv_t *envp);
```

General description

feupdateenv() saves the currently raised floating-point exceptions in its automatic storage, installs the floating-point environment represented by the object pointed to by *envp*, and then raises the saved floating-point exceptions. *envp* should point to an object set by feholdexcept() or fegetenv(), or equal a floating-point environment macro.

Note: The following table shows the viable formats for these functions. See "IEEE binary floating-point" on page 94 for more information about IEEE Binary Floating-Point.

Function	Hex	IEEE
feupdateenv		X

Notes:

1. To use IEEE decimal floating-point, the hardware must have the Decimal Floating-Point Facility installed.
2. If the hardware has the Decimal Floating-Point Facility installed, this function can update the decimal floating-point rounding mode.
3. This function works in IEEE decimal floating-point format. See "IEEE Decimal Floating-Point" for more information.

Returned value

If successful, feupdateenv() returns 0 if the settings have been restored.

If unsuccessful, feupdateenv() returns -1 and sets one of the following errno values:

Error Code
Description

EINVAL
The rounding mode specified is not a valid Decimal Floating Point rounding mode.

EMVSERR
The function was unable to set the specified rounding mode due to an internal error.

Related information

- "fenv.h" on page 28
- "fegetenv() — Store the current floating-point environment" on page 497
- "fesetenv() — Set the floating-point environment" on page 504
- "feholdexcept() — Save the current floating-point environment" on page 499

fflush() — Write buffer to file

Standards

Standards / Extensions	C or C++	Dependencies
ISO C POSIX.1 XPG4 XPG4.2 C99 Single UNIX Specification, Version 3 Language Environment	both	

Format

```
#include <stdio.h>

int fflush(FILE *stream);

#define _OPEN_SYS_UNLOCKED_EXT 1
#include <stdio.h>

int fflush_unlocked(FILE *stream);
```

General description

Flushes the stream pointed to by *stream*. If *stream* is NULL, it flushes all open streams.

The fflush() function is affected by the ungetc() and ungetwc() functions. Calling these functions causes fflush() to back up the file position when characters are pushed back. For details, see the ungetc() and ungetwc() functions respectively. If desired, the _EDC_COMPAT environment variable can be set at open time such that fflush() discards any pushed-back characters and leaves the file position where it was when the first ungetc() or ungetwc() function call was issued.

If fflush() is used after ungetwc() has pushed a wide char on a text stream, the position will be backed up by one wide character from the position the file was at when the ungetwc() was issued. For a wide-oriented binary stream, the position will be backed up based on the number of bytes used to represent the wide char in the ungetc buffer. For this reason, attempting to use ungetwc() on a character when the destination is a binary wide-oriented stream that was never read in the first place results in undefined behavior for fflush(). Note that the _EDC_COMPAT environment variable also changes the behavior of fflush() after ungetwc(), and will cause any wide char pushed-back to be discarded and the position left at the point where the ungetwc() was issued. For details on the _EDC_COMPAT environment variable, see the “Environment Variables” in *z/OS XL C/C++ Programming Guide*.

If fflush() fails, the position is left at the point in the file where the first ungetc() or ungetwc() function call was issued. All pushed-back characters are discarded.

Note: The system automatically flushes buffers when you close the stream or when a program ends normally without closing the stream.

The buffering mode and the file type can have an effect on when output data is flushed. For more information, see “Buffering of C Streams” in *z/OS XL C/C++ Programming Guide*.

stream remains open after the fflush() call. Because a read operation cannot immediately follow or precede a write operation, the fflush() function can be used to allow exchange between these two modes. The fflush() function can also be used to refresh the buffer when working with a reader and a simultaneous writer/updater.

fflush_unlocked() is functionally equivalent to fflush() with the exception that it is not thread-safe. This function can safely be used in a multithreaded application if and only if it is called while the invoking thread owns the (FILE*) object, as is the case after a successful call to either the flockfile() or ftrylockfile() function.

The fflush() function has no effect for files opened with `type=blocked`.

Returned value

If successful in flushing the buffer, fflush() returns 0.

If unsuccessful, fflush() returns EOF. When flushing all open files, a failure to flush any of the files causes EOF to be returned. However, flushing will continue on any other open files that can be flushed successfully.

Example

CELEBF15

```
/* CELEBF15

   This example flushes a stream buffer.
   It tests for the returned value of 0 to see if the flushing was
   successful.

 */
#include <stdio.h>

int retval;
int main(void)
{
   FILE *stream;
   stream = fopen("myfile.dat", "w");

   retval=fflush(stream);
   printf("return value=%i",retval);
}
```

Related information

- "stdio.h" on page 68
- "setbuf() — Control buffering" on page 1522
- "setvbuf() — Control buffering" on page 1596
- "ungetc() — Push character onto input stream" on page 1958
- "ungetwc() — Push a wide character onto a stream" on page 1961

ffs() — Find first set bit in an integer

Standards

Standards / Extensions	C or C++	Dependencies
XPG4.2 Single UNIX Specification, Version 3	both	

Format

```
#define  _XOPEN_SOURCE_EXTENDED 1
#include <strings.h>

int ffs(int i);
```

General description

The ffs() function finds the first bit set (beginning with the least significant bit) and returns the index of that bit. Bits are numbered starting at one (the least significant bit).

Returned value

If successful, ffs() returns the index of the first bit set.

If *i* is 0, ffs() returns 0.

There are no errno values defined.

Related information

- "strings.h" on page 72

fgetc() — Read a character

Standards

Standards / Extensions	C or C++	Dependencies
ISO C POSIX.1 XPG4 XPG4.2 C99 Single UNIX Specification, Version 3 Language Environment	both	

Format

```
#include <stdio.h>

int fgetc(FILE *stream);

#define _OPEN_SYS_UNLOCKED_EXT 1
#include <stdio.h>

int fgetc_unlocked(FILE *stream);
```

General description

Reads a single-byte unsigned character from the input stream pointed to by *stream* at the current position, and increases the associated file pointer so that it points to the next character.

The fgetc() function is not supported for files opened with `type=record` or `type=blocked`.

fgetc() has the same restriction as any read operation for a read immediately following a write or a write immediately following a read. Between a write and a subsequent read, there must be an intervening flush or reposition. Between a read and a subsequent write, there must also be an intervening flush or reposition unless an EOF has been reached.

fgetc_unlocked() is functionally equivalent to fgetc() with the exception that it is not thread-safe. This function can safely be used in a multithreaded application if and only if it is called while the invoking thread owns the (FILE*) object, as is the case after a successful call to either the flockfile() or ftrylockfile() function.

Returned value

If successful, fgetc() returns the character read as an integer.

If unsuccessful, fgetc() returns EOF to indicate an error or an EOF condition. Use feof() or ferror() to determine whether the EOF value indicates an error or the end of the file.

Note: EOF is only reached when an attempt is made to read "past" the last byte of data. Reading up to and including the last byte of data does *not* turn on the EOF indicator.

Example

CELEBF16

```
/* CELEBF16

   This example gathers a line of input from a stream.
   It tests to see if the file can be opened.
   If the file cannot be opened &perror. is called.

 */
#include <stdio.h>
#define  MAX_LEN  80

int main(void)
{
   FILE *stream;
   char buffer[MAX_LEN + 1];
   int i, ch;

   if ((stream = fopen("myfile.dat","r")) != NULL) {
     for (i = 0; (i < (sizeof(buffer)-1) &&
           ((ch = fgetc(stream)) != EOF) && (ch != '\n')); i++)
        printf("character is %d\n",ch);
        buffer[i] = ch;

     buffer[i] = '\0';

     if (fclose(stream))
        perror("fclose error");
   }
   else
     perror("fopen error");
}
```

Related information

- "stdio.h" on page 68
- "feof() — Test end of file (EOF) indicator" on page 500
- "ferror() — Test for read and write errors" on page 503
- "fgetwc() — Get next wide character" on page 532
- "fputc() — Write a character" on page 603
- "getc(), getchar() — Read a character" on page 688

fgetpos() — Get file position

Standards

Standards / Extensions	C or C++	Dependencies
ISO C XPG4 XPG4.2 C99 Single UNIX Specification, Version 3 Language Environment	both	

Format

```
#include <stdio.h>

int fgetpos(FILE * __restrict__stream, fpos_t * __restrict__pos);

#define _OPEN_SYS_UNLOCKED_EXT 1
#include <stdio.h>

int fgetpos_unlocked(FILE * __restrict__stream, fpos_t * __restrict__pos);
```

General description

The fgetpos() function stores the current value of the file pointer associated with *stream* into the object pointed to by *pos*. The value pointed to by *pos* can be used later in a call to fsetpos() to reposition the stream pointed to by *stream*.

Both the fgetpos() and fsetpos() function save state information for wide-oriented files. The value stored in *pos* is unspecified, and it is usable only by fsetpos().

The position returned by the fgetpos() function is affected by the ungetc() and ungetwc() functions. Each call to these functions causes the file position indicator to be backed up from the position where the ungetc() or ungetwc() function was issued. For details on how the ungetc() function affects the fgetpos() function behavior, see “ungetc() — Push character onto input stream” on page 1958. For details on how the ungetwc() function affects the fgetpos() function behavior for a wide-oriented stream, see “ungetwc() — Push a wide character onto a stream” on page 1961.

Multivolume data sets performance: Using the fgetpos() and fsetpos() functions generally results in better repositioning performance compared to the ftell() and fseek() functions when working with multivolume data sets.

Large file support for MVS data sets, VSAM data sets, and z/OS UNIX files: The fgetpos() function implicitly supports operating on large files. Defining the _LARGE_FILES feature test macro is not required to use this function on large files.

Usage notes

1. The _EDC_COMPAT environment variable can be set at open time such that the fgetpos() function will ignore any pushed-back characters. For further details on _EDC_COMPAT, see the environment variables topic in *z/OS XL C/C++ Programming Guide*.
2. The fgetpos_unlocked() function is functionally equivalent to the fgetpos() function with the exception that it is not threadsafe. The fgetpos() function can

safely be used in a multithreaded application if, and only if, iit is called while the invoking thread owns the (FILE*) object, as is the case after a successful call to either the flockfile() or ftrylockfile() functions.

Returned value

If successful, the fgetpos() function returns 0.

If unsuccessful, the fgetpos() function returns nonzero and sets errno to nonzero.

Special behavior for XPG4.2: The fgetpos() function returns -1 and sets errno to ESPIPE if the underlying file type for the stream is a PIPE or a socket.

Example

CELEBF17

```
/* CELEBF17

   This example opens the file myfile.dat for reading.
   The current file pointer position is stored into the variable pos.

 */
#include <stdio.h>

int main(void)
{
   FILE *stream;
   int retcode;
   fpos_t pos;

   stream = fopen("myfile.dat", "rb");

   /* The value returned by fgetpos can be used by fsetpos()
      to set the file pointer if 'retcode' is 0            */

   if ((retcode = fgetpos(stream, &pos)) == 0)
      printf("Current position of file pointer found\n");
   fclose(stream);
}
```

Related information

- “stdio.h” on page 68
- “fseek() — Change file position” on page 639
- “fsetpos() — Set file position” on page 649
- “ftell() — Get current file position” on page 657
- “ungetc() — Push character onto input stream” on page 1958
- “ungetwc() — Push a wide character onto a stream” on page 1961

fgets() — Read a string from a stream

Standards

Standards / Extensions	C or C++	Dependencies
ISO C POSIX.1 XPG4 XPG4.2 C99 Single UNIX Specification, Version 3 Language Environment	both	

Format

```
#include <stdio.h>

char *fgets(char * __restrict__string, int n, FILE * __restrict__stream);

#define _OPEN_SYS_UNLOCKED_EXT 1
#include <stdio.h>

char *fgets_unlocked(char * __restrict__string, int n, FILE * __restrict__stream);
```

General description

Reads bytes from a stream pointed to by *stream* into an array pointed to by *string*, starting at the position indicated by the file position indicator. Reading continues until the number of characters read is equal to *n*-1, or until a newline character (`\n`), or until the end of the stream, whichever comes first. The fgets() function stores the result in *string* and adds a NULL character (\0) to the end of the string. The *string* includes the newline character, if read.

The fgets() function is not supported for files opened with `type=record` or `type=blocked`.

fgets() has the same restriction as any read operation for a read immediately following a write or a write immediately following a read. Between a write and a subsequent read, there must be an intervening flush or reposition. Between a read and a subsequent write, there must also be an intervening flush or reposition unless an EOF has been reached.

fgets_unlocked() is functionally equivalent to fgets() with the exception that it is not thread-safe. This function can safely be used in a multithreaded application if and only if it is called while the invoking thread owns the (FILE*) object, as is the case after a successful call to either the flockfile() or ftrylockfile() function.

Returned value

If successful, fgets() returns a pointer to the *string* buffer.

If unsuccessful, fgets() returns NULL to indicate failure.

If *n* is less than or equal to 0, it indicates a domain error; errno is set to EDOM to indicate the cause of the failure.

When *n* equals 1, it indicates a valid result. It means that the string buffer has only room for the NULL terminator; nothing is physically read from the file. (Such an operation is still considered a read operation, so it cannot immediately follow a write operation unless there is an intervening flush or reposition operation first.)

If *n* is greater than 1, fgets() will only fail if an I/O error occurs or if EOF is reached, and no data is read from the file.

The ferror() and feof() functions are used to distinguish between a read error and an EOF. Note that EOF is only reached when an attempt is made to read "past" the last byte of data. Reading up to and including the last byte of data does *not* turn on the EOF indicator.

If EOF is reached after data has already been read into the string buffer, fgets() returns a pointer to the string buffer to indicate success. A subsequent call would result in NULL being returned since EOF would be reached without any data being read.

Example

CELEBF18

```
/* CELEBF18

   This example gets a line of input from a data stream.
   It reads no more than MAX_LEN - 1 characters, or up
   to a new-line character, from the stream.

 */
#include <stdio.h>
#define  MAX_LEN  100

int main(void)
{
   FILE *stream;
   char line[MAX_LEN], *result;

   stream = fopen("myfile.dat","r");

   if ((result = fgets(line,MAX_LEN,stream)) != NULL)
       printf("The string is %s\n", result);

   if (fclose(stream))
       printf("fclose error\n");
}
```

Related information

- "stdio.h" on page 68
- "feof() — Test end of file (EOF) indicator" on page 500
- "ferror() — Test for read and write errors" on page 503
- "fgetc() — Read a character" on page 526
- "fgetws() — Get a wide-character string" on page 533
- "fputs() — Write a string" on page 605
- "gets() — Read a string" on page 769
- "puts() — Write a string" on page 1353

fgetwc() — Get next wide character

Standards

Standards / Extensions	C or C++	Dependencies
ISO C Amendment XPG4 XPG4.2C99 Single UNIX Specification, Version 3 Language Environment	both	

Format

```
#include <wchar.h>

wint_t fgetwc(FILE *stream);

#define _OPEN_SYS_UNLOCKED_EXT 1
#include <wchar.h>

wint_t fgetwc_unlocked(FILE *stream);
```

General description

Obtains the next multibyte character from the input stream pointed to by *stream*, converts it to a wide character, and advances the associated file position indicator for the stream (if defined).

The behavior of this wide-character function is affected by the LC_CTYPE category of the current locale. If you change the category, undefined results can occur.

Using non-wide-character functions with fgetwc() results in undefined behavior. This happens because fgetwc() processes a whole multibyte character and does not expect to be "within" such a character. In addition, fgetwc() expects state information to be set already. Because functions like fgetc() and fputc() do not obey such rules, their results fail to meet the assumptions made by fgetwc().

fgetwc() has the same restriction as any read operation for read immediately following a write or a write immediately following a read. Between a write and a subsequent read, there must be an intervening flush or reposition. Between a read and a subsequent write, there must also be an intervening flush or reposition unless an EOF has been reached.

fgetwc_unlocked() is functionally equivalent to fgetwc() with the exception that it is not thread-safe. This function can safely be used in a multithreaded application if and only if it is called while the invoking thread owns the (FILE*) object, as is the case after a successful call to either the flockfile() or ftrylockfile() function.

Returned value

If successful, fgetwc() returns the next wide character that corresponds to the multibyte character from the input stream pointed to by *stream*.

If the stream is at EOF, the EOF indicator for the stream is set and fgetwc() returns WEOF.

If a *read* error occurs, the error indicator for the stream is set and fgetwc() returns WEOF. If an *encoding* error occurs (an error converting the multibyte character into a wide character), the value of the macro EILSEQ (illegal sequence) is stored in errno and WEOF is returned.

The ferror() and feof() functions are used to distinguish between a read error and an EOF. Note that EOF is only reached when an attempt is made to read "past" the last byte of data. Reading up to and including the last byte of data does *not* turn on the EOF indicator.

Example

CELEBF19

```
/* CELEBF19 */
#include <stdio.h>
#include <stdlib.h>
#include <wchar.h>
#include <errno.h>

int main(void)
{
   FILE    *stream;
   wint_t   wc;

   if ((stream = fopen("myfile.dat", "r")) == NULL) {
      printf("Unable to open file\n");
      exit(1);
   }

   errno = 0;
   while ((wc = fgetwc(stream)) != WEOF)
      printf("wc=0x%X\n", wc);

   if (errno == EILSEQ) {
      printf("An invalid wide character was encountered.\n");
      exit(1);
   }

   fclose(stream);
}
```

Related information

- "stdio.h" on page 68
- "wchar.h" on page 85
- "fgetc() — Read a character" on page 526
- "fgetws() — Get a wide-character string"
- "fputwc() — Output a wide-character" on page 607

fgetws() — Get a wide-character string

Standards

Standards / Extensions	C or C++	Dependencies
ISO C Amendment XPG4 XPG4.2 C99 Single UNIX Specification, Version 3 Language Environment	both	

Format

```
#include <wchar.h>

wchar_t *fgetws(wchar_t * __restrict__wcs, int n, FILE * __restrict__stream);

#define _OPEN_SYS_UNLOCKED_EXT 1
#include <wchar.h>

wchar_t *fgetws_unlocked(wchar_t * __restrict__wcs,
                         int n, FILE * __restrict__stream);
```

General description

Reads at most one less than the number of the wide characters specified by *n*, from the stream pointed to by *stream*, into the array pointed to by *wcs*. No additional wide characters are read after a newline wide character (which is retained) or after the EOF. A NULL wide character is written immediately after the last wide character read into the array.

The fgetws() function advances the file position unless there is an error, when the file position is undefined.

The behavior of this wide-character function is affected by the LC_CTYPE category of the current locale. If you change the category, undefined results can occur.

Using non-wide-character functions with fgetws() results in undefined behavior. This happens because fgetws() processes a whole multibyte character and does not expect to be "within" such a character. In addition, fgetws() expects state information to be set already. Because functions like fgetc() and fputc() do not obey such rules, their results fail to meet the assumptions made by fgetws().

fgetws() has the same restriction as any read operation for a read immediately following a write or a write immediately following a read. Between a write and a subsequent read, there must be an intervening flush or reposition. Between a read and a subsequent write, there must also be an intervening flush or reposition unless an EOF has been reached.

fgetws_unlocked() is functionally equivalent to fgetws() with the exception that it is not thread-safe. This function can safely be used in a multithreaded application if and only if it is called while the invoking thread owns the (FILE*) object, as is the case after a successful call to either the flockfile() or ftrylockfile() function.

Returned value

If successful, fgetws() returns the new value of *wcs*.

If EOF is encountered and no wide characters have been read into the array, the contents of the array remain unchanged and fgetws() returns a NULL pointer.

If a read error or an encoding error occurs during the operation, the array contents are indeterminate and fgetws() returns a NULL pointer. An *encoding* error is one that occurs when a wide character is converted to a multibyte character. If it occurs, errno is set to EILSEQ and fgetws() returns NULL.

If *n* is less than or equal to 0, it indicates a domain error; errno is set to EDOM to indicate the cause of the failure.

When *n* equals 1, it indicates a valid result. It means that the string buffer has only room for the NULL terminator; nothing is physically read from the file. (Such an operation is still considered a read operation, so it cannot immediately follow a write operation unless there is an intervening flush or reposition operation first.)

If *n* is greater than 1, fgets() will only fail if an I/O error occurs or if EOF is reached, and no data is read from the file. To find out which error has occurred, use either the feof() or the ferror() function. If EOF is reached after data has already been read into the string buffer, fgetws() returns a pointer to the string buffer to indicate success. A subsequent call would result in NULL being returned because EOF would be reached without any data being read.

The ferror() and feof() functions are used to distinguish between a read error and an EOF. Note that EOF is only reached when an attempt is made to read “past” the last byte of data. Reading up to and including the last byte of data does *not* turn on the EOF indicator.

Example

CELEBF20

```
/* CELEBF20 */
#include <errno.h>
#include <stdio.h>
#include <stdlib.h>
#include <wchar.h>

int main(void)
{
   FILE    *stream;
   wchar_t  wcs[100];
   wchar_t  *ptr;

  if ((stream = fopen("myfile.dat", "r")) == NULL) {
      printf("Unable to open file\n");
      exit(1);
   }

   errno = 0;
   ptr = fgetws(wcs, 100, stream);

   if (ptr == NULL) {
      if (errno == EILSEQ) {
         printf("An invalid wide character was encountered.\n");
         exit(1);
      }
      else if (feof(stream))
         printf("end of file reached\n");
      else
         perror("read error");
   }

   printf("wcs=\"%ls\"\n", wcs);

   fclose(stream);
}
```

Related information

- “stdio.h” on page 68
- “wchar.h” on page 85
- “fgets() — Read a string from a stream” on page 530
- “fgetwc() — Get next wide character” on page 532

- “fputws() — Output a wide-character string” on page 609

fileno() — Get the file descriptor from an open stream

Standards

Standards / Extensions	C or C++	Dependencies
POSIX.1a XPG4 XPG4.2 Single UNIX Specification, Version 3 Language Environment	both	

Format

```
#define _POSIX_SOURCE
#include <stdio.h>

int fileno(const FILE *stream);

#define _OPEN_SYS_UNLOCKED_EXT 1
#include <stdio.h>

int fileno_unlocked(const FILE *stream);
```

General description

Returns the file descriptor number associated with a specified z/OS XL C/C++ I/O stream. The argument *stream* points to a FILE structure controlling a z/OS XL C/C++ I/O stream.

The unistd.h header file defines the following macros, which are constants that map to the file descriptors of the standard streams:

STDIN_FILENO
Standard input, stdin (value 0)

STDOUT_FILENO
Standard output, stdout (value 1)

STDERR_FILENO
Standard error, stderr (value 2)

Note that stdin, stdout, and stderr are macros, not constants.

fileno_unlocked() is functionally equivalent to fileno() with the exception that it is not thread-safe. This function can safely be used in a multithreaded application if and only if it is called while the invoking thread owns the (FILE*) object, as is the case after a successful call to either the flockfile() or ftrylockfile() function.

Returned value

If successful, fileno() returns the file descriptor number associated with an open HFS stream (that is, one opened with fopen() or freopen()). MVS data sets are not supported, so fileno() of an MVS data set returns -1.

If unsuccessful, fileno() returns -1 and sets errno to one of the following values:

Error Code
Description

EBADF
One of the following error conditions exists:
- *stream* points to a closed stream
- *stream* is an incorrect *stream* pointer
- *stream* points to a stream associated with an MVS data set.

Example

CELEBF21

```
/* CELEBF21

   This example illustrates one use of fileno().

 */
#define _POSIX_SOURCE
#include <errno.h>
#include <stdio.h>

main() {
  FILE *stream;
  char hfs_file[]="./hfs_file", mvs_ds[]="//mvs.ds";

  printf("fileno(stdin) = %d\n", fileno(stdin));

  if ((stream = fopen(hfs_file, "w")) == NULL)
    perror("fopen() error for HFS file");
  else {
    printf("fileno() of the HFS file is %d\n", fileno(stream));
    fclose(stream);
    remove(hfs_file);
  }

  if ((stream = fopen(mvs_ds, "w")) == NULL)
    perror("fopen() error for MVS data set");
  else {
    errno = 0;
    printf("fileno() returned %d for MVS data set,\n",fileno(stream));
    printf(" errno=%s\n", strerror(errno));
    fclose(stream);
    remove(mvs_ds);
  }
}
```

Output

```
fileno(stdin) = 0
fileno() of the HFS file is 3
fileno() returned -1 for the MVS data set,
errno=Bad file descriptor
```

Related information

- The “Standard Streams” chapter in *z/OS XL C/C++ Programming Guide*
- “stdio.h” on page 68
- “fdopen() — Associate a stream with an open file descriptor” on page 491
- “fopen() — Open a file” on page 565
- “freopen() — Redirect an open file” on page 620
- “open() — Open a file” on page 1152

finite() — Determine the infinity classification of a floating-point number

Standards

Standards / Extensions	C or C++	Dependencies
Language Environment	both	OS/390 V2R6

Format

```
#define _AIX_COMPATIBILITY
#include <math.h>

int finite(x)
double x;
```

General description

The finite() function determines the infinity classification of floating-point number *x*.

Note: This function works in both IEEE Binary Floating-Point and hexadecimal floating-point formats. See "IEEE binary floating-point" on page 94 for more information about IEEE Binary Floating-Point.

Returned value

finite() returns nonzero if the *x* parameter is a finite number, that is, if *x* is not +-, INF, NaNQ, or NaNS.

finite() does not return errors or set bits in the floating-point exception status, even if a parameter is a NaNS.

Special behavior for hex: finite() always returns 1 when it is called from HFP mode.

Related information

- IEEE Standard for Binary Floating-Point Arithmetic (ANSI/IEEE Standards 754-1985 and 854-1987)
- "math.h" on page 44

__flbf() — Determine if a stream is line buffered

Standards

Standards / Extensions	C or C++	Dependencies
Language Environment	both	None

Format

```
#include <stdio.h>
#include <stdio_ext.h>

int __flbf(FILE *stream);
```

General description

The __flbf() function determines if the specified stream is line buffered.

Returned value

The __flbf() function returns nonzero if the stream is line buffered. Otherwise, the __flbf() function returns 0. If an error has occurred, __flbf() returns 0 and sets errno to nonzero.

An application wishing to check for error situations should set errno to 0, then call __flbf(), and then check errno. If errno is nonzero, assume that an error has occurred.

Error Code
Description

EBADF
The stream specified by *stream* is not valid.

Example

CELEBF84

```
/* CELEBF84

   This example flushes all the line-buffered files.

*/

#include <stdio.h>
#include <stdio_ext.h>
#include <string.h>

#define BUF_SIZE 128

int main(void)
{
   char lbuf[BUF_SIZE]; /* line buffer */
   char fbuf[BUF_SIZE]; /* full buffer */
   char *tagstr = "This file was modified!";
   FILE *lfp;
   FILE *ffp;

   lfp = fopen("newlfile.dat", "a+");
   if(lfp == NULL){
      perror("Open file failed!\n");
      return -1;
   }
   if(setvbuf(lfp, lbuf, _IOLBF, sizeof(lbuf)) != 0){ /* set lbuf to line-buffered */
      perror("Format line-buffered failed!\n");
      fclose(lfp);
      return -1;
   }

   if (__flbf(lfp)) printf("newlfile.dat is line-buffered\n");
   else printf("newlfile.dat is not line-buffered\n");

   if(fwrite(lfp,strlen(tagstr), 1, lfp) != 1){ /* write tag string to line buffer*/
      perror("Write line buffer failed!\n");
      fclose(lfp);
      return -1;
   }
   printf("Write to the line buffered file succeeded\n");

   ffp = fopen("newffile.dat", "a+");
   if(ffp == NULL){
      perror("Open file failed!\n");
      fclose(lfp);
      return -1;
   }
   if(setvbuf(ffp, fbuf, _IOFBF, sizeof(fbuf)) != 0){ /* set fbuf to full-buffered */
```

```
      perror("Format full-buffered failed!\n");
      fclose(ffp);
      return -1;
   }

   if (__flbf(ffp)) printf("newffile.dat is line-buffered\n");
   else printf("newffile.dat is not line-buffered\n");

   if(fwrite(tagstr, strlen(tagstr), 1, ffp) != 1){ /* write tag string to full buffer */
      perror("Write full buffer failed!\n");
      fclose(lfp);
      fclose(ffp);
      return -1;
   }
   printf("Write to the full buffered file succeeded\n");
   _flushlbf(); /* flush line buffered files */
   printf("Only line buffered files are flushed...\n");
   fclose(lfp);
   fclose(ffp);
   return 0;
}
```

Output

```
newlfile.dat is line-buffered
Write to the line buffered file succeeded
newffile.dat is not line-buffered
Write to the full buffered file succeeded
Only line buffered files are flushed...
```

Related information

- "stdio.h" on page 68
- "stdio_ext.h" on page 70
- "setbuf() — Control buffering" on page 1522
- "setvbuf() — Control buffering" on page 1596

fldata() — Retrieve file information

Standards

Standards / Extensions	C or C++	Dependencies
Language Environment	both	

Format

```
#include <stdio.h>

int fldata(FILE *file, char *filename, fldata_t *info);

#define _OPEN_SYS_UNLOCKED_EXT 1
#include <stdio.h>

int fldata_unlocked(FILE *file, char *filename, fldata_t *info);
```

General description

Retrieves information about an open stream pointed to by *file*. It returns the file name in *filename* and other information in the structure *info*. The file name returned in *filename* is the name specified in fopen() or freopen(). If the file is opened with a *ddname* (for example, `fopen("DD:A","w")`)), then the *filename* field will contain the *ddname* used to open the file, prefixed with `dd:`. If the file is a DASD data set or a memory file, the field `__dsname` contains the dsname. If the file is an HFS file, the field __dsname contains the path name. For all other files, it is NULL.

After a failure, the contents of the information structure are indeterminate.

To avoid infringing on the user's name space, this nonstandard function has two names. One name is prefixed with two underscore characters, and one name is not. The name without the prefix underscore characters is exposed only when you use LANGLVL(EXTENDED).

To use this function, you must either invoke the function using its external entry point name (that is, the name that begins with two underscore characters), or compile with LANGLVL(EXTENDED). When you use LANGLVL(EXTENDED) any relevant information in the header is also exposed.

For full details about *filename* considerations, see one of the "Opening Files" section s in *z/OS XL C/C++ Programming Guide*.

If *fldata* is the first reference to a standard stream, a call to the fldata() function opens the stream.

See Table 23.

Note:

A file name of NULL indicates that no file name will be returned.

FILENAME_MAX is recommended for the size of the file name buffer.

The table "Elements Returned in fldata_t Data Structure" (below) describes the fields in the fldata_t data structure. For further details, see *z/OS XL C/C++ Programming Guide*.

fldata_unlocked() is functionally equivalent to fldata() with the exception that it is not thread-safe. This function can safely be used in a multithreaded application if and only if it is called while the invoking thread owns the (FILE*) object, as is the case after a successful call to either the flockfile() or ftrylockfile() function.

Special behavior for POSIX C: Under z/OS UNIX services, if there had been an exec to an application that invokes fldata(), the standard streams are opened at the time of the exec. Thus fldata() does not attempt to open them again.

Returned value

If successful, fldata() returns 0.

If unsuccessful, fldata() returns nonzero.

Table 23. Elements Returned in fldata_t Data Structure

Element	Data Type	General Description
__recfmF:1	unsigned int	Indicates whether it has fixed-length records.
__recfmV:1	unsigned int	Indicates whether it has variable-length records.
__recfmU:1	unsigned int	Indicates whether it has undefined-length records.
__recfmS:1	unsigned int	Indicates whether it has either standard (if fixed-length) or spanned (if variable-length) records.
__recfmBlk:1	unsigned int	Indicates whether it has blocked records.
__recfmASA:1	unsigned int	Indicates whether it has ASA print-control characters.

Table 23. Elements Returned in fldata_t Data Structure (continued)

Element	Data Type	General Description
__recfmM:1	unsigned int	Indicates whether it has machine print-control codes.
__dsorgPO:1	unsigned int	Indicates whether it is a partitioned data set.
__dsorgPDSmem:1	unsigned int	Indicates whether a file is a member.
__dsorgPDSdir:1	unsigned int	Indicates whether a file is a PDS or PDSE directory.
__dsorgPS:1	unsigned int	Indicates whether it is a sequential data set.
__dsorgConcat:1	unsigned int	Indicates whether it is a sequentially concatenated file.
__dsorgMem:1	unsigned int	Indicates whether it is a memory file.
__dsorgHiper:1	unsigned int	Indicates whether it is a memory file in hiperspace.
__dsorgTemp:1	unsigned int	Indicates whether it is a temporary file created by tmpfile().
__dsorgVSAM:1	unsigned int	Indicates whether it is a VSAM file.
__dsorgHFS:	unsigned int	Indicates whether it is an HFS file.
__openmode:2	unsigned int	Values are __TEXT, __BINARY, __RECORD, __BLOCKED.
__modeflag:4	unsigned int	Values are __APPEND, __READ, __UPDATE, __WRITE. These macros can be added together to determine the value; for example, a file opened with mode a+ will have the value __APPEND + __UPDATE.
__dsorgPDSE:1	unsigned int	Indicates whether a file is a PDSE.
__vsamRLS:3	unsigned int	Returned values are __NORLS, __RLS, __TVS.
__recfmB:1	unsigned int	Indicates whether it was allocated with blocked records
__reserve2:3	unsigned int	Reserved bits.
__device	char	Returned values are __DISK, __TERMINAL, __PRINTER, __TAPE, __TDQ, __DUMMY, __OTHER, __MEMORY, __MSGFILE, __HFS, __HIPERSPACE, __MSGRTN.
__blksize	unsigned long	Total block size of the file, including all control information needed in the block.
__maxreclen	unsigned long	Maximum length of the data in the record, including ASA control characters, if present.
__vsamtype	unsigned short	Returned values are __NOTVSAM, __ESDS, __KSDS, __RRDS, __ESDS_PATH, __KSDS_PATH. **Note:** Valid only if __dsorgVSAM is set.
__vsamkeylen	unsigned long	Length of VSAM key (if any). **Note:** Valid only if __dsorgVSAM is set.
__vsamRKP	unsigned long	Key position. **Note:** Valid only if __dsorgVSAM is set.

Table 23. Elements Returned in fldata_t Data Structure (continued)

Element	Data Type	General Description
__access_method	uint8_t	Identifies the access method used for the data set. Values include: __AM_UNSPEC __AM_BSAM __AM_QSAM **Note:** Valid only if __dsorgPS or __dsorgPO is set.
__noseek_to_seek	uint8_t	Identifies the reason noseek was changed to seek. Values include: __AM_BSAM_NOSWITCH __AM_BSAM_UPDATE __AM_BSAM_BSAMWRITE __AM_BSAM_FBS_APPEND __AM_BSAM_LRECLX __AM_BSAM_PARTITIONED_DIRECTORY __AM_BSAM_PARTITIONED_INDIRECT **Note:** Valid only if __dsorgPS or __dsorgPO is set.
__dsname	char *	The contents of this field is determined by the following: • If the file is a DASD data set, memory file, or an HFS file, then __dsname contains the real file name of file opened by ddname • If you open by ddname, and the ddname is a concatenation of PDS or PDSE data sets, then __dsname contains the data set name of the first PDS or PDSE. This is because your are only opening the directory of the first PDS or PDSE. • If you open by ddname(member) and the ddname is a concatenation of PDS or PDSE data sets, then __dsname contains the data set name of the first PDS or PDSE containing the member. Otherwise this field is NULL. The char *__dsname field is allocated internally by the library functions and must be saved before the next call to the fldata() function.
__reserve4	unsigned long	Reserved.

Note: The numeric values for the macro names can be found in stdio.h. The meaning of the __noseek_to_seek values are described in topic about using the __amrc structure in *z/OS XL C/C++ Programming Guide*.

Example

```
#include <stdio.h>

int main(void)
{
    FILE *stream;
    char filename[100];
    fldata_t fileinfo;
    int rc;

    stream = fopen("myfile.dat","rb+");
⋮
    rc = fldata(stream, filename, &fileinfo);
    if (rc != 0)
       printf("fldata failed\n");
    else
       printf("filename is %s\n",filename);
}
```

Related information

- "stdio.h" on page 68
- "fopen() — Open a file" on page 565
- "freopen() — Redirect an open file" on page 620

flocate() — Locate a VSAM record

Standards

Standards / Extensions	C or C++	Dependencies
Language Environment	both	

Format

```
#include <stdio.h>

int flocate(FILE *stream, const void *key, size_t key_len, int options);

#define _OPEN_SYS_UNLOCKED_EXT 1
#include <stdio.h>

int flocate_unlocked(FILE *stream, const void *key, size_t key_len, int options);
```

General description

Moves the VSAM file position indicator associated with the stream pointed to by *stream*, according to the rest of the arguments specified.

To avoid infringing on the user's name space, this nonstandard function has two names. One name is prefixed with two underscore characters, and one name is not. The name without the prefix underscore characters is exposed only when you use LANGLVL(EXTENDED).

To use this function, you must either invoke the function using its external entry point name (that is, the name that begins with two underscore characters), or compile with LANGLVL(EXTENDED). When you use LANGLVL(EXTENDED) any relevant information in the header is also exposed.

key points to a key used for positioning.

key_len specifies the length of the search key. The *key_len* argument value must always be nonzero, except for `__KEY_FIRST` and `__KEY_LAST`.

KSDS, KSDS PATH, and ESDS PATH

The *key* can point to a field of any storage type except `register`. Typically it points to a character string whose length is *key_len*. The *key_len* must be less than or equal to the key length of the data set. If *key_len* is the same as the file's key length, a full key search is automatically used; otherwise, a generic search is used. A generic key search is one in which the search key is a leading portion of the key field. The record positioned to is the first of the records having the same generic key.

ESDS The *key* points to a relative byte address that may be stored as 4 or 8 byte value. *key_len* is either 4 or 8.

RRDS The *key* points to a relative record number stored as an `unsigned long int`. *key_len* is either 4 or 8.

options specifies the position options described in Table 24.

Table 24. Position Options Parameter for flocate()

__KEY_FIRST	Positions to the first record in the file. Subsequent reads are in the forward direction. *key* and *key_len* are ignored.
__KEY_LAST	Positions to the last record in the file. Subsequent reads are in backward order. *key* and *key_len* are ignored. Only applies to VSAM files opened in record mode.
__KEY_EQ	Positions to the first record with the specified key. Subsequent reads are in the forward direction.
__KEY_EQ_BWD	Positions to the first record with the specified key. Subsequent reads are in backward order. Using this option requires a full key search. Key_len must be equal to the key length as defined for the data set. Only applies to VSAM files opened in record mode.
__KEY_GE	Positions to the first record with a key greater than or equal to the specified key.
__RBA_EQ	Positions to the record with the specified RBA. Subsequent reads are in the forward direction. You cannot use __RBA_EQ with an alternative index path. Using this option with RRDS is *not* recommended. The underlying VSAM utilities do not support seeking to an RBA in an RRDS file. The flocate() function attempts to convert the RBA to a Relative Record Number by dividing the value by the LRECL of the file and using the equivalent __KEY_EQ. Using this option with KSDS is *not* recommended because the RBA of a given record may change over time, because of inserts, deletions, or updates of other records.
__RBA_EQ_BWD	Positions to the record with the specified RBA. Subsequent reads are in backward order. You cannot use __RBA_EQ_BWD with an alternative index path. Using this option with RRDS is not recommended. The underlying VSAM utilities do not support seeking to an RBA in an RRDS file. The flocate() function attempts to convert the RBA to a Relative Record Number by dividing the value by the LRECL of the file and using the equivalent __KEY_EQ_BWD. Using this option with KSDS is *not* recommended because the RBA of a given record may change over time, because of inserts, deletions, or updates of other records. Only applies to VSAM files opened in record mode.

Table 24. Position Options Parameter for flocate() (continued)

_KEY_EQ_BWD	Positions to the first record with the specified key. Subsequent reads are in backward order. Using this option requires a full key search. key_len must be set equal to the key length as defined for the data set. Only applies to VSAM files opened in record mode.

Notes:

1. When you are trying to use flocate() in a path to a nonunique key, the resulting position will be at the first physical record of the duplicate key set.
2. flocate() releases all record locking.
3. Writes to VSAM data sets are not affected by preceding calls to flocate().
4. If a record was not found, you must successfully relocate to another position before reading or writing (using the flocate() function). The exception to this is that a write that follows a failed flocate() will succeed if the file was opened for initial loading, but no records have been written to it yet.

flocate_unlocked() is functionally equivalent to flocate() with the exception that it is not thread-safe. This function can safely be used in a multithreaded application if and only if it is called while the invoking thread owns the (FILE*) object, as is the case after a successful call to either the flockfile() or ftrylockfile() function.

Considerations for VSAM extended addressability data sets: flocate() accepts key lengths of 4 or 8 when relative byte address (RBA) values are used for positioning. A key length of 8 is required only when the working with a VSAM extended addressability data set, because when the address grows past the 4GB boundary, the key needs to be large enough to hold the value.

When using the value 4GB-1 as the key to flocate(), the key length must be 8 and the data type used must be 8 bytes in size (for example, X' 00000000FFFFFFFF') . If the key length is 4, flocate() treats the key as -1(EOF).

Returned value

If successful, flocate() returns 0.

If a record was not found or the position is beyond the EOF, flocate() returns EOF.

Example

```
#include <stdio.h>

int main(void)
{
   FILE *stream;
   int   vsam_rc;
   char *key = "RECORD 27";

   stream = fopen("DD:MYCLUS", "rb+,type=record");
   vsam_rc = flocate(stream, key, 9, __KEY_EQ);
⋮
}
```

Related information

- "Performing VSAM I/O Operations" in *z/OS XL C/C++ Programming Guide*
- "stdio.h" on page 68
- "fdelrec() — Delete a VSAM record" on page 486
- "fgetpos() — Get file position" on page 528
- "fseek() — Change file position" on page 639
- "fsetpos() — Set file position" on page 649
- "ftell() — Get current file position" on page 657
- "fupdate() — Update a VSAM record" on page 669

flockfile()— stdio locking

Standards

Standards / Extensions	C or C++	Dependencies
Single UNIX Specification, Version 3	both	z/OS V1R8

Format

```
#define _UNIX03_SOURCE
#include <stdio.h>

void flockfile(FILE *file);
```

General description

This function provides explicit application-level locking of stdio (FILE*) objects. The flockfile() family of the functions can be used by a thread to delineate a sequence of I/O statements that are executed as a unit.

The flockfile() function acquires ownership of a (FILE*) object for the thread, waiting if necessary, and increases the internal lock count. If the thread has previously been granted ownership, the internal lock count is increased.

The internal lock count allows matching calls to flockfile() (or successful calls to ftrylockfile()) and funlockfile() to be nested.

z/OS Consideration: The flockfile() family of functions acts upon FILE * objects. It is possible to have the same physical file represented by multiple FILE * objects that are not recognized as being equivalent. For example, fopen() opens a file and open() opens the same file, and then fdopen() creates a FILE * object. In this case, locking the first FILE * does not prevent the second FILE * from also being locked and used.

Returned value

None.

Notes:

1. Because the flockfile() function returns void, no error information can be returned.
2. It is the application's responsibility to prevent deadlock (or looping). For example, deadlock (or looping) may occur if a (FILE *) object is closed, or a thread is terminated, before relinquishing all locked (FILE *) objects.

Related information

- “ftrylockfile() — stdio locking” on page 665
- “funlockfile() — stdio unlocking” on page 668

floor(), floorf(), floorl() — Round down to integral value

Standards

Standards / Extensions	C or C++	Dependencies
ISO C POSIX.1 XPG4 XPG4.2 ISO/ANSI C++ C99 Single UNIX Specification, Version 3 C++ TR1 C99	both	

Format

```
#include <math.h>

double floor(double x);
float floor(float x);                  /* C++ only */
long double floor(long double x);      /* C++ only */
float floorf(float x);
long double floorl(long double x);
```

General description

Calculates the largest integer that is less than or equal to *x*.

Note: These functions work in both IEEE Binary Floating-Point and hexadecimal floating-point formats. See “IEEE binary floating-point” on page 94 for more information about IEEE Binary Floating-Point.

Returned value

Returns the calculated integral value expressed as a double, float, or long double value. The result cannot have a range error.

Example

CELEBF24

```
/* CELEBF24

   This example assigns y the value of the largest integer that is less
   than or equal to 2.8, and it assigns z the value of the largest integer
   that is less than or equal to -2.8.

 */
#include <math.h>
#include <stdio.h>

int main(void)
{
   double y, z;

   y = floor(2.8);
   z = floor(-2.8);
   printf("floor(  2.8 ) = %f\n", y);
   printf("floor( -2.8 ) = %f\n", z);
}
```

Output

```
floor(  2.8 ) = 2.000000
floor( -2.8 ) = -3.000000
```

Related information

- "math.h" on page 44
- "ceil(), ceilf(), ceill() — Round up to integral value" on page 249
- "fmod(), fmodf(), fmodl() — Calculate floating-point remainder" on page 559

floord32(), floord64(), floord128() — Round down to integral value

Standards

Standards / Extensions	C or C++	Dependencies
C/C++ DFP	both	z/OS V1.8

Format

```
#define __STDC_WANT_DEC_FP__
#include <math.h>

_Decimal32  floord32(_Decimal32 x);
_Decimal64  floord64(_Decimal64 x);
_Decimal128 floord128(_Decimal128 x);
_Decimal32  floor(_Decimal32 x);      /* C++ only */
_Decimal64  floor(_Decimal64 x);      /* C++ only */
_Decimal128 floor(_Decimal128 x);     /* C++ only */
```

General description

Calculates the largest integer that is less than or equal to *x*.

Notes:

1. To use IEEE decimal floating-point, the hardware must have the Decimal Floating-Point Facility installed.
2. These functions work in IEEE decimal floating-point format. See "IEEE Decimal Floating-Point" for more information.

Returned value

Returns the calculated integral value expressed as a _Decimal32, _Decimal64, or _Decimal128 value. The result cannot have a range error.

Example

```
/* CELEBF78

   This example illustrates the floord64() function.

   This example assigns y the value of the largest integer that is less
   than or equal to 2.8, and it assigns z the value of the largest
   integer that is less than or equal to -2.8.

*/
```

```
#define  __STDC_WANT_DEC_FP__
#include <math.h>
#include <stdio.h>

int main(void)
{
   _Decimal64 y, z;

   y = floord64(+2.8DD);
   z = floord64(-2.8DD);

   printf("floord64(+2.8) = %+Df\n", y);
   printf("floord64(-2.8) = %+Df\n", z);
}
```

Related information

- "math.h" on page 44
- "ceild32(), ceild64(), ceild128() — Round up to integral value" on page 250
- "floor(), floorf(), floorl() — Round down to integral value" on page 548

_flushlbf() — Flush all open line-buffered files

Standards

Standards / Extensions	C or C++	Dependencies
Language Environment	both	None

Format

```
#include <stdio.h>
#include <stdio_ext.h>

void _flushlbf(void);

#define _OPEN_SYS_UNLOCKEKD 1
#include <stdio.h>
#include <stdio_ext.h>

void _flushlbf_unlocked(void);
```

General description

The _flushlbf() function flushes all open line-buffered streams.

The _flushlbf() function is affected by the ungetc() and ungetwc() functions. Calling this function causes _flushlbf() to back up the file position when characters are pushed back. For details, see "ungetc() — Push character onto input stream" on page 1958 and "ungetwc() — Push a wide character onto a stream" on page 1961. If needed, the _EDC_COMPAT environment variable can be set at open time such that _flushlbf() discards any pushed-back characters and leaves the file position where it was when the first ungetc() or ungetwc() function call was issued.

If the _flushlbf() function is used after the ungetwc() function pushed a wide char on a text stream, the position will be backed up by one wide character from the position of the file when the ungetwc() function was issued. For a wide-oriented binary stream, the position will be backed up based on the number of bytes that are used to represent the wide char in the ungetc buffer. For this reason, attempting to use ungetwc() on a character when the destination is a binary wide-oriented stream that was never read in the first place results in undefined behavior for _flushlbf(). Note that the _EDC_COMPAT environment variable also changes the behavior of _flushlbf() after ungetwc(), and will cause any wide char pushed-back to be discarded and the position left at the point where the ungetwc() was issued. For details about the _EDC_COMPAT environment variable, see *Environment Variables* in *z/OS XL C/C++ Programming Guide*.

If _flushlbf() fails, the position is left at the point in the file where the first ungetc() or ungetwc() function call was issued. All pushed-back characters are discarded.

Note:

1. The system automatically flushes buffers when you close the stream or when a program ends normally without closing the stream.
2. The _flushlbf() function has no effect on line-buffered text files, because z/OS XL C/C++ writes all records to the system as they are completed. All incomplete new records remain in the buffer.

The buffering mode and the file type can have an effect on when output data is flushed. For more information, see *Buffering of C Streams* in *z/OS XL C/C++ Programming Guide*.

All streams remain open after the _flushlbf() call. Because a read operation cannot immediately follow or precede a write operation, the _flushlbf() function can be used to allow exchange between these two modes. The _flushlbf() function can also be used to refresh the buffer when working with a reader and a simultaneous writer or updater.

The _flushlbf_unlocked() function is equivalent to the _flushlbf() function with the exception that it is not thread-safe. This function can be safely used in a multithreaded application whether the user has locked all open line-buffered files or not.

When flushing all open line-buffered files, a failure to flush any of the files will leave it unchanged. However, flushing will continue on any other open line-buffered files that can be flushed successfully.

Returned value

The _flushlbf() function returns no values.

Example

CELEBF84

```
/* CELEBF84

   This example flushes all the line-buffered files.

*/

#include <stdio.h>
#include <stdio_ext.h>
#include <string.h>

#define BUF_SIZE 128

int main(void)
{
   char lbuf[BUF_SIZE]; /* line buffer */
   char fbuf[BUF_SIZE]; /* full buffer */
   char *tagstr = "This file was modified!";
   FILE *lfp;
   FILE *ffp;
```

```
    lfp = fopen("newlfile.dat", "a+");
    if(lfp == NULL){
       perror("Open file failed!\n");
       return -1;
    }
    if(setvbuf(lfp, lbuf, _IOLBF, sizeof(lbuf)) != 0){ /* set lbuf to line-buffered */
       perror("Format line-buffered failed!\n");
       fclose(lfp);
       return -1;
    }

    if (__flbf(lfp)) printf("newlfile.dat is line-buffered\n");
    else printf("newlfile.dat is not line-buffered\n");

    if(fwrite(lfp,strlen(tagstr), 1, lfp) != 1){ /* write tag string to line buffer*/
       perror("Write line buffer failed!\n");
       fclose(lfp);
       return -1;
    }
    printf("Write to the line buffered file succeeded\n");

    ffp = fopen("newffile.dat", "a+");
    if(ffp == NULL){
       perror("Open file failed!\n");
       fclose(lfp);
       return -1;
    }
    if(setvbuf(ffp, fbuf, _IOFBF, sizeof(fbuf)) != 0){ /* set fbuf to full-buffered */
       perror("Format full-buffered failed!\n");
       fclose(ffp);
       return -1;
    }

    if (__flbf(ffp)) printf("newffile.dat is line-buffered\n");
    else printf("newffile.dat is not line-buffered\n");

    if(fwrite(tagstr, strlen(tagstr), 1, ffp) != 1){ /* write tag string to full buffer */
       perror("Write full buffer failed!\n");
       fclose(lfp);
       fclose(ffp);
       return -1;
    }
    printf("Write to the full buffered file succeeded\n");
    _flushlbf(); /* flush line buffered files */
    printf("Only line buffered files are flushed...\n");
    fclose(lfp);
    fclose(ffp);
    return 0;
}
```

Output

```
newlfile.dat is line-buffered
Write to the line buffered file succeeded
newffile.dat is not line-buffered
Write to the full buffered file succeeded
Only line buffered files are flushed...
```

Related information

- "stdio.h" on page 68
- "stdio_ext.h" on page 70
- "setbuf() — Control buffering" on page 1522
- "setvbuf() — Control buffering" on page 1596
- "ungetc() — Push character onto input stream" on page 1958
- "ungetwc() — Push a wide character onto a stream" on page 1961

fma(), fmaf(), fmal() — Multiply then add

Standards

Standards / Extensions	C or C++	Dependencies
C99 Single UNIX Specification, Version 3 C++ TR1 C99	both	z/OS V1R7

Format

```
#define _ISOC99_SOURCE
#include <math.h>

double fma(double x, double y, double z);
float fmaf(float x, float y, float z);
long double fmal(long double x, long double y, long double z);
```

C++ TR1 C99:

```
#define _TR1_C99
#include <math.h>

float fma(float x, float y, float z);
long double fma(long double x, long double y, long double z);
```

General description

The fma() family of functions compute (*x* * *y*) + *z*, rounded as one ternary operation: they compute the value to infinite precision and round once to the resulting format according to the rounding mode characterized by the value of **FLT_ROUNDS**.

Note: The following table shows the viable formats for these functions. See "IEEE binary floating-point" on page 94 for more information about IEEE Binary Floating-Point.

Function	Hex	IEEE
fma	X	X
fmaf	X	X
fmal	X	X

Restriction: The fmaf() function does not support the _FP_MODE_VARIABLE feature test macro.

Returned value

If successful, they return the rounded value of (*x* * *y*) + *z* as one ternary operation.

Related information

- "math.h" on page 44

fmad32(), fmad64(), fmad128() — Multiply then add

Standards

Standards / Extensions	C or C++	Dependencies
C/C++ DFP	both	z/OS V1.11

Format

```
#define __STDC_WANT_DEC_FP__
#include <math.h>

_Decimal32  fmad32(_Decimal32 x, _Decimal32 y, _Decimal32 z);
_Decimal64  fmad64(_Decimal64 x, _Decimal64 y, _Decimal64 z);
_Decimal128 fmad128(_Decimal128 x, _Decimal128 y, _Decimal128 z);

_Decimal32  fma(_Decimal32 x, _Decimal32 y, _Decimal32 z);     /* C++ only */
_Decimal64  fma(_Decimal64 x, _Decimal64 y, _Decimal64 z);     /* C++ only */
_Decimal128 fma(_Decimal128 x, _Decimal128 y, _Decimal128 z);    /* C++ only */
```

General description

The fma() family of functions compute (*x* * *y*) + *z* rounded as one ternary operation: they compute the value to infinite precision and round once to the resulting format according to the current rounding mode.

Notes:

1. These functions work in IEEE decimal floating-point format. See “IEEE decimal floating-point” on page 95IEEE Decimal Floating-Point for more information.
2. To use IEEE decimal floating-point, the hardware must have the Decimal Floating-Point Facility installed.

Returned value

If successful, they return the rounded value of (*x* * *y*) + *z* as one ternary operation.

Example

```
/* CELEBF82

   This example illustrates the fmad64() function.

*/

#define  __STDC_WANT_DEC_FP__
#include <math.h>
#include <stdio.h>

void main(void)
{
   _Decimal64 w, x, y, z;

   x = 2.5DD;
   y = 6.7DD;
   z = 1.0DD;
   w = fmad64(x,y,z);

   printf("fmad64( %Df, %Df, %Df ) = %Df\n", x, y, z, w);
}
```

Related information

- “math.h” on page 44

fmax(), fmaxf(), fmaxl() — Calculate the maximum numeric value

Standards

Standards / Extensions	C or C++	Dependencies
C99 Single UNIX Specification, Version 3 C++ TR1 C99	both	 z/OS V1R7

Format

```
#define _ISOC99_SOURCE
#include <math.h>

double fmax(double x, double y);
float fmaxf(float x, float y);
long double fmaxl(long double x, long double y);
```

C++ TR1 C99:

```
#define _TR1_C99
#include <math.h>

float fmax(float x, float y);
long double fmax(long double x, long double y);
```

General description

The fmax() family of functions determine the maximum numeric value of their arguments. NaN arguments are treated as missing data. If one argument is a NaN and the other numeric, then the numeric value will be chosen.

Note: The following table shows the viable formats for these functions. See “IEEE binary floating-point” on page 94 for more information about IEEE Binary Floating-Point.

Function	Hex	IEEE
fmax	X	X
fmaxf	X	X
fmaxl	X	X

Restriction: The fmaxf() function does not support the _FP_MODE_VARIABLE feature test macro.

Returned value

If successful, they return the maximum numeric value of their arguments.

Related information

- “math.h” on page 44
- “fdim(), fdimf(), fdiml() — Calculate the positive difference” on page 489
- “fmin(), fminf(), fminl() — Calculate the minimum numeric value” on page 557

fmaxd32(), fmaxd64(), fmaxd128() — Calculate the maximum numeric value

Standards

Standards / Extensions	C or C++	Dependencies
C/C++ DFP	both	z/OS V1.8

Format

```
#define __STDC_WANT_DEC_FP__
#include <math.h>

_Decimal32  fmaxd32(_Decimal32 x, _Decimal32 y);
_Decimal64  fmaxd64(_Decimal64 x, _Decimal64 y);
_Decimal128 fmaxd128(_Decimal128 x, _Decimal128 y);

_Decimal32  fmax(_Decimal32 x, _Decimal32 y);      /* C++ only */
_Decimal64  fmax(_Decimal64 x, _Decimal64 y);      /* C++ only */
_Decimal128 fmax(_Decimal128 x, _Decimal128 y);    /* C++ only */
```

General description

The fmax() family of functions determine the maximum numeric value of their arguments. NaN arguments are treated as missing data. If one argument is a NaN and the other numeric, then the numeric value will be chosen.

Notes:

1. To use IEEE decimal floating-point, the hardware must have the Decimal Floating-Point Facility installed.
2. These functions work in IEEE decimal floating-point format. See "IEEE Decimal Floating-Point" for more information.

Returned value

If successful, they return the maximum numeric value of their arguments.

Example

```
/* CELEBF79

   This example illustrates the fmaxd128() function.

*/

#define  __STDC_WANT_DEC_FP__
#include <math.h>
#include <stdio.h>

int main(void)
{
   _Decimal128 x = 3.5DL, y = 4.0DL, z;

   z = fmaxd128(x, y);

   printf("The maximum number between %DDf and %DDf is %DDf\n", x, y, z);
}
```

Related information

- "math.h" on page 44

- "fdimd32(), fdimd64(), fdimd128() — Calculate the positive difference" on page 490
- "fmax(), fmaxf(), fmaxl() — Calculate the maximum numeric value" on page 555
- "fmind32(), fmind64(), fmind128() — Calculate the minimum numeric value" on page 558

fmin(), fminf(), fminl() — Calculate the minimum numeric value

Standards

Standards / Extensions	C or C++	Dependencies
C99 Single UNIX Specification, Version 3 C++ TR1 C99	both	z/OS V1R7

Format

```
#define _ISOC99_SOURCE
#include <math.h>

double fmin(double x, double y);
float fminf(float x, float y);
long double fminl(long double x, long double y);
```

C++ TR1 C99:

```
#define _TR1_C99
#include <math.h>

float fmin(float x, float y);
long double fmin(long double x, long double y);
```

General description

The fmin() family of functions determine the minimum numeric value of their arguments. NaN arguments are treated as missing data. If one argument is a NaN and the other numeric, then the numeric value will be chosen.

Note: The following table shows the viable formats for these functions. See "IEEE binary floating-point" on page 94 for more information about IEEE Binary Floating-Point.

Function	Hex	IEEE
fmin	X	X
fminf	X	X
fminl	X	X

Restriction: The fminf() function does not support the _FP_MODE_VARIABLE feature test macro.

Returned value

If successful, they return the minimum numeric value of their arguments.

Related information

- "math.h" on page 44
- "fdim(), fdimf(), fdiml() — Calculate the positive difference" on page 489
- "fmax(), fmaxf(), fmaxl() — Calculate the maximum numeric value" on page 555

fmind32(), fmind64(), fmind128() — Calculate the minimum numeric value

Standards

Standards / Extensions	C or C++	Dependencies
C/C++ DFP	both	z/OS V1.8

Format

```
#define __STDC_WANT_DEC_FP__
#include <math.h>

_Decimal32  fmind32(_Decimal32 x, _Decimal32 y);
_Decimal64  fmind64(_Decimal64 x, _Decimal64 y);
_Decimal128 fmind128(_Decimal128 x, _Decimal128 y);

_Decimal32  fmin(_Decimal32 x, _Decimal32 y);    /* C++ only */
_Decimal64  fmin(_Decimal64 x, _Decimal64 y);    /* C++ only */
_Decimal128 fmin(_Decimal128 x, _Decimal128 y);  /* C++ only */
```

General description

The fmin() family of functions determine the minimum numeric value of their arguments. NaN arguments are treated as missing data. If one argument is a NaN and the other numeric, then the numeric value will be chosen.

Notes:

1. To use IEEE decimal floating-point, the hardware must have the Decimal Floating-Point Facility installed.
2. These functions work in IEEE decimal floating-point format. See "IEEE Decimal Floating-Point" for more information.

Returned value

If successful, they return the minimum numeric value of their arguments.

Example

```
/* CELEBF70
   This example illustrates the fmind32() function
*/
```

Related information

- “math.h” on page 44
- “fdimd32(), fdimd64(), fdimd128() — Calculate the positive difference” on page 490
- “fmaxd32(), fmaxd64(), fmaxd128() — Calculate the maximum numeric value” on page 556
- “fmin(), fminf(), fminl() — Calculate the minimum numeric value” on page 557

fmod(), fmodf(), fmodl() — Calculate floating-point remainder

Standards

Standards / Extensions	C or C++	Dependencies
ISO C POSIX.1 XPG4 XPG4.2 ISO/ANSI C++ C99 Single UNIX Specification, Version 3 C++ TR1 C99	both	

Format

```
#include <math.h>

double fmod(double x, double y);
float fmod(float x, float y);                          /* C++ only */
long double fmod(long double x, long double y);  /* C++ only */
float fmodf(float x, float y);
long double fmodl(long double x, long double y);
```

General description

Calculates the floating-point remainder of *x*/*y*. The absolute value of the result is always less than the absolute value of *y*. The result will have the same sign as *x*.

Note: These functions work in both IEEE Binary Floating-Point and hexadecimal floating-point formats. See "IEEE binary floating-point" on page 94 for more information about IEEE Binary Floating-Point.

Restriction: The fmodf() function does not support the _FP_MODE_VARIABLE feature test macro.

Returned value

If *y* is 0, or the result would overflow, then the function returns 0. Errno remains unchanged.

Special behavior for IEEE

If successful, the function returns the floating-point remainder of *x*/*y*.

If *y* is 0, the function sets errno to EDOM and returns NaNQ. No other errors will occur.

Example

CELEBF25

```
/* CELEBF25

   This example computes z as the remainder of x/y; here x/y is -3 with a
   remainder of -1.

 */
#include <math.h>
```

```
#include <stdio.h>

int main(void)
{
   double x, y, z;

   x = -10.0;
   y = 3.0;
   z = fmod(x,y);      /* z = -1.0 */

   printf("fmod( %f, %f) = %lf\n", x, y, z);
}
```

Output

```
fmod( -10.000000, 3.000000) = -1.000000
```

Related information

- “math.h” on page 44
- “modf(), modff(), modfl() — Extract fractional and integral parts of floating-point value” on page 1086

fmodd32(), fmodd64(), fmodd128() — Calculate floating-point remainder

Standards

Standards / Extensions	C or C++	Dependencies
C/C++ DFP	both	z/OS V1.11

Format

```
#define __STDC_WANT_DEC_FP__
#include <math.h>

_Decimal32  fmodd32(_Decimal32 x, _Decimal32 y);
_Decimal64  fmodd64(_Decimal64 x, _Decimal64 y);
_Decimal128 fmodd128(_Decimal128 x, _Decimal128 y);

_Decimal32  fmod(_Decimal32 x, _Decimal32 y);     /* C++ only */
_Decimal64  fmod(_Decimal64 x, _Decimal64 y);     /* C++ only */
_Decimal128 fmod(_Decimal128 x, _Decimal128 y);    /* C++ only */
```

General description

Calculates the floating-point remainder of *x*/*y*. The absolute value of the result is always less than the absolute value of *y*. The result will have the same sign as *x*.

Notes:

1. These functions work in IEEE decimal floating-point format. See “IEEE decimal floating-point” on page 95IEEE Decimal Floating-Point for more information.
2. To use IEEE decimal floating-point, the hardware must have the Decimal Floating-Point Facility installed.

Returned value

If successful, the function returns the floating-point remainder of *x/y*.

If *y* is 0, the function sets errno to EDOM and returns NaNQ. No other errors will occur.

Example

```
/* CELEBF83

   This example illustrates the fmodd32() function.

*/

#define  __STDC_WANT_DEC_FP__
#include <math.h>
#include <stdio.h>

void main(void)
{
   _Decimal32 x, y, z;

   x = -10.0DF;
   y = 3.0DF;
   z = fmodd32(x, y);

   printf("fmodd32( %Hf, %Hf ) = %Hf\n", x, y, z);
}
```

Related information

- “math.h” on page 44
- “modfd32(), modfd64(), modfd128() — Extract fractional and integral parts of decimal floating-point value” on page 1087

fmtmsg() — Display a message in the specified format

Standards

Standards / Extensions	C or C++	Dependencies
XPG4.2 Single UNIX Specification, Version 3	both	

Format

```
#define _XOPEN_SOURCE_EXTENDED 1
#include <fmtmsg.h>

int fmtmsg(long classification, const char *label, int severity,
           const char *text, const char *action, const char *tag);
```

General description

The fmtmsg() function can be used to display messages in a specified format instead of the traditional printf() function.

Based on a message's classification component, fmtmsg() writes a formatted message either to standard error, to the console, or to both.

A formatted message consists of up to five components as defined below. The component classification is not part of a message displayed to the user, but defines the source of the message and directs the display of the formatted message.

classification

Contains identifiers from the following groups of major classifications and subclassifications. Any one identifier from a subclass may be used in combination with a single identifier from a different subclass. Two or more identifiers from the same subclass should not be used together, with the exception of identifiers from the display subclass. (Both display subclass identifiers may be used so that messages can be displayed to both standard error and the system console).

Major Classifications
Identifies the source of the condition. Identifiers are: **MM_HARD** (hardware), **MM_SOFT** (software), and **MM_FIRM** (firmware).

Message Source Subclassifications
Identifies the type of software in which the problem is detected. Identifiers are: **MM_APPL** (application), **MM_UTIL** (utility), and **MM_OPSYS** (operating system).

Display Subclassifications
Indicates where the message is to be displayed. Identifiers are: **MM_PRINT** to display the message on the standard error stream, **MM_CONSOLE** to display the message on the system console. One or both identifiers may be used.

Status Subclassifications
Indicates whether the application will recover from the condition. Identifiers are: **MM_RECOVER** (recoverable) and **MM_NRECOV** (non-recoverable).

An additional identifier, **MM_NULLMC**, indicates that no classification component is supplied for the message.

label Identifies the source of the message. The format is two fields separated by a colon. The first field is up to 10 bytes, the second is up to 14 bytes. The constant **__MM_MXLABELLN** defines the maximum length of *label*.

severity
Indicates the seriousness of the condition. Identifiers for the levels of severity are:

MM_HALT
indicates that the application has encountered a severe fault and is halting. Produces the string “HALT”.

MM_ERROR
indicates that the application has detected a fault. Produces the string “ERROR”.

MM_WARNING
indicates a condition that is out of the ordinary, that might be a problem, and should be watched. Produces the string “WARNING”.

MM_INFO
provides information about a condition that is not in error. Produces the string “INFO”.

MM_NOSEV
indicates that no severity level is supplied for the message.

Other provides an unknown severity. Produce the string “SV=n”, where n is the *severity* value specified.

text Describes the error condition that produced the message. The character string is not limited to a specific size. If the character string is empty, then the text produced is unspecified.

action Describes the first step to be taken in the error-recovery process. The fmtmsg() function precedes the action string with the prefix: “TO FIX:”. The action string is not limited to a specific size.

tag An identifier that references on-line documentation for the message. Suggested usage is that tag includes the label and a unique identifying number. A sample tag is “XSI:cat:146”. The constant **__MM_MXTAGLN** defines the maximum length of *tag*.

The MSGVERB environment variable (for message verbosity) tells fmtmsg() which message components it is to select when writing messages to standard error. The value of MSGVERB is a colon-separated list of optional keywords. Valid keywords are: label, severity, text, action, and tag. If MSGVERB contains a keyword for a component and the component's value is not the component's NULL value, fmtmsg() includes that component in the message when writing the message to standard error. If MSGVERB does not include a keyword for a message component, that component is not included in the display of the message. The keywords may appear in any order. If MSGVERB is not defined, if its value is the NULL string, if its value is not of the correct format, or if it contains keywords other than the valid ones listed above, fmtmsg() selects all components.

MSGVERB affects only which components are selected for display to standard error. All message components are included in console messages.

Returned value

fmtmsg() returns one of the following values:

Value **Description**

MM_OK
The function succeeded.

MM_NOCON
The function was unable to generate a console message, but otherwise succeeded.

MM_NOMSG
The function was unable to generate a message on standard error, but otherwise succeeded.

MM_NOTOK
The function failed completely.

Examples

The following is an example of fmtmsg():

```
fmtmsg(MM_PRINT, "XSI:cat", MM_ERROR, "illegal option",
"refer to cat in user's reference manual", "XSI:cat:001")

       produces a complete message in the specified message format:

XSI:cat: ERROR: illegal option
TO FIX: refer to cat in user's reference manual XSI:cat:001
```

The following is another example when the environment variable MSGVERB is set.

```
export MSGVERB=severity:text:action

fmtmsg(MM_PRINT, "XSI:cat", MM_ERROR, "illegal option",
"refer to cat in user's reference manual", "XSI:cat:001")
```

```
    produces a complete message in the specified message format:

ERROR: illegal option
TO FIX: refer to cat in user's reference manual
```

Related information

- “fmtmsg.h” on page 31
- “fprintf(), printf(), sprintf() — Format and write data” on page 588

fnmatch() — Match file name or path name

Standards

Standards / Extensions	C or C++	Dependencies
XPG4 XPG4.2 Single UNIX Specification, Version 3	both	

Format

```
#define _XOPEN_SOURCE
#include <fnmatch.h>

int fnmatch(const char *pattern, const char *string, int flags);
```

General description

The fnmatch() function matches patterns as described in the **XCU** specification, **Section 2.13.1**, **Patterns Matching a Single Character**, and **Section 2.13.2**, **Patterns Matching Multiple Characters**. It checks the string specified by the *string* argument to see if it matches the pattern specified by the *pattern* argument.

The *flags* argument modifies the interpretation of *pattern* and *string*. It is the bitwise inclusive-OR of zero or more of the flags defined in the header <fnmatch.h>. If the FNM_PATHNAME flag is set in *flags*, then a slash character in *string* will be explicitly matched by a slash in *pattern*; it will not be matched by either the asterisk or question-mark special characters, nor by a bracket expression. If FNM_PATHNAME is set and either of these characters would match a slash, the function returns FNM_ESLASH. If the FNM_PATHNAME flag is not set, the slash character is treated as an ordinary character.

If FNM_NOESCAPE is not set in *flags*, a backslash character (\) in *pattern* followed by any other character will match that second character in *string*. In particular, \\ will match a backslash in *string*. If FNM_NOESCAPE is set, a backslash character will be treated as an ordinary character.

If FNM_PERIOD is set in *flags*, then a leading period in *string* will match a period in *pattern*; as described by rule 2 in the **XCU** specification, **Section 2.13.3**, **Patterns Used for Filename Expansion**, where the location of “leading” is indicated by the value of FNM_PATHNAME:

- If FNM_PATHNAME is set, a period is “leading” if it is the first character in *string* or if it immediately follows a slash.
- If FNM_PATHNAME is not set, a period is “leading” only if it is the first character of *string*.

If FNM_PERIOD is not set, then no special restrictions are placed on matching a period. If FNM_PERIOD is set, and a pattern wildcard would match a leading period as defined by the above rules, then the function returns FNM_EPERIOD.

Returned value

If *string* matches the pattern specified by *pattern*, fnmatch() returns 0.

If there is no match, fnmatch() returns FNM_NOMATCH, which is defined in the header <fnmatch.h>.

If an error occurs, fnmatch() returns another nonzero value. See the discussion above for the various possible nonzero returns.

Related information

- "fnmatch.h" on page 31
- "glob() — Generate path names matching a pattern" on page 807
- "wordexp() — Perform shell word expansions" on page 2081

fopen() — Open a file

Standards

Standards / Extensions	C or C++	Dependencies
ISO C POSIX.1 XPG4 XPG4.2 C99 Single UNIX Specification, Version 3	both	

Format

```
#include <stdio.h>

FILE *fopen(const char *__restrict__filename, const char *__restrict__mode);
```

General description

The fopen() function opens the file specified by *filename* and associates a stream with it. The *mode* variable is a character string specifying the type of access requested for the file. The *mode* variable contains one positional parameter followed by optional keyword parameters. The positional parameters are described in Table 25 on page 566 and Table 26 on page 567.

The positional parameters must be passed as lowercase characters.

The keyword parameters can be passed in mixed case. They must be separated by commas. Only one instance of a keyword can be specified.

The file name passed to fopen() often determines the type of file that is opened. A set of file-naming rules exist, which allow you to create an application that references both MVS and HFS files specifically. For details on how fopen() determines the type of file from the *filename* and *mode* strings, see the topics about opening files in *z/OS XL C/C++ Programming Guide*.

Large file support for z/OS UNIX files: Large z/OS UNIX files are supported automatically for AMODE 64 C/C++ applications. AMODE 31 C/C++ applications must be compiled with the option LANGLVL(LONGLONG) and define the _LARGE_FILES feature test macro before any headers are included to enable this function to operate on z/OS UNIX files that are larger than 2 GB in size. File size and offset fields are enlarged to 63 bits in width. Therefore, any other function operating on the file is required to define the _LARGE_FILES feature test macro as well.

Named pipes in multithreaded environment: Do not use fopen() to open named pipes in multithreaded environment. If used, a deadlock will occur. See *z/OS XL C/C++ Programming Guide* for a detailed explanation.

File mode

Restriction: When running with POSIX(OFF) and specifying a mode parameter that includes `t`, for example, `rt`, `rt+`, `r+t`, `wt`, `wt+`, `w+t`, `at`, `at+` or `a+t`, the fopen() request will fail with a message indicating a non-valid mode was specified.

Table 25. Values for the Positional Parameter

File Mode	General Description
r	Open a text file for reading. (The file must exist.)
w	Open a text file for writing. If the `w` mode is specified for a ddname that has DISP=MOD, the behavior is the same as if `a` had been specified. Otherwise, if the file already exists, its contents are destroyed.
a	Open a text file in append mode for writing at the end of the file. fopen() creates the file if it does not exist.
r+	Open a text file for both reading and writing. (The file must exist.)
w+	Open a text file for both reading and writing. If the `w+` mode is specified for a ddname that has DISP=MOD, the behavior is the same as if `a+` had been specified. Otherwise, if the file already exists, its contents are destroyed.
a+	Open a text file in append mode for reading or updating at the end of the file. fopen() creates the file if it does not exist.
rb	Open a binary file for reading. (The file must exist.)
wb	Open an empty binary file for writing. If the `wb` mode is specified for a ddname that has DISP=MOD, the behavior is the same as if `ab` had been specified. Otherwise, if the file already exists, its contents are destroyed.
ab	Open a binary file in append mode for writing at the end of the file. fopen() creates the file if it does not exist.
rt	Open a text file for reading. (The file must exist.)
wt	Open a text file for writing. If the file already exists, its contents are destroyed.
at	Open a text file in append mode for writing at the end of the file. fopen() creates the file if it does not exist.
r+b *or* rb+	Open a binary file for both reading and writing. (The file must exist.)
w+b *or* wb+	Open an empty binary file for both reading and writing. If the `w+b` (or `wb+`) mode is specified for a ddname that has DISP=MOD, the behavior is the same as if `ab+` had been specified. Otherwise, if the file already exists, its contents are destroyed.
a+b *or* ab+	Open a binary file in append mode for reading or updating at the end of the file. fopen() creates the file if it does not exist.
r+t *or* rt+	Open a text file for both reading and writing. (The file must exist.)

Table 25. Values for the Positional Parameter (continued)

File Mode	General Description
w+t *or* wt+	Open a text file for both reading and writing. If the file already exists, its contents are destroyed.
a+t *or* at+	Open a text file in append mode for reading or updating at the end of the file. fopen() creates the file if it does not exist.

Attention: Use the `w`, `w+`, `wb`, `w+b`, and `wb+` parameters with care; data in existing files of the same name will be lost.

Text files contain printable characters and control characters organized into lines. Each line ends with a newline character. The system may insert or convert control characters in an output text stream. For example, `\r` written to an MVS DASD text file will be treated as if `\n` (newline) was written.

Note: When compared, data output to a text stream may not be equal to data input from the same text stream.

Binary files contain a series of characters. For binary files, the system does not translate control characters on input or output. Under z/OS XL C/C++, some types of files are always treated as binary files, even when opened in text mode.

In such cases, a control character is written to the file as binary data. On input, the control character will be read back as it was written. See the topic about the byte stream model in *z/OS XL C/C++ Programming Guide* for more information.

z/OS XL C/C++ has *Record I/O* and *Blocked I/O* file extensions. These files are binary in nature—no data interpretation—and require additional qualifiers: `type=record` and `type=blocked`. See the topics about writing to record I/O files and writing to blocked I/O files in *z/OS XL C/C++ Programming Guide* for more information.

When you open a file with `a`, `a+`, `ab`, `a+b`, or `ab+` mode, all write operations take place at the end of the file. Although you can reposition the file pointer using fseek(), fsetpos(), or rewind(), the write functions move the file pointer back to the end of the file before they carry out any output operation. This action prevents you from overwriting existing data.

When you specify the update mode (using `+` in the second or third position), you can both read from and write to the file. However, when switching between reading and writing, you must include an intervening positioning function such as fseek(), fsetpos(), rewind(), or fflush(). Output may immediately follow input if the EOF was detected.

Table 26. Keyword Parameters for File Mode

Parameter	Description
`abend=`*value*	Controls whether the runtime library should attempt to recover from an abend issued during OS I/O operations against the stream being opened. The *value* can be **abend** or **recover**. See *z/OS XL C/C++ Programming Guide* for more information.
`acc=`*value*	Indicator of the direction of the access of the VSAM data set. The *value* can be **fwd** or **bwd**.

Table 26. Keyword Parameters for File Mode (continued)

Parameter	Description
`acc=bwd`	Sets the file position indicator to the last record. The access direction may be changed by a call to flocate().
`asis`	Indicates that the file name is not to be converted to uppercase but used as is. This option is the default under POSIX. It is also the default for HFS file names (see *z/OS XL C/C++ Programming Guide* for more information).
`blksize=`*value*	Specifies the maximum length, in bytes, of a physical block of records. To check whether your `blksize` parameter is valid and is within its limits, see the appropriate topic in *z/OS XL C/C++ Programming Guide* for the type of file you are opening.
`byteseek`	Indicator to allow byte seeks for a binary file. For more information, see the ftell() and fseek() functions.
`lrecl=`*value*	Specifies the length, in bytes, for fixed-length records and the maximum length for variable-length records. To check whether your `lrecl` parameter is valid and is within its limits, see the appropriate topic in *z/OS XL C/C++ Programming Guide* for the type of file you are opening.
`noseek`	Indicates that the stream may not use any of the reposition functions. This may improve performance.
`password=`*xxxxxxx*	Specifies the password for a VSAM data set.
`recfm=A`	ASA print-control characters
`recfm=F`	Fixed-length, unblocked
`recfm=FA`	Fixed-length, ASA print-control characters
`recfm=FB`	Fixed-length, blocked
`recfm=FM`	Fixed-length, machine print-control codes
`recfm=FS`	Fixed-length, unblocked, standard
`recfm=FBA`	Fixed-length, blocked, ASA print-control characters
`recfm=FBM`	Fixed-length, blocked, machine print-control codes
`recfm=FBS`	Fixed-length, unblocked, standard ASA print-control characters
`recfm=FSA`	Fixed-length, unblocked, standard, ASA print-control characters
`recfm=FSM`	Fixed-length, unblocked, standard, machine print-control codes
`recfm=FBSA`	Fixed-length, blocked, standard, ASA print-control characters
`recfm=FBSM`	Fixed-length, blocked, standard, machine print-control codes
`recfm=U`	Undefined-length
`recfm=UA`	Undefined-length, ASA print control characters
`recfm=UM`	Undefined-length, machine print control codes
`recfm=V`	Variable, unblocked
`recfm=VA`	Variable, ASA print-control characters
`recfm=VB`	Variable, blocked
`recfm=VM`	Variable, machine print-control codes
`recfm=VS`	Variable, unblocked, spanned
`recfm=VBA`	Variable, blocked, ASA print-control characters

Table 26. Keyword Parameters for File Mode (continued)

Parameter	Description
`recfm=VBM`	Variable, blocked, machine print-control codes
`recfm=VBS`	Variable, blocked, spanned
`recfm=VSA`	Variable, unblocked, spanned, ASA print-control characters
`recfm=VSM`	Variable, unblocked, spanned, machine print-control codes
`recfm=VBSA`	Variable, blocked, spanned, ASA print-control characters
`recfm=VBSM`	Variable, blocked, spanned, machine print-control codes
`recfm=*`	Existing file attributes are used if file is opened in write mode. **Note:** Using `recfm=*` is only valid for existing DASD data sets. It is ignored in all other cases.
`recfm=+`	Identical to `recfm=*` with the following exceptions: • If there is no record format for the existing DASD data set, defaults are assigned as if the data set did not exist. • When append mode is used, the fopen() fails. See *z/OS XL C/C++ Programming Guide* for more information about fopen() default attributes.
`samethread`	This parameter specifies that I/O operations against the stream are restricted to the thread in which the stream was opened. The library will not lock the stream in a multithread environment. Use of this keyword can improve performance when the stream does not need to be accessed on different threads.
space	Space attributes for MVS data sets. Within the parameter, you cannot have any imbedded blanks. Where: **u** unit type of space requested **p** primary amount of space requested **s** secondary amount of space requested **d** number of directory space requested See the topic about fopen() and freopen() parameters in *z/OS XL C/C++ Programming Guide* for more information about the syntax of this parameter.
`type=blocked`	This parameter specifies that the file is to be opened for sequential blocked I/O. The file must be opened as a binary file; otherwise, fopen() fails. Read and write operations are done with the fread() and fwrite() functions.
`type=memory`	This parameter identifies this file as a memory file that is accessible only from C programs.
`type=memory(hiperspace)`	If you are using MVS/ESA, you can specify the HIPERSPACE suboption to open a hiperspace memory file. **Restriction:** For AMODE 64 applications, `type=memory(hiperspace)` is treated as `type=memory`.
`type=record`	This parameter specifies that the file is to be opened for sequential record I/O. The file must be opened as a binary file; otherwise, fopen() fails. Read and write operations are done with the fread() and fwrite() functions. This is the default fopen() mode for accessing VSAM clusters.

Returned value

If successful, fopen() returns a pointer to the object controlling the associated stream.

If unsuccessful, fopen() returns a NULL pointer.

fopen() generally fails if parameters are mismatched.

Special behavior for large files for HFS: The following is a possible value of errno:

Error Code
Description

EOVERFLOW
The named file is a regular file and the size of the file cannot be represented correctly in an object of type `off_t`.

Example

CELEBF26

```
/* CELEBF26

   This example attempts to open two files for reading, myfile.dat
   and myfile2.dat.

 */
#include <stdio.h>

int main(void)
{
   FILE *stream;

   /* The following call opens a text file for reading */

   if ((stream = fopen("myfile.dat", "r")) == NULL)
      printf("Could not open data file for reading\n");

   /* The following call opens:
              the file myfile2.dat,
              a binary file for reading and writing,
              whose record length is 80 bytes,
              and maximum length of a physical block is 240 bytes,
              fixed-length, blocked record format
              for sequential record I/O.
   */

   if ( (stream = fopen("myfile2.dat", "rb+, lrecl=80,\
      blksize=240, recfm=fb, type=record")) == NULL )
      printf("Could not open data file for read update\n");
}
```

Related information

- See the topics about dealing with I/O in *z/OS XL C/C++ Programming Guide*.
- “stdio.h” on page 68
- “fclose() — Close file” on page 473
- “fldata() — Retrieve file information” on page 540
- “freopen() — Redirect an open file” on page 620

fork() — Create a new process

Standards

Standards / Extensions	C or C++	Dependencies
POSIX.1 XPG4 XPG4.2 Single UNIX Specification, Version 3	both	

Format

```
#define _POSIX_SOURCE
#include <sys/types.h>
#include <unistd.h>

pid_t fork(void);
```

Note: Although POSIX.1 does not require that the <sys/types.h> include file be included, XPG4 has it as an optional header. Therefore, it is recommended that you include it for portability.

General description

Creates a new process. The new process (the *child process*) is an exact duplicate of the process that calls fork() (the *parent process*), except for the following:

- The child process has a unique process ID (PID) that does not match any active process group ID.
- The child has a different parent process ID, that is, the process ID of the process that called fork().
- The child has its own copy of the parent's file descriptors. Each file descriptor in the child refers to the same open file description as the corresponding file descriptor in the parent.
- The child has its own copy of the parent's open directory streams. Each child's open directory stream can share directory stream positioning with the corresponding parent's directory stream.
- The following elements in the tms structure are set to 0 in the child:

  ```
  tms_utime
  tms_stime
  tms_cutime
  tms_cstime
  ```

  For more information about these elements, see "times() — Get process and child process times" on page 1879.
- The child does not inherit any file locks previously set by the parent.
- The child process has no alarms set (similar to the results of a call to alarm() with an argument value of 0).
- The child has no pending signals.
- The child process has only a single thread. That thread is a copy of the thread in the parent that called fork(). The child process has a different thread ID. If the parent process was multithreaded (invoked pthread_create() at least once), the child process can only safely invoke async-signal-safe functions before it invokes an exec() family function. (This restriction also applies to any process created as the result of the child invoking fork() before it invokes an exec() family function

because the child process is still considered multithreaded.) The child process does not inherit pthread attributes or pthread security environment. See Table 27 for a list of async-signal-safe functions.

In all other respects, the child is identical to the parent. Because the child is a duplicate, it contains the same call to fork() that was in the parent. Execution begins with this fork() call, which returns a value of 0; the child then proceeds with normal execution.

The child address space inherits the following address space attributes of the parent address space:
- Region size
- Time limit

If the parent process is multithreaded, it is the responsibility of the application to ensure that the application data is in a consistent state when the fork() occurs. For example, mutexes that are used to serialize updates to application data may need to be locked before the fork() and unlocked afterwards.

For more information on fork(), refer to *z/OS UNIX System Services Programming: Assembler Callable Services Reference, SA23-2281*.

You can use MVS memory files from a z/OS UNIX program. However, use of the fork() function from the program removes access from a hiperspace memory file for the child process. Use of an exec function from the program clears a memory file when the process address space is cleared.

The child process that results from a fork() in a multithreaded environment can only invoke async-signal-safe functions.

An async-signal-safe function is defined as a function that may be invoked, without restriction, from signal-catching functions. All supported async-signal-safe functions are listed in Table 27.

Table 27. Async-signal-safe library functions

abort()
accept()
access()
aio_error()
aio_return()
aio_suspend()
alarm()
bind()
cfgetispeed()
cfgetospeed()
cfsetispeed()
cfsetospeed()
chdir()
chmod()
chown()
close()
connect()
creat()
dup()
dup2()
execle()
fpathconf()
fstat()
fsync()
ftruncate()
getegid()
geteuid()
getgid()
getgroups()
getpeername()
getpgrp()
getpid()
getppid()
getsockname()
getsockopt()
getuid()
kill()
link()
listen()
lseek()
lstat()
mkdir()
raise()
read()
readlink()
recv()
recvfrom()
recvmsg()
rename()
rmdir()
select()
send()
sendmsg()
sendto()
setgid()
setpgid()
setsid()
setsockopt()
setuid()
shutdown()
sigaction()
sigaddset()
sigdelset()
sigpending()
sigprocmask()
sigqueue()
sigset()
sigsuspend()
socket()
socketpair()
stat()
symlink()
sysconf()
tcdrain()
tcflow()
tcflush()
tcgetattr()
tcgetpgrp()
tcsendbreak()
tcsetattr()
tcsetpgrp()
time()
times()
umask()

Table 27. Async-signal-safe library functions (continued)

execve()	mkfifo()	sigemptyset()	uname()
_Exit()	open()	sigfillset()	unlink()
_exit()	pathconf()	sigismember()	utime()
fchmod()	pause()	sleep()	wait()
fchown()	pipe()	signal()	waitpid()
fcntl()	poll()	sigpause()	write()
fork()			

Interoperability restriction: For POSIX resources, fork() behaves as just described. But in general, MVS resources that existed in the parent do *not* exist in the child. This is true for open streams in MVS data sets and assembler-accessed MVS facilities, such as STIMERS. In addition, MVS allocations (through JCL, SVC99, or ALLOCATE) are not passed to the child process.

Special behavior for z/OS UNIX Services:

1. A prior loaded copy of an HFS program in the same address space is reused under the same circumstances that apply to the reuse of a prior loaded MVS unauthorized program from an unauthorized library by the MVS XCTL service with the following exceptions:
   - If the calling process is in Ptrace debug mode, a prior loaded copy is not reused.
   - If the calling process is not in Ptrace debug mode, but the only prior loaded usable copy found of the HFS program is in storage modifiable by the caller, the prior copy is not reused.
2. If the specified file name represents an external link or a sticky bit file, the program is loaded from the caller's MVS load library search order. For an external link, the external name is only used if the name is eight characters or less, otherwise the caller receives an error from the loadhfs service. For a sticky bit program, the file name is used if it is eight characters or less. Otherwise, the program is loaded from the HFS.
3. If the calling task is in a WLM enclave, the resulting task in the new process image is joined to the same WLM enclave. This allows WLM to manage the old and new process images as one ‘business unit of work’ entity for system accounting and management purposes.

Returned value

If successful, fork() returns 0 to the child process and the process ID of the newly created child to the parent process.

If unsuccessful, fork() fails to create a child process, returns -1 to the parent, and sets errno to one of the following values:

Error Code
    Description

EAGAIN
    There are insufficient resources to create another process, or the process has already reached the maximum number of processes you can run.

ELEMSGERR
    Language Environment message file not available.

ELEMULTITHREAD
    Application contains a language that does not support fork() in a

multithreaded environment, or the multithreaded fork() is being attempted while running in a Language Environment preinitialization (CEEPIPI) environment.

ELENOFORK
Application contains a language that does not support fork().

ENOMEM
The process requires more space than is available.

Example

CELEBF27

```
/* CELEBF27

   This example creates a new child process.

 */
#define _POSIX_SOURCE
#include <sys/types.h>
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <sys/wait.h>

main() {
  pid_t pid;
  int status;

  if ((pid = fork()) < 0)
    perror("fork() error");
  else if (pid == 0) {
    puts("This is the child.");
    printf("Child's pid is %d and my parent's is %d\n",
           (int) getpid(), (int) getppid());
    exit(42);
  }
  else {
    puts("This is the parent.");
    printf("Parent's pid is %d and my child's is %d\n",
           (int) getpid(), (int) pid);
    puts("I'm waiting for my child to complete.");
    if (wait(&status) == -1)
      perror("wait() error");
    else if (WIFEXITED(status))
           printf("The child exited with status of %d\n",
                  WEXITSTATUS(status));
         else
           puts("The child did not exit successfully");
  }
}
```

Output

```
This is the parent.
This is the child.
Child's pid is 1114120 and my parent's is 2293766
Parent's pid is 2293766 and my child's is 1114120
I'm waiting for my child to complete.
The child exited with status of 42
```

Related information

- "sys/types.h" on page 75
- "unistd.h" on page 82
- "alarm() — Set an alarm" on page 156

- “exec functions” on page 436
- “fcntl() — Control open file descriptors” on page 474
- “getrlimit() — Get current or maximum resource consumption” on page 765
- “kill() — Send a signal to a process” on page 931
- “nice() — Change priority of a process” on page 1145
- “putenv() — Change or add an environment variable” on page 1348
- “semop() — Semaphore operations” on page 1488
- “shmat() — Shared memory attach operation” on page 1598
- “sysconf() — Determine system configuration options” on page 1798
- “times() — Get process and child process times” on page 1879
- “ulimit() — Get or set process file size limits” on page 1942
- “wait() — Wait for a child process to end” on page 1992

fortrc() — Return FORTRAN return code

Standards

Standards / Extensions	C or C++	Dependencies
C Library	both	

Format

```
#include <stdlib.h>

int fortrc(void);
```

External entry point: `@@FORTRC`, `__fortrc`

General description

Restriction: This function is not supported in AMODE 64.

The fortrc() function returns the value specified on the FORTRAN RETURN statement issued by the last FORTRAN routine called from the C program.

To avoid infringing on the user's name space, this nonstandard function has two names. One name is prefixed with two underscore characters, and one name is not. The name without the prefix underscore characters is exposed only when you use LANGLVL(EXTENDED).

To use this function, you must either invoke the function using its external entry point name (that is, the name that begins with two underscore characters), or compile with LANGLVL(EXTENDED). When you use LANGLVL(EXTENDED) any relevant information in the header is also exposed.

The FORTRAN routine called must be identified to C as a FORTRAN routine using the following preprocessor directive:

```
#pragmalinkage(identifier,FORTRAN,RETURNCODE).
```

The function fortrc() should be called immediately after a call to the FORTRAN routine *identifier* or else results are unpredictable.

If you do not include `stdlib.h` in your source code or you use the compile-time option LANGLVL(ANSI), then you must use `_ _fortrc` to call the function.

Related information

- “stdlib.h” on page 70

fp_clr_flag() — Reset floating-point exception status flag

Standards

Standards / Extensions	C or C++	Dependencies
	both	OS/390 V2R6

Format

```
#include <float.h>
#include <fpxcp.h>

void fp_clr_flag(mask)
fpflag_t mask;
```

General description

The fp_clr_flag() function resets the exception status flags defined by the *mask* parameter to 0 (false). The remaining flags in the exception status remain unchanged.

Note: This function works only in IEEE Binary Floating-Point. See “IEEE binary floating-point” on page 94 for more information about IEEE Binary Floating-Point.

The **fpxcp.h** file defines the following names for the flags indicating floating-point exception status:

FP_INVALID
Invalid operation summary

FP_OVERFLOW
Overflow

FP_UNDERFLOW
Underflow

FP_DIV_BY_ZERO
Division by 0

FP_INEXACT
Inexact result

Users can reset multiple exception flags using the fp_clr_flag() function by OR-ing the names of individual flags. For example, the following resets both the overflow and inexact flags.

```
fp_clr_flag(FP_OVERFLOW | FP_INEXACT)
```

Returned value

fp_clr_flag() returns no values.

Related information

- “float.h” on page 29
- “fpxcp.h” on page 32
- “fp_raise_xcp() — Raise a floating-point exception” on page 577

- "fp_read_flag() — Return the current floating-point exception status" on page 578
- "__isBFP() — Determine application floating-point format" on page 904

fp_raise_xcp() — Raise a floating-point exception

Standards

Standards / Extensions	C or C++	Dependencies
	both	OS/390 V2R6

Format

```
#include <fpxcp.h>

int fp_raise_xcp(mask)
fpflag_t mask;
```

General description

The fp_raise_xcp() function causes floating-point exceptions defined by the *mask* parameter to be raised immediately.

If the exceptions defined by the *mask* parameter are enabled and the program is running in serial mode, the signal for floating-point exceptions, SIGFPE, is raised.

Note: This function works only in IEEE Binary Floating-Point. See "IEEE binary floating-point" on page 94 for more information about IEEE Binary Floating-Point.

The **fpxcp.h** file defines the following names for the flags indicating floating-point exception status:

FP_INVALID
Invalid operation summary

FP_OVERFLOW
Overflow

FP_UNDERFLOW
Underflow

FP_DIV_BY_ZERO
Division by 0

FP_INEXACT
Inexact result

Users can cause multiple exceptions using fp_raise_xcp() by OR-ing the names of individual flags. For example, the following causes both overflow and division by 0 exceptions to occur.

```
fp_raise_xcp(FP_OVERFLOW | FP_DIV_BY_ZERO)
```

If more than one exception is included in the mask variable, the exceptions are raised in the following order:

1. Non-valid operation
2. Division by zero
3. Underflow
4. Overflow

5. Inexact result

Thus, if the user exception handler does not disable further exceptions, one call to the fp_raise_xcp() function can cause the exception handler to be entered many times.

Returned value

If successful, fp_raise_xcp() returns 0.

If unsuccessful, fp_raise_xcp() returns nonzero.

Related information

- "fpxcp.h" on page 32
- "fp_clr_flag() — Reset floating-point exception status flag" on page 576
- "fp_read_flag() — Return the current floating-point exception status"
- "__isBFP() — Determine application floating-point format" on page 904

fp_read_flag() — Return the current floating-point exception status

Standards

Standards / Extensions	C or C++	Dependencies
	both	OS/390 V2R6

Format

```
#include <float.h>
#include <fpxcp.h>

fpflag_t fp_read_flag()
```

General description

The fp_read_flag() function returns the current floating-point exception status.

These functions aid in determining both when an exception has occurred and the exception type. These functions can be called explicitly around blocks of code that may cause a floating-point exception.

According to the IEEE Standard for Binary Floating-Point Arithmetic, the following types of floating-point operations must be signaled when detected in a floating-point operation:
- Non-valid operation
- Division by zero
- Overflow
- Underflow
- Inexact

A non-valid operation occurs when the result cannot be represented (for example, a square root operation on a number less than 0).

The IEEE Standard for Binary Floating-Point Arithmetic states: "For each type of exception, the implementation shall provide a status flag that shall be set on any occurrence of the corresponding exception when no corresponding trap occurs. It shall be reset only at the user's request. The user shall be able to test and to alter the status flags individually, and should further be able to save and restore all five at one time.”

Floating-point operations can set flags in the floating-point exception status but cannot clear them. Users can clear a flag in the floating-point exception status using an explicit software action such as the fp_clr_flag (0) subroutine.

Note: This function works only in IEEE Binary Floating-Point. See “IEEE binary floating-point” on page 94 for more information about IEEE Binary Floating-Point.

The **fpxcp.h** file defines the following names for the flags indicating floating-point exception status:

FP_INVALID
non-valid operation summary

FP_OVERFLOW
Overflow

FP_UNDERFLOW
Underflow

FP_DIV_BY_ZERO
Division by 0

FP_INEXACT
Inexact result

Returned value

fp_read_flag() returns the current floating-point exception status. The flags in the returned exception status can be tested using the flag definitions above. You can test individual flags or sets of flags.

Related information

- IEEE Standard for Binary Floating-Point Arithmetic (ANSI/IEEE Standards 754-1985 and 854-1987)
- “float.h” on page 29
- “fpxcp.h” on page 32
- “fp_clr_flag() — Reset floating-point exception status flag” on page 576
- “fp_raise_xcp() — Raise a floating-point exception” on page 577
- “__isBFP() — Determine application floating-point format” on page 904

fp_read_rnd() — Determine rounding mode

Standards

Standards / Extensions	C or C++	Dependencies
	both	OS/390 V2R6

Format

```
#define _AIX_COMPATIBILITY 1
#include <float.h>

fprnd_t fp_read_rnd(void);
```

General description

For an application running in binary floating-point mode, the fp_read_rnd() function returns the current rounding mode indicated by the rounding mode field of the floating-point control (FPC) register. For an application running in hexadecimal floating-point mode, fp_read_rnd() returns 0.

Note: This function will not return or update decimal floating-point rounding mode bits.

Returned value

For an application running in IEEE Binary Floating-Point mode, fp_read_rnd() returns the following:

Value **Rounding Mode**

_FP_RND_RZ
Round toward 0

_FP_RND_RN
Round to nearest

_FP_RND_RP
Round toward +infinity

_FP_RND_RM
Round toward -infinity

For an application running in hexadecimal floating-point mode, fp_read_rnd() returns 0.

Related information

- “float.h” on page 29
- “fp_swap_rnd() — Swap rounding mode”
- “__isBFP() — Determine application floating-point format” on page 904

fp_swap_rnd() — Swap rounding mode

Standards

Standards / Extensions	C or C++	Dependencies
	both	OS/390 V2R6

Format

```
#define _AIX_COMPATIBILITY 1
#include <float.h>

fprnd_t fp_swap_rnd(RoundMode)
fprnd_t RoundMode
```

General description

For an application running in IEEE Binary Floating-Point mode, the fp_swap_rnd() function returns the current rounding mode specified by the rounding mode field of the floating-point control (FPC) register and sets the rounding mode field in the FPC register based on the value of *RoundMode* as follows:

Value **Rounding Mode**

_FP_RND_RZ
Round toward 0

_FP_RND_RN
Round to nearest

_FP_RND_RP
Round toward +infinity

_FP_RND_RM
Round toward -infinity

Notes:

1. When processing IEEE Binary Floating-Point values, the z/OS XL C/C++ runtime library math functions require IEEE rounding mode of round to nearest. The z/OS XL C/C++ runtime library takes care of setting round to nearest rounding mode while executing math functions and restoring application rounding mode before returning to the caller.
2. This function will not return or update decimal floating-point rounding mode bits.

Returned value

For an application running in hexadecimal floating-point mode, fp_swap_rnd() returns 0.

For an application running in IEEE Binary Floating-Point mode, fp_swap_rnd() returns the previous (changed from) rounding mode as follows:

Value **Rounding Mode**

_FP_RND_RZ
Round toward 0

_FP_RND_RN
Round to nearest

_FP_RND_RP
Round toward +infinity

_FP_RND_RM
Round toward -infinity

Related information

- “float.h” on page 29
- “fp_read_rnd() — Determine rounding mode” on page 579
- “__isBFP() — Determine application floating-point format” on page 904

fpathconf() — Determine configurable path name variables

Standards

Standards / Extensions	C or C++	Dependencies
POSIX.1 XPG4 XPG4.2 Single UNIX Specification, Version 3	both	

Format

```
#define _POSIX_SOURCE
#include <unistd.h>

long fpathconf(int fildes, int varcode);
```

General description

Determines the value of a configuration variable (*varcode*) associated with a particular file descriptor (*fildes*).

fpathconf() works exactly like pathconf(), except that it takes a file descriptor as an argument rather than taking a path name.

The *varcode* argument can be any one of a set of symbols defined in the unistd.h header file. Each symbol stands for a configuration variable. These are the possible symbols:

_PC_LINK_MAX
Represents **LINK_MAX**, the maximum number of links the file can have. If *pathname* is a directory, fpathconf() returns the maximum number of links that can be established to the directory itself.

_PC_MAX_CANON
Represents **MAX_CANON**, the maximum number of bytes in a terminal canonical input line. *pathname* must refer to a character special file for a terminal.

_PC_MAX_INPUT
Represents **MAX_INPUT**, the minimum number of bytes for which space will be available in a terminal input queue. This input space is the maximum number of bytes that a portable application will allow an end user to enter before the application actually reads the input. *pathname* must refer to a character special file for a terminal.

_PC_NAME_MAX
Represents **NAME_MAX**, the maximum number of characters in a file name (not including any terminating NULL character if the file name is stored as a string). This limit refers only to the file name itself, that is, the last component of the file's path name. fpathconf() returns the maximum length of file names.

_PC_PATH_MAX
Represents **PATH_MAX**, the maximum number of characters in a complete path name (not including any terminating NULL if the path name is stored as a string). fpathconf() returns the maximum length of a relative path name.

_PC_PIPE_BUF
Represents **PIPE_BUF**, the maximum number of bytes that can be written to a pipe as one unit. If more than this number of bytes is written to a pipe, the operation can take more than one physical write operation and can require more than one physical read operation to read the data on the other end of the pipe. If *pathname* is a FIFO special file, fpathconf() returns the value for the file itself. If *pathname* is a directory, fpathconf() returns the value for any FIFOs that exist or can be created under the directory. If *pathname* is any other kind of file, an errno of EINVAL (see description below) will be returned.

_PC_CHOWN_RESTRICTED
Represents _POSIX_CHOWN_RESTRICTED defined in the unistd.h header file. This symbol indicates that the use of chown() is restricted; see the callable service chown() for more details. If *pathname* is a directory, fpathconf() returns the value for any kind of file under the directory, but not for subdirectories of the directory.

_PC_NO_TRUNC
Represents _POSIX_NO_TRUNC defined in the unistd.h header file. This symbol indicates that an error should be generated if a file name is longer than **NAME_MAX**. If *pathname* refers to a directory, the value returned by fpathconf() applies to all files under that directory.

_PC_VDISABLE
Represents _POSIX_VDISABLE defined in the unistd.h header file. This symbol indicates that terminal special characters can be disabled using this character value, if it is defined. See the callable service tcsetattr() for details. *pathname* must refer to a character special file for a terminal.

_PC_ACL
Returns 1 if an access control mechanism is supported by the security product for the file identified by the file descriptor.

_PC_ACL_ENTRIES_MAX
Returns the maximum number of ACL entries in an ACL for the file or directory identified by the file descriptor.

Returned value

If a particular variable has no limit, fpathconf() returns -1 but does not change errno.

If successful, fpathconf() returns the value of the variable requested in *varcode*.

If unsuccessful, fpathconf() returns -1 and sets errno to one of the following values:

Error Code
Description

EBADF
fildes is not a valid open file descriptor.

EINVAL
varcode is not a valid variable code, or the given variable cannot be associated with the specified file.

- If *varcode* refers to **MAX_CANON**, **MAX_INPUT**, or _POSIX_VDISABLE, and *pathname* does not refer to a character special file, fpathconf() returns -1 and sets errno to EINVAL.
- If *varcode* refers to **NAME_MAX**, **PATH_MAX**, or POSIX_NO_TRUNC, and *pathname* does not refer to a directory, fpathconf() returns the requested information.
- If *varcode* refers to PC_PIPE_BUF and *pathname* refers to a pipe or a FIFO, the value returned applies to the referenced object itself. If *pathname* refers to a directory, the value returned applies to any FIFOs that exist or can be created within the directory. If *pathname* refers to any other type of file, the function sets errno to EINVAL.

Example

CELEBF29

```
/* CELEBF29

   This example uses fpathconf() with __PC_NAME_MAX to determine the value
   of the NAME_MAX configuration variable.

 */
#define _POSIX_SOURCE
#include <errno.h>
#include <fcntl.h>
#include <sys/stat.h>
#include <sys/types.h>
#include <unistd.h>
#undef _POSIX_SOURCE
#include <stdio.h>

main() {
  long result;
  char fn[]="temp.file";
  int  fd;

  if ((fd = creat(fn, S_IRUSR)) < 0)
    perror("creat() error");
  else {
    errno = 0;
    puts("examining NAME_MAX limit for current working directory's");
    puts("filesystem:");
    if ((result = fpathconf(fd, _PC_NAME_MAX)) == -1)
      if (errno == 0)
        puts("There is no limit to NAME_MAX.");
      else perror("fpathconf() error");
    else
      printf("NAME_MAX is %ld\n", result);
    close(fd);
    unlink(fn);
  }
}
```

Output

```
examining NAME_MAX limit for current working directory's
file system:
NAME_MAX is 255
```

Related information

- “unistd.h” on page 82
- “open() — Open a file” on page 1152
- “pathconf() — Determine configurable path name variables” on page 1169

fpclassify() — Classifies an argument value

Standards

Standards / Extensions	C or C++	Dependencies
C99 Single UNIX Specification, Version 3 C/C++ DFP C++ TR1 C99	both	z/OS V1R8

Format

```
#define _ISOC99_SOURCE
#include <math.h>

int fpclassify (real-floating x);

#define __STDC_WANT_DEC_FP__
#include <math.h>

int fpclassify([real-floating|decimal-floating] x);

#define _TR1_C99
#include <math.h>

int fpclassify(real-floating x);
```

General description

This macro or function template classifies its argument value as NaN, infinite, normal, subnormal or zero based on the type of its argument. If the argument is represented in a format wider than its semantic type, then it is converted to its semantic type and then it is classified.

Function	Hex	IEEE
fpclassify	X	X

Notes:

1. To use IEEE decimal floating-point, the hardware must have the Decimal Floating-Point Facility installed.
2. This function works in IEEE decimal floating-point format. See "IEEE Decimal Floating-Point" for more information.

Returned value

fpclassify() returns:

- FP_NAN if the argument is Not-a-Number.
- FP_INFINITE if the argument is plus or minus infinity.
- FP_ZERO if the argument is of value zero.
- FP_SUBNORMAL if the argument is too small to be represented in the normal format.
- FP_NORMAL if none of the above.

Special behavior in hex:

- FP_ZERO if the argument is of value zero.
- FP_NORMAL if the argument is a normalized number.
- FP_SUBNORMAL if the argument is an unnormalized number.

Related information

- “math.h” on page 44

__fpending() — Retrieve number of bytes pending for write

Standards

Standards / Extensions	C or C++	Dependencies
Language Environment	both	

Format

```
#include <stdio.h>
#include <stdio_ext.h>

size_t __fpending(FILE *stream);

#define _OPEN_SYS_UNLOCKED 1
#include <stdio.h>
#include <stdio_ext.h>

size_t __fpending_unlocked(FILE *stream);
```

General description

The __fpending() function retrieves the number of bytes pending to be written in the output buffer associated with the specified binary stream. When the stream is opened for text processing, the __fpending() function retrieves the number of characters pending to be written.

The __fpending_unlocked() function is equivalent to the __fpending() function with the exception that it is not thread-safe. This function can be safely used in a multithreaded application if it is called while the invoking thread owns the (FILE *) object, such as after a successful call to either the flockfile() or ftrylockfile() function.

The __fpending() function cannot be used on buffers in read mode or opened read-only.

Usage notes

1. If __fpending() is called when the stream is currently reading, __fpending() returns 0 and sets errno to a nonzero value.
2. If __fpending() is called on a data set opened `type=record`, __fpending() returns 0 and sets errno to a nonzero value.
3. For text data sets, any newline characters representing record boundaries written to the buffer will be included in the returned value.

Returned value

The __fpending() functions return the number of bytes or characters pending to be written in the current buffer, depending on the type of stream. Otherwise, the __fpending() functions return 0. If an error has occurred, __fpending() functions return 0 and set errno to nonzero.

When the stream is wide-oriented text, the __fpending() functions return a value measured in wide characters.

An application wishing to check for error situations should set errno to 0, then call __fpending(), and then check errno. If errno is nonzero, assume that an error has occurred.

Error Code
Description

EBADF
The stream specified by *stream* is not valid.

Example

CELEBF89

```
/* CELEBF89

   This example writes and reads data to a file while querying the
   stream for information about data in the I/O buffer.

*/

#include <stdio.h>
#include <stdio_ext.h>

void main() {
   FILE *f;
   char filename[FILENAME_MAX] = "myfile.dat";
   char data[128] = "There are 34 bytes in this buffer\n";
   int datalen = strlen(data);
   size_t n = 0;

   f = fopen(filename,"wb+");
   if (f == NULL) {
      perror("fopen() failed\n");
      return;
   }

   if (__fwritable(f)) printf("Writing is allowed on the open stream\n");
   if (__freadable(f)) printf("Reading is allowed on the open stream\n");

   n = fputs(data,f);
   if (n == EOF) {
      perror("fputs() failed\n");
      return;
   }

   n = __fpending(f);
   printf("There are %d bytes in the buffer pending to be written\n", n);

   if (__fwriting(f)) printf("The last operation on the stream was a write\n");

   rewind(f);

   n = fgetc(f);

   n = __freadahead(f);
   printf("There are %d bytes remaining to be read from the buffer\n", n);

   if (__freading(f)) printf("The last operation on the stream was a read\n");

   return;
}
```

Output

```
Writing is allowed on the open stream
Reading is allowed on the open stream
There are 34 bytes in the buffer pending to be written
The last operation on the stream was a write
There are 33 bytes remaining to be read from the buffer
The last operation on the stream was a read
```

Related information

- "stdio.h" on page 68
- "stdio_ext.h" on page 70
- "__freadahead() — Retrieve number of bytes remaining in input buffer" on page 614

fprintf(), printf(), sprintf() — Format and write data

Standards

Standards / Extensions	C or C++	Dependencies
C/C++ DFP ISO C POSIX.1 XPG4 XPG4.2 C99 Single UNIX Specification, Version 3 Language Environment	both	z/OS V1.8

Format

```
#include <stdio.h>

int fprintf(FILE *__restrict__stream, const char *__restrict__format-string, ...);
int printf(const char *__restrict__format-string, ...);
int sprintf(char *__restrict__buffer, const char *__restrict__format-string, ...);

#define _OPEN_SYS_UNLOCKED_EXT 1
#include <stdio.h>

int fprintf_unlocked(FILE *__restrict__stream,
                     const char *__restrict__format-string, ...);
int printf_unlocked(const char *__restrict__format-string, ...);
```

General description

These three related functions are referred to as the fprintf family.

The fprintf() function formats and writes output to a *stream*. It converts each entry in the *argument list*, if any, and writes to the stream according to the corresponding format specification in the *format-string*. The fprintf() function cannot be used with a file that is opened using `type=record` or `type=blocked`.

The printf() function formats and writes output to the standard output stream `stdout`. printf() cannot be used if `stdout` has been reopened using `type=record` or `type=blocked`.

The sprintf() function formats and stores a series of characters and values in the array pointed to by *buffer*. Any *argument-list* is converted and put out according to the corresponding format specification in the *format-string*. If the strings pointed to by *buffer* and *format* overlap, behavior is undefined.

fprintf() and printf() have the same restriction as any write operation for a read immediately following a write or a write immediately following a read. Between a write and a subsequent read, there must be an intervening flush or reposition. Between a read and a subsequent write, there must also be an intervening flush or reposition unless an EOF has been reached.

The *format-string* consists of ordinary characters, escape sequences, and conversion specifications. The ordinary characters are copied in order of their appearance. Conversion specifications, beginning with a percent sign (%) or the sequence (%n$) where n is a decimal integer in the range [1,NL_ARGMAX], determine the output format for any *argument-list* following the *format-string*. The *format-string* can contain multibyte characters beginning and ending in the initial shift state. When the *format-string* includes the use of the optional prefix *ll* to indicate the size expected is a long long datatype then the corresponding value in the argument list should be a long long datatype if correct output is expected.

fprintf_unlocked() is functionally equivalent to fprintf() with the exception that it is not thread-safe. This function can safely be used in a multithreaded application if and only if it is called while the invoking thread owns the (FILE*) object, as is the case after a successful call to either the flockfile() or ftrylockfile() function.

printf_unlocked() is functionally equivalent to printf() with the exception that it is not thread-safe. This function can safely be used in a multithreaded application if and only if it is called while the invoking thread owns the (FILE*) object, as is the case after a successful call to either the flockfile() or ftrylockfile() function.

The *format-string* is read from left to right. When the first format specification is found, the value of the first *argument* after the *format-string* is converted and output according to the format specification. The second format specification causes the second *argument* after the *format-string* to be converted and output, and so on through the end of the *format-string*. If there are more arguments than there are format specifications, the extra arguments are evaluated and ignored. The results are undefined if there are not enough arguments for all the format specifications. The format specification is illustrated below.

Format Specification for fprintf(), printf(), and sprintf()

```
►►─%──┬───────┬──┬───────┬──┬───────────────┬──┬─────┬─conversion specifier──────────────►◄
      └─flags─┘  └─width─┘  └─.─precision─┘  ├─h───┤
                                             ├─hh──┤
                                             ├─l───┤
                                             ├─ll──┤
                                             ├─j───┤
                                             ├─t───┤
                                             ├─z───┤
                                             ├─D───┤
                                             ├─DD──┤
                                             ├─H───┤
                                             ├─L───┤
                                             ├─v───┤
                                             ├─vh──┤
                                             ├─hv──┤
                                             ├─vl──┤
                                             ├─lv──┤
                                             ├─vll─┤
                                             ├─llv─┤
                                             ├─vL──┤
                                             └─Lv──┘
```

Each field of the format specification is a single character or number signifying a particular format option. The *type* character, which appears after the last optional format field, determines whether the associated argument is interpreted as a character, a string, a number, or pointer. The simplest format specification contains only the percent sign and a *type* character (for example, %s).

Percent sign: If a percent sign (%) is followed by a character that has no meaning as a format field, the character is simply copied to `stdout`. For example, to print a percent sign character, use %%.

Flag characters: The *flag* characters in Table 28 are used for the justification of output and printing of thousands' grouping characters, signs, blanks, decimal-points, octal and hexadecimal prefixes, the semantics for `wchar_t` precision unit and the separator for vector data type output conversion. Notice that more than one *flag* can appear in a format specification. This is an optional field.

Table 28. Flag Characters for fprintf Family

Flag	Meaning	Default
'	**Added for XPG4:**The integer portion of the result of a decimal conversion(%i,%d,%u, %f,%g or %G) will be formatted with the thousands' grouping characters.	No grouping.
-	Left-justify the result within the field width.	Right-justify.
+	Prefix the output value with a sign (+ or -) if the output value is of a signed type.	Sign appears only for negative signed values (-).

Table 28. Flag Characters for fprintf Family (continued)

Flag	Meaning	Default
blank(' ')	Prefix the output value with a blank if the output value is signed and positive. The `+` flag overrides the *blank* flag if both appear, and a positive signed value will be output with a sign.	No blank.
#	When used with the `o`, `x`, or `X` formats, the `#` flag prefixes any nonzero output value with 0, 0x, or 0X, respectively. For `o` conversion, it increases the precision, if and only if necessary, to force the first digit of the result to be a zero (if the value and precision are both 0, a single 0 will be printed)	No prefix.
	When used with the `f`, `e`, or `E` formats, the `#` flag forces the output value to contain a decimal-point in all cases. The decimal-point is sensitive to the LC_NUMERIC category of the same current locale.	Decimal-point appears only if digits follow it.
	When used with the `g` or `G` formats, the `#` flag forces the output value to contain a decimal-point in all cases and prevents the truncation of trailing zeros.	Decimal-point appears only if digits follow it; trailing zeros are truncated.
	When used with the `ls` or `S` format, the `#` flag causes precision to be measured in wide characters.	Precision indicates the maximum number of bytes to be output.
0	When used with the `d`, `i`, `o`, `u`, `x`, `X`, `e`, `E`, `f`, `g`, or `G` formats, the `0` flag causes leading `0`'s to pad the output to the field width. The `0` flag is ignored if precision is specified for an integer or if the `-` flag is specified.	Space padding.
`,` `;` `:` `_`	Indicates the separator character between each two components in the vector data type conversion. Only one separator character can be specified.	The default value for the separator character is a space (' '), unless the `c` format is being used. For the `c` format, there is no separator at all if no separator character is specified.

The code point for the # character varies between the EBCDIC encoded character sets. The definition of the # character is based on the current LC_SYNTAX category. The default C locale expects the # character to use the code point for encoded character set IBM-1047.

When the LC_SYNTAX category is set using setlocale(), the format strings passed to the printf() functions must use the same encoded character set as is specified for the LC_SYNTAX category.

The `#` flag should not be used with `c`, `lc`, `C`, `d`, `i`, `u`, `s`, or `p` conversion specifier.

Output width: The *width* argument is a nonnegative decimal integer controlling the minimum number of characters printed. If the number of characters in the output value is less than the specified *width*, blanks are added on the left or the right (depending on whether the – flag is specified) until the minimum width is reached.

The width never causes a value to be truncated; if the number of characters in the output value is greater than the specified *width*, or *width* is not given, all characters of the value are output (subject to the *precision* specification).

The *width* specification can be an asterisk (*); if it is, an argument from the argument list supplies the value. The *width* argument must precede the value being formatted in the argument list. This is an optional field.

If *format-string* contains the %n$ form of conversion specification, *width* can be indicated by the sequence *m$, where m is a decimal integer in the range [1,NL_ARGMAX] giving the position of an integer argument in the argument list containing the field width.

Output precision: The *precision* argument is a nonnegative decimal integer preceded by a period. It specifies the number of characters to be output, or the number of decimal places. Unlike the *width* specification, the *precision* can cause truncation of the output value or rounding of a floating-point value.

Be aware that the rounding of floating-point values may not always occur as expected based on the decimal value of the number. This is because the internal binary representation cannot always be an exact representation of the decimal value, so the rounding may occur on an inexact value. This is true of both OS/390 hexadecimal and IEEE 754 binary floating- point formats.

The *precision* argument can be an asterisk (*); if it is, an argument from the argument list supplies the value. The *precision* argument must precede the value being formatted in the argument list. The *precision* field is optional.

If *format-string* contains the %n$ form of conversion specification, *precision* can be indicated by the sequence *m$, where m is a decimal integer in the range [1,NL_ARGMAX] giving the position of an integer argument in the argument list containing the field precision.

The interpretation of the *precision* value and the default when the *precision* is omitted depend upon the *type*, as shown in Table 29.

Table 29. Precision Argument in fprintf Family

Type	Meaning	Default
d i o u x X	*Precision* specifies the minimum number of digits to be output. If the number of digits in the argument is less than *precision*, the output value is padded on the left with zeros. The value is not truncated when the number of digits exceeds *precision*.	Default *precision* is 1. If *precision* is 0, or if the period (.) appears without a number following it, the *precision* is set to 0. When *precision* is 0, conversion of the value zero results in no characters.

Table 29. Precision Argument in fprintf Family (continued)

Type	Meaning	Default
e E f F	*Precision* specifies the number of digits to be output after the decimal-point. The last digit output is rounded. The decimal-point is sensitive to the LC_NUMERIC category of the current locale.	Default *precision* is 6. If *precision* is 0 or the period appears without a number following it, no decimal-point is output.
a A	For non decimal floating-point numbers, *precision* specifies the number of hexadecimal digits to be output after the decimal-point character. For decimal floating-point numbers, precision determines the style of formatting to be used.	For non decimal floating-point numbers, default *precision* is 6. If *precision* is 0, no decimal-point is output. For decimal floating-point numbers, precision is determined by the quantum of the decimal floating-point number. Refer to Table 30 on page 594 for more details.
g G	*Precision* specifies the maximum number of significant digits output.	All significant digits are output.
c	No effect.	The character is output.
C lc	No effect.	The wide character is output.
s	*Precision* specifies the maximum number of characters to be output. Characters in excess of *precision* are not output.	Characters are output until a NULL character is encountered.
S ls	*Precision* specifies the maximum number of bytes to be output. Bytes in excess of *precision* are not output; however, multibyte integrity is always preserved.	`wchar_t` characters are output until a NULL character is encountered.

Optional prefix: The optional prefix characters used to indicate the size of the argument expected or size of a single component in a vector data type output conversion are explained:

Prefix	Meaning
h	Specifies that the `d`, `i`, `o`, `u`, `x`, or `X` conversion specifier applies to a short or unsigned short argument (the argument will have been promoted according to the integer promotions, but its value shall be converted to short or unsigned short before printing); or that a following `n` conversion specifier applies to a pointer to a short argument.
hh	Specifies that the `d`, `i`, `o`, `u`, `x`, or `X` conversion specifier applies to a signed char or unsigned char argument (the argument will have been promoted according to the integer promotions, but its value shall be converted to signed char or unsigned char before printing); or that a following `n` conversion specifier applies to a pointer to a signed char argument.
l	(ell) Specifies that the `d`, `i`, `o`, `u`, `x`, or `X` conversion specifier applies to a long or unsigned long argument; that a following `n` conversion specifier applies to a pointer to a long argument; that the `c` conversion specifier applies to a wint_t argument; that the `s` conversion specifier applies to a pointer to a wchar_t argument. It has no effect on the `a`, `A`, `e`, `E`, `f`, `F`, `g`, or `G` conversion specifier.

ll (ell-ell) Specifies that the `d`, `i`, `o`, `u`, `x`, or `X` conversion specifier applies to a long long or unsigned long long argument; or that the `n` conversion specifier applies to a pointer to a long long argument.

j Specifies that the `d`, `i`, `o`, `u`, `x`, or `X` conversion specifier applies to an intmax_t or uintmax_t argument; or that the `n` conversion specifier applies to a pointer to an intmax_t argument.

t Specifies that the `d`, `i`, `o`, `u`, `x`, or `X` conversion specifier applies to a ptrdiff_t or the corresponding unsigned type argument; or that the `n` conversion specifier applies to a pointer to a ptrdiff_t argument.

z Specifies that the `d`, `i`, `o`, `u`, `x`, or `X` conversion specifier applies to a size_t or the corresponding signed integer type argument; or that the `n` conversion specifier applies to a pointer to a signed integer type corresponding to a size_t argument.

D Specifies that the `a`, `A`, `e`, `E`, `f`, `F`, `g`, or `G` conversion specifier applies to an _Decimal64 argument.

DD Specifies that the `a`, `A`, `e`, `E`, `f`, `F`, `g`, or `G` conversion specifier applies to an _Decimal128 argument.

H Specifies that the `a`, `A`, `e`, `E`, `f`, `F`, `g`, or `G` conversion specifier applies to an _Decimal32 argument.

L Specifies that the `a`, `A`, `e`, `E`, `f`, `F`, `g`, or `G` conversion specifier applies to a long double argument.

v Specifies that a following `c`, `d`, `i`, `u`, `o`, `x`, or `X` conversion specifier applies to a `vector signed char`, `vector unsigned char`, or `vector bool char` parameter. It consumes one argument and interprets the data as a series of sixteen 1-byte components.

vh, hv Specifies that a following `d`, `i`, `u`, `o`, `x`, or `X` conversion specifier applies to a `vector signed short`, `vector unsigned short`, or `vector bool short` parameter. It consumes one argument and interprets the data as a series of eight 2-byte integer components.

vl, lv Specifies that a following `d`, `i`, `u`, `o`, `x`, or `X` conversion specifier applies to a `vector signed int`, `vector unsigned int`, or `vector bool int` parameter. It consumes one argument and interprets the data as a series of four 4-byte integer components.

vll, llv Specifies that a following `d`, `i`, `u`, `o`, `x`, or `X` conversion specifier applies to a `vector signed long long`, `vector unsigned long long`, or `vector bool long long` parameter. It consumes one argument and interprets the data as a series of two 8-byte integer components.

vL, Lv specifies that a following `a`, `A`, `e`, `E`, `f`, `F`, `g`, or `G` conversion specifier applies to a `vector double` parameter. It consumes one argument and interprets the data as a series of two 8-byte floating point components.

Conversion specifier: Table 30 explains the meaning of the type characters used in the precision argument.

Table 30. Type Characters and their Meanings

Conversion specifier	Argument	Output Format
d, i	Integer	Signed decimal integer.
u	Integer	Unsigned decimal integer.

Table 30. Type Characters and their Meanings (continued)

Conversion specifier	Argument	Output Format
o	Integer	Unsigned octal integer.
x	Integer	Unsigned hexadecimal integer, using abcdef.
X	Integer	Unsigned hexadecimal integer, using ABCDEF.
f, F	Double	Signed value having the form [-]*dddd.dddd*, where *dddd* is one or more decimal digits. The number of digits before the decimal-point depends on the magnitude of the number. The number of digits after the decimal-point is equal to the requested precision. The decimal-point is sensitive to the LC_NUMERIC category of the current locale.
e	Double	Signed value having the form [-]*d.dddd*`e`[*sig n*]*ddd*, where *d* is a single-decimal digit, *dddd* is one or more decimal digits, *ddd* is 2 or more decimal digits, and *sign* is + or -. A double argument representing an infinity or NaN is converted in the style of an f or F conversion specifier.
E	Double	Identical to the `e` format, except that `E` introduces the exponent, not `e`.
g	Double	Signed value output in `f` or `e` format. The `e` format is used only when the exponent of the value is less than -4 or greater than or equal to the *precision*. Trailing zeros are truncated, and the decimal-point appears only if one or more digits follow it. A double argument representing an infinity or NaN is converted in the style of an f or F conversion specifier.
G	Double	Identical to the `g` format, except that `E` introduces the exponent (where appropriate), not *e*.
D(n,p)	Decimal type argument.	Fixed-point value consisting of a series of one or more decimal digits possibly containing a decimal-point.
c C or lc	Character Wide Character	Single character. The argument of wchar_t type is converted to an array of bytes representing a multibyte character as if by call to wctomb().
s S or ls	String Wide String	Characters output up to the first NULL character (`\0`) or until *precision* is reached. The argument is a pointer to an array of wchar_t type. Wide characters from the array are converted to multibyte characters up to and including a terminating NULL wide character. Conversion takes place as if by a call to wcstombs(), with the conversion state described by the mbstate_t object initialized to 0. The result written out will not include the terminating NULL character. If no precision is specified, the array contains a NULL wide character. If a precision is specified, its sets the maximum number of characters written, including shift sequences. A partial multibyte character cannot be written.
n	Pointer to integer	Number of characters successfully output so far to the *stream* or buffer; this value is stored in the integer whose address is given as the argument.
p	Pointer	Pointer to void converted to a sequence of printable characters. Refer to the individual system reference guides for the specific format.

Table 30. Type Characters and their Meanings (continued)

Conversion specifier	Argument	Output Format
a, A	Double _Decimal32 _Decimal64 _Decimal128	A double argument representing a floating-point number is converted to the “[-]0xh.hhhhp±d” format, where there is one hexadecimal digit (non-zero when the argument is a normalized floating-point number; otherwise unspecified) before the decimal-point character and the number of hexadecimal digits after it is equal to the precision. If the precision is missing and FLT_RADIX is a power of 2, then the precision will be sufficient for an exact representation of the value. If the precision is missing and FLT_RADIX is not a power of 2, then the precision will be sufficient to distinguish values of type double, except that trailing zeros may be omitted. If the precision is zero and the “#” flag is not specified, no decimal-point will appear. The letters “abcdef” are used for the a conversion and the letters “ABCDEF” for the A conversion. The A conversion specifier produces a number with letters “X” and “P” instead of letters “x” and “p”. The exponent always contains at least one digit, and only as many more digits as necessary to represent the decimal exponent of 2. If the value is zero, the exponent is zero. A double argument representing an infinity or NaN is converted in the style of an f or F conversion specifier. If precision is zero, results can be different for Hexadecimal floating point format and IEEE floating point format. For Hexadecimal floating point, a decimal point will not appear in the output. For IEEE floating point, a decimal point will appear. If an H, D, or DD modifier is present and the precision is missing, then for a decimal floating type argument, either the f or e style formatting is used based on the following criteria: • f style formatting is used when the quantum is less than or equal to 0 but greater than or equal to -(n+5). The quantum of a number can be determined by calling the quantexp() functions. The number of digits (n) in the digit-sequence includes trailing zeros only and ignores the decimal point. For example: **0.000005** contains 1 digit in the digit sequence, n = 1 **0.0000050** contains 2 digits in the digit sequence, n = 2 **12.30** contains 4 digits in the digit sequence, n = 4 The precision is equal to -quantum. • e style formatting is used when the quantum does not satisfy the f style criteria. The precision is equal to n-1, except if the argument is equal to 0 then the digit-sequence in the exponent-part is equal to the quantum. For example: **0.0000000** produces 0e-07 **-1870** produces -1.87e+03 \If the precision is present and at least as large as the precision of the decimal floating type, the conversion is as if the precision were missing. If the precision is present and less than the precision of the decimal floating type, the input is first rounded according to the current rounding direction to the number of digits specified by the precision . The result is then converted as if the precision were missing. The "quantum" when referring to a finite decimal floating-point number is defined as the "magnitude of a value of one in the rightmost digit position of the significand". For example, pennies and dollars can be represented respectively as 1 * 10 with a quantum of -2 and 0, or, 1*10^-2 and 1*10^0. For more information on the term quantum, see *z/Architecture Principles of Operation*.

Special behavior for XPG4:

- If the %n$ conversion specification is found, the value of the nth *argument* after the *format-string* is converted and output according to the conversion specification. Numbered arguments in the argument list can be referenced from *format-string* as many times as required.
- The *format-string* can contain either form of the conversion specification, that is, % or %n$ but the two forms cannot be mixed within a single *format-string* except that %% can be mixed with the %n$ form. When numbered conversion specifications are used, specifying the 'nth' argument requires that the first to (n-1)th arguments are specified in the *format-string*.

Floating-point and the fprintf family of formatted output functions: The fprintf family functions match `a`, `A`, `e`, `E`, `f`, `F`, `g`, or `G` conversion specifiers to floating-point arguments for which they produce floating-point number substrings in the output stream. The fprintf family functions have been extended to determine the floating-point format, hexadecimal floating-point or IEEE Binary Floating-Point, of types `a`, `A`, `e`, `E`, `f`, `F`, `g`, or `G` by using __isBFP().

The fprintf family functions convert IEEE Binary Floating-Point infinity and NaN argument values to special infinity and NaN floating-point number output sequences.

- The special output sequence for infinity values is a plus or minus sign, then the character sequence INF followed by a white space character (space, tab, or newline), a NULL character (`\0`) or EOF.
- The special output sequence for NaN values is a plus or minus sign, then the character sequence NANS for a signalling NaN or NANQ for a quiet NaN, then a NaN ordinal sequence, and then a a white space character (space, tab, or newline), a NULL character (`\0`) or EOF.

  For binary floating point NANs: A NaN ordinal sequence is a left-parenthesis character, “(”, followed by a digit sequence representing an integer n, where 1 <= n <= INT_MAX-1, followed by a right-parenthesis character, “)”. The integer value, n, is determined by the fraction bits of the NaN argument value as follows:

  1. For a signalling NaN value, NaN fraction bits are reversed (left to right) to produce bits (right to left) of an even integer value, 2*n. Then formatted output functions produce a (signalling) NaN ordinal sequence corresponding to the integer value n.
  2. For a quiet NaN value, NaN fraction bits are reversed (left to right) to produce bits (right to left) of an odd integer value, 2*n-1. Then formatted output functions produce a (quiet) NaN ordinal sequence corresponding to the integer value n.

  For decimal floating point NANs: A NaN ordinal sequence is a left parenthesis character, "(", followed by a decimal digit sequence of up to 6 digits for a _Decimal32 output number, up to 15 digits for a _Decimal64 output value, or up to 33 digits for a _Decimal128 output value, followed by a right parenthesis, ")". If the NaN ordinal sequence is omitted, NaN ordinal sequence "(0)" is assumed. If the NaN ordinal sequence is shorter than 6, 15, or 33 digits, it will be padded on the left with "0" digits so that the length becomes 6, 15, or, 33 digits for _Decimal32, _Decimal64, and _Decimal128 values respectively.

  For decimal floating point numbers, the digits are not reversed, and both odd or even NaN ordinal sequences can be specified for either a Quiet NAN or Signalling NAN.

The C99 standard does not distinguish between the quiet NaN and signaling NaN values. An argument representing a NaN (Not a Number) is to be displayed as [-]nan or [-]nan(n-char-sequence); where the implementation decides the representation. For the `A`, `E`, `F`, and `G` conversion specifiers, NaN values are displayed as the uppercase versions of the aforementioned character string representations. To get this behavior set the environment variable _EDC_C99_NAN to YES.

Some compatibility with NaN sequences output by AIX formatted output functions can be achieved by setting a new environment variable, _AIX_NAN_COMPATIBILITY, which z/OS formatted output functions recognize, to one of the following (string) values:

Value Output function

1 Formatted output functions which produce special NaN output sequences omit the NaN ordinal output sequence (1). This results in output NaN sequences of plus or minus sign followed by NANS or NANQ instead of plus or minus sign followed by NANS(1) or NANQ(1). All other NaN ordinal sequences are explicitly output.

ALL Formatted output functions which produce special NaN output sequences omit the NaN ordinal output sequence for all NaN values. This results in output NaN sequences of plus or minus sign followed by NANS or NANQ instead of plus or minus sign followed by NANS(n) or NANQ(n) for all NaN values.

Note: _AIX_NAN_COMPATIBILITY does not affect the formatting of DFP NAN values. It affects only the formatting of BFP NAN values.

Vector types and the fprintf family of formatted output functions: The fprintf family functions support character, integer, and floating-point vector data types in the output conversion specification. The vector value is displayed in the following general form:

```
Value1Cvalue2C...CValueN
```

where C is a separator character optionally defined in flag characters, and there can be 2, 4, 8, or 16 output values depending on the optional prefix v, vh, hv, vl, lv, vll, llv, vL, or Lv. Each value is formatted according to the conversion specifier.

The sprintf() function is available to C applications in a stand-alone System Programming C (SPC) Environment.

Usage notes

1. FLOAT(HEX) normalizes differently than FLOAT(IEEE). FLOAT(HEX) produces output in 0x0.hhhhhp+/-dd format, not in the 0x1.hhhhhhp+/-dd format.
2. To use IEEE decimal floating-point, the hardware must have the Decimal Floating-Point Facility installed.

Returned value

If successful, fprintf(), printf(), and sprintf() return the number of characters output. The ending NULL character is not counted.

If unsuccessful, they return a negative value.

Example

CELEBF30

```
/* CELEBF30

   This example prints data using &printf. in a variety of
   formats.

 */
#include <stdio.h>

int main(void)
{
   char ch = 'h', *string = "computer";
   int count = 234, hex = 0x10, oct = 010, dec = 10;
   double fp = 251.7366;
   unsigned int a = 12;
   float b = 123.45;
   int c;
   void *d = "a";

   printf("the unsigned int is %u\n\n",a);

   printf("the float number is %g, and %G\n\n",b,b);

   printf("RAY%n\n\n",&c);

   printf("last line prints %d characters\n\n",c);

   printf("Address of d is %p\n\n",d);

   printf("%d   %+d    %06d     %X    %x     %o\n\n",
      count, count, count, count, count, count);

   printf("1234567890123%n4567890123456789\n\n", &count);

   printf("Value of count should be 13; count = %d\n\n", count);

   printf("%10c%5c\n\n", ch, ch);

   printf("%25s\n%25.4s\n\n", string, string);

   printf("%f    %.2f    %e    %E\n\n", fp, fp, fp, fp);

   printf("%i    %i     %i\n\n", hex, oct, dec);
}
```

Output

```
the unsigned int is 12

the float number is 123.45 and 123.45

RAY

last line prints 3 characters

Address of d is DD72F9

234   +234    000234     EA    ea     352

12345678901234567890123456789

Value of count should be 13; count = 13

         h    h
```

```
                 computer
                     comp

251.736600    251.74    2.517366e+02    2.517366E+02

16    8     10
```

CELEBF31

```
/* CELEBF31
   The following example illustrates the use of printf() to print
   fixed-point decimal data types.
   This example works under C only, not C++.
 */
#include <stdio.h>
#include <decimal.h>

decimal(10,2) pd01 = -12.34d;
decimal(12,4) pd02 = 12345678.9876d;
decimal(31,10) pd03 = 123456789013579246801.9876543210d;

int main(void) {
  printf("pd01 %%D(10,2)      = %D(10,2)\n", pd01);
  printf("pd02 %%D( 12 , 4 )  = %D( 12 , 4 )\n", pd02);

  printf("pd01 %%010.2D(10,2) = %010.2D(10,2)\n", pd01);
  printf("pd02 %%20.2D(12,4)  = %20.2D(12,4)\n", pd02);
  printf("\n Give strange result if the specified size is wrong!\n");
  printf("pd03 %%D(15,3)      = %D(15,3)\n\n", pd03);
}
```

Output

```
pd01 %D(10,2)      = -12.34
pd02 %D( 12 , 4 )  = 12345678.9876
pd01 %010.2D(10,2) = -000012.34
pd02 %20.2D(12,4)  =          12345678.98

Give strange result if the specified size is wrong!
pd03 %D(15,3)      = -123456789013.579
```

CELEBF32

```
/* CELEBF32
   This example illustrates the use of sprintf() to format and print
   various data.
 */
#include <stdio.h>

char buffer[200];
int i, j;
double fp;
char *s = "baltimore";
char c;

int main(void)
{
   c = 'l';
   i = 35;
   fp = 1.7320508;

   /* Format and print various data */
   j = sprintf(buffer, "%s\n", s);
   j += sprintf(buffer+j, "%c\n", c);
   j += sprintf(buffer+j, "%d\n", i);
   j += sprintf(buffer+j, "%f\n", fp);
   printf("string:\n%s\ncharacter count = %d\n", buffer, j);
}
```

Output

```
string:
Baltimore
l
35
1.732051

character count = 24
```

CELEBF90

```
/* CELEBF90
   The following example illustrates the use of printf() to print
   vector data types.
 */
#include <stdio.h>

vector signed char s8 =
                {'a','b',' ','d','e','f','g','h',
                 'i','j','k','l','m','!','o','p'};
vector unsigned short u16 = {1,2,3,4,5,6,7,8};
vector signed int s32 = {1, 2, 3, 99};
vector unsigned long long u64 = {1, 2};
vector double d64 = {0.1, 1.2};

int main(void)
{
   printf("s8 = %vc\n", s8);
   printf("s8 = %,vc\n", s8);
   printf("u16 = %vhu\n", u16);
   printf("s32 = %,2lvd\n", s32);
   printf("u64 = %vllu\n", u64);
   printf("d64 = %vLf\n", d64);
}
```

Output

```
s8 = ab defghijklm!op
s8 = a,b, ,d,e,f,g,h,i,j,k,l,m,!,o,p
u16 = 1 2 3 4 5 6 7 8
s32 =  1, 2, 3,99
u64 = 1 2
d64 = 0.100000 1.200000
```

Related information

- See the topic about internationalization of locales and characters sets in *z/OS XL C/C++ Programming Guide*
- See the topic about system programming C (SPC) facilities in *z/OS XL C/C++ Programming Guide*
- “locale.h” on page 40
- “stdio.h” on page 68
- “wchar.h” on page 85
- “fscanf(), scanf(), sscanf() — Read and format data” on page 625
- “__isBFP() — Determine application floating-point format” on page 904
- “localeconv() — Query numeric conventions” on page 982
- “setlocale() — Set locale” on page 1552
- “wcrtomb() — Convert a wide character to a multibyte character” on page 2001

__fpurge() — Discard pending data in a stream

Standards

Standards / Extensions	C or C++	Dependencies
Language Environment	both	None

Format

```
#include <stdio.h>
#include <stdio_ext.h>

void __fpurge(FILE *stream);

#define _OPEN_SYS_UNLOCKEKD 1
#include <stdio.h>
#include <stdio_ext.h>

void __fpurge_unlocked(FILE *stream);
```

General description

The _fpurge() function requests that any pending data in the stream be discarded.

After a call to __fpurge(), if stream is currently reading, any data that has been read from the system but not yet presented to the application will be discarded. Similarly, any data that has been pushed back into the steam with ungetc() will also be discarded.

After a call to __fpurge(), if stream is currently writing, any data that the application has requested to be written, but has not yet been flushed to an output device will be discarded. Any data that is written to the buffer after a call to __fpurge() is destined to be written out, to the block where the discarded data was originally intended to be placed, either explicitly via fflush() or implicity once the buffer is full.

The __fpurge_unlocked() function is equivalent to the __fpurge() function with the exception that it is not thread-safe. This function can be safely used in a multithreaded application if it is called while the invoking thread owns the (FILE *) object, such as after a successful call to either the flockfile() or ftrylockfile() function.

Usage note

The __fpurge() function is allowed to be called on UNIX files only. Calling this function on any other type of file will not cause pending data in the buffer to be discarded. The function will return and result in errno being set to a nonzero value.

Returned value

The _fpurge() function returns no values.

An application wishing to check for error situations should set errno to 0, then call __fpurge(), and then check errno. If errno is nonzero, assume that an error has occurred.

Error Code
Description

EBADF
The stream specified by *stream* is not valid.

Example

CELEBF86

```
/* CELEBF86

   This example purges pending data in the I/O buffer.

*/

#include <stdio.h>
#include <errno.h>
#include <stdio_ext.h>

void main(int argc,char** argv){
   char *bufflush="data will be written into file";
   char *bufpurge="data will be not written into file";
   FILE *fp=fopen("./myfile","a");
   if(NULL==fp){
      perror("failed to open file\n");
      exit(0);
   }
   fwrite(bufflush,strlen(bufflush),1,fp);
   fflush(fp);
   fwrite(bufpurge,strlen(bufpurge),1,fp);
   errno=0;
   __fpurge(fp);
   if(errno!=0)
      perror("call __fpurge() failed\n");
   fclose(fp);
}
```

Output

```
The file "myfile" contains the text - data will be written into file
```

Related information

- “stdio.h” on page 68
- “stdio_ext.h” on page 70
- “_flushlbf() — Flush all open line-buffered files” on page 550
- “fflush() — Write buffer to file” on page 523
- “ungetc() — Push character onto input stream” on page 1958
- “ungetwc() — Push a wide character onto a stream” on page 1961

fputc() — Write a character

Standards

Standards / Extensions	C or C++	Dependencies
ISO C POSIX.1 XPG4 XPG4.2 C99 Single UNIX Specification, Version 3 Language environment	both	

Format

```
#include <stdio.h>

int fputc(int c, FILE *stream);

#define _OPEN_SYS_UNLOCKED_EXT 1
#include <stdio.h>

int fputc_unlocked(int c, FILE *stream);
```

General description

Converts *c* to an `unsigned char` and then writes *c* to the output stream pointed to by *stream* at the current position and advances the file position appropriately. The fputc() function is identical to *putc* but is always a function, because it is not available as a macro.

If the stream is opened with one of the append modes, the character is appended to the end of the stream regardless of the current file position.

The fputc() function is not supported for files opened with `type=record` or `type=blocked`.

fputc() has the same restriction as any write operation for a read immediately following a write, or a write immediately following a read. Between a write and a subsequent read, there must be an intervening flush or reposition. Between a read and a subsequent write, there must also be an intervening flush or reposition unless an EOF has been reached.

fputc_unlocked() is functionally equivalent to fputc() with the exception that it is not thread-safe. This function can safely be used in a multithreaded application if and only if it is called while the invoking thread owns the (FILE*) object, as is the case after a successful call to either the flockfile() or ftrylockfile() function.

Returned value

If successful, fputc() returns the character written.

If unsuccessful, fputc() returns EOF.

Example

CELEBF34

```
/* CELEBF34

   This example writes the contents of buffer to a file called
   myfile.dat.
   Because the output occurs as a side effect within the second
   expression of the for statement, the statement body is null.

 */
#include <stdio.h>
#define NUM_ALPHA  26

int main(void)
{
  FILE * stream;
  int i;
  int ch;
  char buffer[NUM_ALPHA + 1] = "abcdefghijklmnopqrstuvwxyz";

  if (( stream = fopen("myfile.dat", "w"))!= NULL )
  {
    /* Put buffer into file */
    for ( i = 0; ( i < sizeof(buffer) ) &&
           ((ch = fputc( buffer[i], stream)) != EOF ); ++i );
    fclose( stream );
  }
  else
    printf( "Error opening myfile.dat\n" );
}
```

Related information

- “stdio.h” on page 68
- “fgetc() — Read a character” on page 526
- “putc(), putchar() — Write a character” on page 1347

fputs() — Write a string

Standards

Standards / Extensions	C or C++	Dependencies
ISO C POSIX.1 XPG4 XPG4.2 C99 Single UNIX Specification, Version 3 Language environment	both	

Format

```
#include <stdio.h>

int fputs(const char * __restrict__string, FILE * __restrict__stream);

#define _OPEN_SYS_UNLOCKED_EXT 1
#include <stdio.h>

int fputs_unlocked(const char * __restrict__string, FILE * __restrict__stream);
```

General description

Writes the string pointed to by *string* to the output stream pointed to by *stream*. It does not write the terminating `\0` at the end of the string.

For a text file, truncation may occur if the record is too long. *Truncation* means that excess characters are discarded after the record is full, up to a control character that ends the line (\n). Characters after the \n start at the next record. For more information, see the section on “Truncation” in *z/OS XL C/C++ Programming Guide*.

fputs() is not supported for files opened with `type=record` or `type=blocked`.

fputs() has the same restriction as any write operation for a read immediately following a write or a write immediately following a read. Between a write and a subsequent read, there must be an intervening flush or reposition. Between a read and a subsequent write, there must also be an intervening flush or reposition unless an EOF has been reached.

fputs_unlocked() is functionally equivalent to fputs() with the exception that it is not thread-safe. This function can safely be used in a multithreaded application if and only if it is called while the invoking thread owns the (FILE*) object, as is the case after a successful call to either the flockfile() or ftrylockfile() function.

Returned value

If successful, fputs() returns the number of bytes written.

If unsuccessful, fputs() returns EOF.

Example

CELEBF35

```
/* CELEBF35

   This example writes a string to a stream.

 */
#include <stdio.h>
#define NUM_ALPHA  26

int main(void)
{
  FILE * stream;
  int num;

  /* Do not forget that the '\0' char occupies one character */
  static char buffer[NUM_ALPHA + 1] = "abcdefghijklmnopqrstuvwxyz";

  if ((stream = fopen("myfile.dat", "w")) != NULL )
  {
     /* Put buffer into file */
     if ( (num = fputs( buffer, stream )) != EOF )
     {
       /* Note that fputs() does not copy the \0 character */
       printf( "Total number of characters written to file = %i\n", num );
       fclose( stream );
     }
     else   /* fputs failed */
       printf( "fputs failed" );
  }
  else
     printf( "Error opening myfile.dat" );
}
```

Related information

- “stdio.h” on page 68
- “fgets() — Read a string from a stream” on page 530
- “gets() — Read a string” on page 769
- “puts() — Write a string” on page 1353

fputwc() — Output a wide-character

Standards

Standards / Extensions	C or C++	Dependencies
XPG4 XPG4.2 Single UNIX Specification, Version 3 Language Environment	both	

Format

Non-XPG4:

```
#include <stdio.h>
#include <wchar.h>

wint_t fputwc(wchar_t wc, FILE *stream);

#define _OPEN_SYS_UNLOCKED_EXT 1
#include <wchar.h>

wint_t fputwc_unlocked(wchar_t wc, FILE *stream);
```

XPG4:

```
#define _XOPEN_SOURCE
#include <stdio.h>
#include <wchar.h>

wint_t fputwc(wint_t wc, FILE *stream);

#define _OPEN_SYS_UNLOCKED_EXT 1
#include <wchar.h>

wint_t fputwc_unlocked(wchar_t wc, FILE *stream);
```

XPG4 and MSE:

```
#define _XOPEN_SOURCE
#define _MSE_PROTOS
#include <stdio.h>
#include <wchar.h>

wint_t fputwc(wchar_t wc, FILE *stream);

#define _OPEN_SYS_UNLOCKED_EXT 1
#include <wchar.h>

wint_t fputwc_unlocked(wchar_t wc, FILE *stream);
```

General description

Converts the wide character specified by *wc* to a multibyte character and writes it to the output stream pointed to by *stream*, at the position indicated by the associated file position indicator for the stream (if defined), and advances the indicator appropriately. If the file cannot support positioning requests or if the stream was opened with append mode, the character is appended to the output stream.

The behavior of this wide-character function is affected by the LC_CTYPE category of the current locale. Using non-wide-character functions with fputwc() results in undefined behavior.

fputwc() has the same restriction as any write operation for a read immediately following a write or a write immediately following a read. Between a write and a subsequent read, there must be an intervening flush or reposition. Between a read and a subsequent write, there must also be an intervening flush or reposition unless an EOF has been reached.

fputwc_unlocked() is functionally equivalent to fputwc() with the exception that it is not thread-safe. This function can safely be used in a multithreaded application if and only if it is called while the invoking thread owns the (FILE*) object, as is the case after a successful call to either the flockfile() or ftrylockfile() function.

Special behavior for XPG4: If you define any feature test macro specifying XPG4 behavior before the statement in your program source file to include the wchar header, then the compiler assumes that your program is using the XPG4 variety of the fputwc() function, unless you also define the _MSE_PROTOS feature test macro. Please see Table 2 on page 4 for a list of XPG4 and other feature test macros.

The prototype for the XPG4 variety of the fputwc() function is:

```
wint_t fputwc(wint_t wc, FILE *stream);
```

The difference between this variety and the MSE variety of the fputwc() function is that the first parameter has type wint_t rather than type wchar_t.

Returned value

If successful, fputwc() returns the wide character written.

If a write error occurs, the error indicator for the stream is set and WEOF is returned. If an encoding error occurs during conversion from wide character to a multibyte character, the value of the macro EILSEQ is stored in errno and WEOF is returned.

Example

CELEBF36

```
/* CELEBF36 */
#include <stdio.h>
#include <stdlib.h>
#include <wchar.h>
#include <errno.h>

int main(void)
{
   FILE    *stream;
   wchar_t *wcs = L"This test string should not cause a WEOF condition";
   int     i;
   int     rc;

   if ((stream = fopen("myfile.dat", "w")) == NULL) {
      printf("Unable to open file.\n");
      exit(1);
   }

   for (i=0; wcs[i] != L'\0'; i++) {
      errno = 0;
      if ((rc = fputwc(wcs[i], stream)) == WEOF) {
         printf("Unable to fputwc() the wide character.\n");
         printf("wcs[%d] = 0x%lx\n", i, wcs[i]);
         if (errno == EILSEQ)
            printf("An invalid wide character was encountered.\n");
         exit(1);
      }
   }

   fclose(stream);
}
```

Related information

- "stdio.h" on page 68
- "wchar.h" on page 85

fputws() — Output a wide-character string

Standards

Standards / Extensions	C or C++	Dependencies
XPG4 XPG4.2 Single UNIX Specification, Version 3 Language Environment	both	

Format

```
#include <wchar.h>

int fputws(const wchar_t * __restrict__wcs, FILE * __restrict__stream);

#define _OPEN_SYS_UNLOCKED_EXT 1
#include <wchar.h>

int fputws_unlocked(const wchar_t * __restrict__wcs, FILE * __restrict__stream);
```

General description

Converts the wide-character string pointed to by *wcs* to a multibyte character string and writes it to the stream pointed to by *stream*, as a multibyte character string. The terminating NULL byte is not written.

The behavior of this wide-character function is affected by the LC_CTYPE category of the current locale. Using non-wide-character functions with fputws() results in undefined behavior.

fputws() has the same restriction as any write operation for a read immediately following a write or a write immediately following a read. Between a write and a subsequent read, there must be an intervening flush or reposition. Between a read and a subsequent write, there must also be an intervening flush or reposition unless an EOF has been reached.

fputws_unlocked() is functionally equivalent to fputws() with the exception that it is not thread-safe. This function can safely be used in a multithreaded application if and only if it is called while the invoking thread owns the (FILE*) object, as is the case after a successful call to either the flockfile() or ftrylockfile() function.

Returned value

If successful, fputws() returns a nonnegative value.

If a *stream* error occurs, fputws() returns -1 and the error indicator for the stream is set.

If an *encoding* error occurs, fputws() returns -1 and the value of the macro EILSEQ is stored in errno. An encoding error is one that occurs when converting a wide character to a multibyte character.

Example

CELEBF37

```
/* CELEBF37 */
#include <errno.h>
#include <stdio.h>
#include <stdlib.h>
#include <wchar.h>

int main(void)
{
   FILE    *stream;
   wchar_t *wcs = L"This test string should not return -1";
   int     rc;

   if ((stream = fopen("myfile.dat", "w")) == NULL) {
      printf("Unable to open file.\n");
      exit(1);
   }

   errno = 0;
   rc = fputws(wcs, stream);

   if (rc == EOF) {
      printf("Unable to complete fputws() function.\n");
      if (errno == EILSEQ)
         printf("An invalid wide character was encountered.\n");
      exit(1);
   }

   fclose(stream);
}
```

Related information

- “stdio.h” on page 68
- “wchar.h” on page 85

fread() — Read items

Standards

Standards / Extensions	C or C++	Dependencies
ISO C POSIX.1 XPG4 XPG4.2 C99 Single UNIX Specification, Version 3 Language environment	both	

Format

```
#include <stdio.h>

size_t fread(void * __restrict__buffer, size_t size, size_t count, FILE * __restrict__stream);

#define _OPEN_SYS_UNLOCKED_EXT 1
#include <stdio.h>

size_t fread_unlocked(void * __restrict__buffer, size_t size,
                      size_t count, FILE * __restrict__stream);
```

General description

Reads up to *count* items of *size* length from the input stream pointed to by *stream* and stores them in the given *buffer*. The file position indicator advances by the number of bytes read.

If there is an error during the read operation, the file position indicator is undefined. If a partial element is read, the element's value is undefined.

When you are using fread() for record I/O, set *size* to 1 and *count* to the maximum expected length of the record, to obtain the number of bytes. Only one record is read, regardless of *count*, when using record I/O.

When you are using fread() for blocked I/O, set *size* to 1 and *count* to the maximum expected length of the block, to obtain the number of bytes. Only one block is read, regardless of *count*, when using blocked I/O.

fread() has the same restriction as any read operation for a read immediately following a write or a write immediately following a read. Between a write and a subsequent read, there must be an intervening flush or reposition. Between a read and a subsequent write, there must also be an intervening flush or reposition unless an EOF has been reached.

fread_unlocked() is functionally equivalent to fread() with the exception that it is not thread-safe. This function can safely be used in a multithreaded application if and only if it is called while the invoking thread owns the (FILE*) object, as is the case after a successful call to either the flockfile() or ftrylockfile() function.

Returned value

fread() returns the number of complete items successfully read.

If *size* or *count* is 0, fread() returns 0, and the contents of the array and the state of the stream remain unchanged. For record I/O and blocked I/O, it is possible that the number of complete items can be less than *count*. However, this result does not necessarily indicate that an error has occurred.

If the Physical File System does not support simple reads from directories, fread() will return 0 if it is used for a directory. Users should use Opendir() and readdir() instead.

The ferror() and feof() functions are used to distinguish between a read error and an EOF. Note that EOF is only reached when an attempt is made to read “past” the last byte of data. Reading up to and including the last byte of data does *not* turn on the EOF indicator.

Example

CELEBF38

```
/* CELEBF38

   This example attempts to read NUM_ALPHA characters from the
   file myfile.dat.
   If there are any errors with either &fread. or &fopen., a
   message is printed.

 */
#include <stdio.h>
#define NUM_ALPHA  26

int main(void)
{
  FILE * stream;
  int num;       /* number of characters read from stream */

  /* Do not forget that the '\0' char occupies one character too! */
  char buffer[NUM_ALPHA + 1];
  buffer[NUM_ALPHA] = '\0';

  if (( stream = fopen("myfile.dat", "r"))!= NULL )
  {
    num = fread( buffer, sizeof( char ), NUM_ALPHA, stream );
    if (num == NUM_ALPHA) {  /* fread success */
      printf( "Number of characters read = %i\n", num );
      printf( "buffer = %s\n", buffer );
      fclose( stream );
    }
    else {  /* fread failed */
      if ( ferror(stream) )         /* possibility 1 */
        printf( "Error reading myfile.dat" );
      else if ( feof(stream)) {     /* possibility 2 */
        printf( "EOF found\n" );
        printf( "Number of characters read %d\n", num );
        printf( "buffer = %.*s\n", num, buffer);
      }
    }
  }
  else
    printf( "Error opening myfile.dat" );
}
```

Related information

- “stdio.h” on page 68
- “fopen() — Open a file” on page 565
- “freopen() — Redirect an open file” on page 620
- “fwrite() — Write items” on page 676

__freadable() — Determine if a stream is open for reading

Standards

Standards / Extensions	C or C++	Dependencies
Language Environment	both	

Format

```
#include <stdio.h>
#include <stdio_ext.h>

int __freadable(FILE *stream);
```

General description

The __freadable() function determines if the specified stream has been opened for reading.

Returned value

The __freadable() function returns nonzero if the stream is opened for reading. Otherwise, the __freadable() function returns 0. If an error has occurred, __freadable() function returns 0 and sets errno to nonzero.

An application wishing to check for error situations should set errno to 0, then call __freadable(), and then check errno. If errno is nonzero, assume that an error has occurred.

Error Code
    Description

EBADF
    The stream specified by *stream* is not valid.

Example

CELEBF89

```
/* CELEBF89

   This example writes and reads data to a file while querying the
   stream for information about data in the I/O buffer.

*/

#include <stdio.h>
#include <stdio_ext.h>

void main() {
   FILE *f;
   char filename[FILENAME_MAX] = "myfile.dat";
   char data[128] = "There are 34 bytes in this buffer\n";
   int datalen = strlen(data);
   size_t n = 0;

   f = fopen(filename,"wb+");
   if (f == NULL) {
      perror("fopen() failed\n");
      return;
   }

   if (__fwritable(f)) printf("Writing is allowed on the open stream\n");
   if (__freadable(f)) printf("Reading is allowed on the open stream\n");

   n = fputs(data,f);
   if (n == EOF) {
      perror("fputs() failed\n");
      return;
   }
   n = __fpending(f);
   printf("There are %d bytes in the buffer pending to be written\n", n);

   if (__fwriting(f)) printf("The last operation on the stream was a write\n");

   rewind(f);

   n = fgetc(f);

   n = __freadahead(f);
   printf("There are %d bytes remaining to be read from the buffer\n", n);

   if (__freading(f)) printf("The last operation on the stream was a read\n");

   return;
}
```

Output

```
Writing is allowed on the open stream
Reading is allowed on the open stream
There are 34 bytes in the buffer pending to be written
The last operation on the stream was a write
There are 33 bytes remaining to be read from the buffer
The last operation on the stream was a read
```

Related information

- “stdio.h” on page 68
- “stdio_ext.h” on page 70
- “fopen() — Open a file” on page 565
- “freopen() — Redirect an open file” on page 620
- “__freading() — Determine if last operation on stream is a read operation” on page 616
- “__fwritable() — Determine if a stream is open for writing” on page 674
- “__fwriting() — Determine if last operation on stream is a write operation” on page 677

__freadahead() — Retrieve number of bytes remaining in input buffer

Standards

Standards / Extensions	C or C++	Dependencies
Language Environment	both	

Format

```
#include <stdio.h>
#include <stdio_ext.h>

size_t __freadahead(FILE *stream);

#define _OPEN_SYS_UNLOCKED 1
#include <stdio.h>
#include <stdio_ext.h>

size_t __freadahead_unlocked(FILE *stream);
```

General description

The __freadahead() function retrieves the number of bytes remaining to be read in the input buffer that is associated with the specified binary stream. When the stream is opened for text processing, __fpending retrieves the number of characters pending to be written.

The __freadahead_unlocked() function is equivalent to the __freadahead() function with the exception that it is not thread-safe. This function can be safely used in a multithreaded application if it is called while the invoking thread owns the (FILE *) object, such as after a successful call to either the flockfile() or ftrylockfile() function.

Usage notes

1. If __freadahead() is called when the stream is currently writing, __freadahead() returns 0 and sets errno to a nonzero value.
2. If __freadahead() is called on a data set opened `type=record`, __freadahead() returns 0 and sets errno to a nonzero value.
3. For text data sets, any newline characters representing record boundaries will be included in the returned value.

Returned value

The __freadahead() functions return the number of bytes or characters remaining to be read in the current buffer, depending on the type of stream. Otherwise, the __freadahead() functions return 0. If an error has occurred, __freadahead() functions return 0 and set errno to nonzero.

When the stream is wide-oriented text, the __freadahead() function returns a value measured in wide characters.

An application wishing to check for error situations should set errno to 0, then call __freadahead(), and then check errno. If errno is nonzero, assume that an error has occurred.

Error Code
Description

EBADF
The stream specified by *stream* is not valid.

Example

CELEBF89

```
/* CELEBF89

   This example writes and reads data to a file while querying the
   stream for information about data in the I/O buffer.

*/

#include <stdio.h>
#include <stdio_ext.h>

void main() {
   FILE *f;
   char filename[FILENAME_MAX] = "myfile.dat";
   char data[128] = "There are 34 bytes in this buffer\n";
   int datalen = strlen(data);
   size_t n = 0;

   f = fopen(filename,"wb+");
   if (f == NULL) {
      perror("fopen() failed\n");
      return;
   }

   if (__fwritable(f)) printf("Writing is allowed on the open stream\n");
   if (__freadable(f)) printf("Reading is allowed on the open stream\n");

   n = fputs(data,f);
   if (n == EOF) {
      perror("fputs() failed\n");
      return;
   }

   n = __fpending(f);
   printf("There are %d bytes in the buffer pending to be written\n", n);

   if (__fwriting(f)) printf("The last operation on the stream was a write\n");

   rewind(f);

   n = fgetc(f);

   n = __freadahead(f);
   printf("There are %d bytes remaining to be read from the buffer\n", n);

   if (__freading(f)) printf("The last operation on the stream was a read\n");

   return;
}
```

Output

```
Writing is allowed on the open stream
Reading is allowed on the open stream
There are 34 bytes in the buffer pending to be written
The last operation on the stream was a write
There are 33 bytes remaining to be read from the buffer
The last operation on the stream was a read
```

Related information

- "stdio.h" on page 68
- "stdio_ext.h" on page 70
- "__fpending() — Retrieve number of bytes pending for write" on page 586

__freading() — Determine if last operation on stream is a read operation

Standards

Standards / Extensions	C or C++	Dependencies
Language Environment	both	

Format

```
#include <stdio.h>
#include <stdio_ext.h>

int __freading(FILE *stream);
```

```
#define _OPEN_SYS_UNLOCKED 1
#include <stdio.h>
#include <stdio_ext.h>

int __freading_unlocked(FILE *stream);
```

General description

The __freading() function determines if the last operation on the specified stream is a read operation or if the specified stream is open for read-only.

The __freading_unlocked() function is equivalent to the __freading() function with the exception that it is not thread-safe. This function can be safely used in a multithreaded application if it is called while the invoking thread owns the (FILE *) object, such as after a successful call to either the flockfile() or ftrylockfile() function.

Returned value

The __freading() functions return nonzero when the last operation is a read operation or the stream is open for read-only. Otherwise, the __freading() functions return 0. If an error has occurred, __freading() functions return 0 and set errno to nonzero.

An application wishing to check for error situations should set errno to 0, then call __freading(), and then check errno. If errno is nonzero, assume that an error has occurred.

Error Code
Description

EBADF
The stream specified by *stream* is not valid.

Example

CELEBF89

```
/* CELEBF89

   This example writes and reads data to a file while querying the
   stream for information about data in the I/O buffer.

*/

#include <stdio.h>
#include <stdio_ext.h>

void main() {
   FILE *f;
   char filename[FILENAME_MAX] = "myfile.dat";
   char data[128] = "There are 34 bytes in this buffer\n";
   int datalen = strlen(data);
   size_t n = 0;

   f = fopen(filename,"wb+");
   if (f == NULL) {
      perror("fopen() failed\n");
      return;
   }

   if (__fwritable(f)) printf("Writing is allowed on the open stream\n");
   if (__freadable(f)) printf("Reading is allowed on the open stream\n");

   n = fputs(data,f);
   if (n == EOF) {
      perror("fputs() failed\n");
      return;
   }

   n = __fpending(f);
   printf("There are %d bytes in the buffer pending to be written\n", n);

   if (__fwriting(f)) printf("The last operation on the stream was a write\n");

   rewind(f);

   n = fgetc(f);

   n = __freadahead(f);
   printf("There are %d bytes remaining to be read from the buffer\n", n);

   if (__freading(f)) printf("The last operation on the stream was a read\n");

   return;
}
```

Output

```
Writing is allowed on the open stream
Reading is allowed on the open stream
There are 34 bytes in the buffer pending to be written
The last operation on the stream was a write
There are 33 bytes remaining to be read from the buffer
The last operation on the stream was a read
```

Related information

- "stdio.h" on page 68
- "stdio_ext.h" on page 70
- "fopen() — Open a file" on page 565
- "freopen() — Redirect an open file" on page 620
- "__freadable() — Determine if a stream is open for reading" on page 612
- "__fwritable() — Determine if a stream is open for writing" on page 674
- "__fwriting() — Determine if last operation on stream is a write operation" on page 677

free() — Free a block of storage

Standards

Standards / Extensions	C or C++	Dependencies
ISO C POSIX.1 XPG4 XPG4.2 C99 Single UNIX Specification, Version 3	both	

Format

```
#include <stdlib.h>

void free(void *ptr);
```

General description

Frees a block of storage pointed to by *ptr*. The *ptr* variable points to a block previously reserved with a call to calloc(), malloc(), realloc(), or strdup(). The number of bytes freed is the number of bytes specified when you reserved (or reallocated, in the case of realloc()), the block of storage. If *ptr* is NULL, free() simply returns without freeing anything. Since *ptr* is passed by value free() will not set *ptr* to NULL after freeing the memory to which it points.

This function is also available to C applications in a stand-alone System Programming C (SPC) Environment.

Note: Attempting to free a block of storage not allocated with calloc(), malloc(), realloc(), strdup(), or previously freed storage, can affect the subsequent reserving of storage and lead to an abend.

Special behavior for C++: Under C++, you cannot use free() with an item that was allocated using the C++ new keyword.

Returned value

free() returns no values.

Example

```
/* This example illustrates the use of calloc() to allocate storage for x
   array elements and then calls free() to free them.
 */
#include <stdio.h>
#include <stdlib.h>

int main(void)
{
  long * array;    /* start of the array */
  long * index;    /* index variable     */
  int    i;        /* index variable     */
  int  num;        /* number of entries of the array */

  printf( "Enter the size of the array\n" );
  scanf( "%i", &num );

  /* allocate num entries */
  if ( (index = array = (long *)calloc( num, sizeof( long ))) != NULL )
  {
⋮
   /*  do something with the array  */
    free( array );                        /* deallocates array  */
  }
  else
  { /* Out of storage */
    printf( "Error: out of storage\n" );
    abort();
  }
}
```

Related information

- “Using the System Programming C Facilities” in *z/OS XL C/C++ Programming Guide*
- “spc.h” on page 64
- “stdlib.h” on page 70

- “calloc() — Reserve and initialize storage” on page 231
- “malloc() — Reserve storage block” on page 1031
- “realloc() — Change reserved storage block size” on page 1391

freeaddrinfo() — Free addrinfo storage

Standards

Standards / Extensions	C or C++	Dependencies
RFC2553 Single UNIX Specification, Version 3	both	z/OS V1R4

Format

```
#define _OPEN_SYS_SOCK_IPV6
#include <sys/socket.h>
#include <netdb.h>

void *freeaddrinfo(struct addrinfo *ai);
```

SUSV3

```
#define _POSIX_C_SOURCE 200112L
#include <sys/socket.h>
#include <netdb.h>

void *freeaddrinfo(struct addrinfo *ai);
```

General description

The freeaddrinfo() function frees one or more addrinfo structures returned by getaddrinfo(), along with any additional storage associated with those structures. If the ai_next field of the structure is not null, the entire list of structures is freed.

Returned value

No return value is defined.

Related information

- “connect() — Connect a socket” on page 308
- “gai_strerror() — Address and name information error description” on page 681
- “getaddrinfo() — Get address information” on page 684
- “socket() — Create a socket” on page 1688
- “netdb.h” on page 49
- “sys/socket.h” on page 74

freopen() — Redirect an open file

Standards

Standards / Extensions	C or C++	Dependencies
ISO C POSIX.1 XPG4 XPG4.2 C99 Single UNIX Specification, Version 3	both	

Format

```
#include <stdio.h>

FILE *freopen(const char *__restrict__filename, const char *__restrict__mode,
              FILE *__restrict__stream);
```

General description

Closes the file currently associated with *stream* and pointed to by *stream*, opens the file specified by the *filename*, and then associates the stream with it.

The freopen() function opens the new file with the type of access requested by the *mode* argument. The *mode* argument is used as in the fopen() function. See "fopen() — Open a file" on page 565 for a description of the *mode* parameter.

You can also use the freopen() function to redirect the standard stream files `stdin`, `stdout`, and `stderr` to files that you specify. The file pointer input to the freopen() function must point to a valid open file. If the file pointer has been closed, the behavior is undefined.

You could use the following freopen() call to redirect `stdout` to a memory file `a.b`:

```
freopen("a.b","wb,type=memory",stdout);
```

If *filename* is an empty string, freopen() closes the file and reuses the original file name. For details on how the file name and open mode is interpreted, see *z/OS XL C/C++ Programming Guide*.

A standard stream can be opened by default to a type of file not available to a general fopen(). This is true for standard streams under CICS, and also true for the default stderr, when running a non-POSIX Language Environment application.

The following statement uses freopen() to have `stdin` use binary mode instead of text mode:

```
fp = freopen("", "rb", stdin);
```

You can use the same empty string method to change the mode from binary back to text. This method is not allowed for:

- The default CICS data queues used by the standard streams under CICS
- The Language Environment Message File (MSGFILE), which is the default for `stderr`
- z/OS UNIX files.
- -CELQPIPI MSGRTN file, which is the default for stderr if MSGRTN service routine is specified.

Note: Using the empty string method is included in the SAA C definition, but not in the ANSI C standard.

Large file support for z/OS UNIX files: Large z/OS UNIX files are supported automatically for AMODE 64 C/C++ applications. AMODE 31 C/C++ applications must be compiled with the option LANGLVL(LONGLONG) and define the _LARGE_FILES feature test macro before any headers are included to enable this function to operate on z/OS UNIX files that are larger than 2 GB in size. File size and offset fields are enlarged to 63 bits in width. Therefore, any other function operating on the file is required to define the _LARGE_FILES feature test macro as well.

Returned value

If successful, freopen() returns the value of *stream*, the same value that was passed to it, and clears both the error and EOF indicators associated with the stream.

A failed attempt to close the original file is ignored.

If an error occurs in reopening the requested file, freopen() closes the original file, and returns a NULL pointer value.

Special behavior for large z/OS UNIX files: The following is the possible value of errno:

Error Code
Description

EOVERFLOW
The named file is a regular file and the size of the file cannot be represented correctly in an object of type off_t.

Example

This example illustrates the z/OS XL C extension that allows you to change characteristics of a file by reopening it.

```
#include <stdio.h>

int main(void)
{
   FILE *stream, *stream2;

   stream = fopen("myfile.dat","r");
   stream2 = freopen("", "w+", stream);
}
```

This example closes the `stream` data stream and reassigns its stream pointer:

```
#include <stdio.h>

int main(void)
{
   FILE *stream, *stream2;

   stream = fopen("myfile.dat","r");
   stream2 = freopen("myfile2.dat", "w+", stream);
}
```

Note: `stream` and `stream2` will have the same value.

Related information

- "stdio.h" on page 68
- "fclose() — Close file" on page 473
- "fopen() — Open a file" on page 565

frexp(), frexpf(), frexpl() — Extract mantissa and exponent of the floating-point value

Standards

Standards / Extensions	C or C++	Dependencies
ISO C POSIX.1 XPG4 XPG4.2 ISO/ANSI C++ C99 Single UNIX Specification, Version 3 C++ TR1 C99	both	

Format

```
#include <math.h>

double frexp(double x, int *expptr);
float frexp(float x, int *expptr);              /* C++ only */
long double frexp(long double x, int *expptr);   /* C++ only */
float frexpf(float x, int *expptr);
long double frexpl(long double x, int *expptr);
```

General description

Breaks down the floating-point value *x* into a component *m* for the normalized fraction component and another term *n* for the exponent, such that the absolute value of *m* is greater than or equal to 0.5 and less than 1.0 or equal to 0, and $x = m * 2^n$. The function stores the integer exponent *n* at the location to which *expptr* points.

Note: These functions work in both IEEE Binary Floating-Point and hexadecimal floating-point formats. See “IEEE binary floating-point” on page 94 for more information about IEEE Binary Floating-Point.

Restriction: The frexpf() and frexpl() functions do not support the _FP_MODE_VARIABLE feature test macro.

Returned value

Returns the normalized fraction *m*. If *x* is 0, the function returns 0 for both the fraction and exponent. The fraction has the same sign as the argument *x*. The result of the function cannot have a range error.

Example

CELEBF41

```
/* CELEBF41

   This example decomposes the floating-point value of x, 16.4, into its
   normalized fraction 0.5125, and its exponent 5.
   It stores the mantissa in y and the exponent in n.

 */

#include <math.h>
```

```
#include <stdio.h>

int main(void)
{
   double x, m;
   int n;

   x = 16.4;
   m = frexp(x, &n);

   printf("The fraction is %lf and the exponent is %d\n", m, n);
}
```

Output

```
The mantissa is 0.512500 and the exponent is 5
```

Related information

- "math.h" on page 44
- "ldexp(), ldexpf(), ldexpl() — Multiply by a power of two" on page 941
- "modf(), modff(), modfl() — Extract fractional and integral parts of floating-point value" on page 1086

frexpd32(), frexpd64(), frexpd128() — Extract mantissa and exponent of the decimal floating-point value

Standards

Standards / Extensions	C or C++	Dependencies
C/C++ DFP	both	z/OS V1.8

Format

```
#define __STDC_WANT_DEC_FP__
#include <math.h>

_Decimal32  frexpd32(_Decimal32 x, int *expptr);
_Decimal64  frexpd64(_Decimal64 x, int *expptr);
_Decimal128 frexpd128(_Decimal128 x, int *expptr);

_Decimal32  frexp(_Decimal32 x, int *expptr);   /* C++ only */
_Decimal64  frexp(_Decimal64 x, int *expptr);   /* C++ only */
_Decimal128 frexp(_Decimal128 x, int *expptr);  /* C++ only */
```

General description

Breaks down the decimal floating-point value *x* into a component *m* for the normalized fraction component and another term *n* for the exponent, such that the absolute value of *m* is greater than or equal to 0.1 and less than 1.0 or equal to 0, and $x = m * 10^{n}$. The function stores the integer exponent *n* at the location to which *expptr* points.

Notes:

1. To use IEEE decimal floating-point, the hardware must have the Decimal Floating-Point Facility installed.
2. These functions work in IEEE decimal floating-point format. See "IEEE Decimal Floating-Point" for more information.

Returned value

Returns the normalized fraction m. If *x* is 0, the function returns 0 for both the fraction and exponent. The fraction has the same sign as the argument *x*. The result of the function cannot have a range error.

Example

```
/* CELEBF81

   This example illustrates the frexpd64() function.

*/

#define  __STDC_WANT_DEC_FP__
#include <math.h>
#include <stdio.h>

int main(void)
{
   _Decimal64 x, m;
   int n;

   x = 164.5DD;
   m = frexpd64(x, &n);

   printf("The fraction of %Df is %Df and the exponent is %d\n",
          x, m, n);
}
```

Related information

- “math.h” on page 44
- “frexp(), frexpf(), frexpl() — Extract mantissa and exponent of the floating-point value” on page 623
- “ilogbd32(), ilogbd64(), ilogbd128() — Integer unbiased exponent” on page 837
- “ldexpd32(), ldexpd64(), ldexpd128() — Multiply by a power of ten” on page 942

fscanf(), scanf(), sscanf() — Read and format data

Standards

Standards / Extensions	C or C++	Dependencies
ISO C POSIX.1 XPG4 XPG4.2 Single UNIX Specification, Version 3 C/C++ DFP Language environment	both	z/OS V1R8

Format

```
#include <stdio.h>

int fscanf(FILE *__restrict__stream, const char *__restrict__format-string, ...);
int scanf(const char *__restrict__format-string, ...);
int sscanf(const char *__restrict__buffer, const char *__restrict__format, ...);

#define _OPEN_SYS_UNLOCKED_EXT 1
#include <stdio.h>
```

```
int fscanf_unlocked(FILE *__restrict__stream,
                    const char *__restrict__format-string, ...);
int scanf_unlocked(const char *__restrict__format-string, ...);
```

General description

These three related functions are referred to as the fscanf family.

Reads data from the current position of the specified *stream* into the locations given by the entries in the argument list, if any. The argument list, if it exists, follows the format string. The fscanf() function cannot be used for a file opened with `type=record` or `type=blocked`.

The scanf() function reads data from the standard input stream `stdin` into the locations given by each entry in the argument list. The argument list, if it exists, follows the format string. scanf() *cannot* be used if `stdin` has been reopened as a `type=record` or `type=blocked` file.

The sscanf() function reads data from *buffer* into the locations given by *argument-list*. Reaching the end of the string pointed to by *buffer* is equivalent to fscanf() reaching EOF. If the strings pointed to by *buffer* and *format* overlap, behavior is undefined.

fscanf() and scanf() have the same restriction as any read operation for a read immediately following a write or a write immediately following a read. Between a write and a subsequent read, there must be an intervening flush or reposition. Between a read and a subsequent write, there must also be an intervening flush or reposition unless an EOF has been reached.

For all three functions, each entry in the argument list must be a pointer to a variable of a type that matches the corresponding conversion specification in *format-string*. If the types do not match, the results are undefined.

For all three functions, the *format-string* controls the interpretation of the argument list. The *format-string* can contain multibyte characters beginning and ending in the initial shift state.

The format string pointed to by *format-string* can contain one or more of the following:

- White space characters, as specified by isspace(), such as blanks and newline characters. A white space character causes fscanf(), scanf(), and sscanf() to read, but not to store, all consecutive white space characters in the input up to the next character that is not white space. One white space character in *format-string* matches any combination of white space characters in the input.
- Characters that are not white space, except for the percent sign character (%). A non-white space character causes fscanf(), scanf(), and sscanf() to read, but not to store, a matching non-white space character. If the next character in the input stream does not match, the function ends.
- Conversion specifications which are introduced by the percent sign (%) or the sequence (%n$) where n is a decimal integer in the range [1,NL_ARGMAX]. A conversion specification causes fscanf(), scanf(), and sscanf() to read and convert characters in the input into values of a conversion specifier. The value is assigned to an argument in the argument list.

All three functions read *format-string* from left to right. Characters outside of conversion specifications are expected to match the sequence of characters in the input stream; the matched characters in the input stream are scanned but not stored. If a character in the input stream conflicts with *format-string*, the function ends, terminating with a "matching" failure. The conflicting character is left in the input stream as if it had not been read.

When the first conversion specification is found, the value of the first *input field* is converted according to the conversion specification and stored in the location specified by the first entry in the argument list. The second conversion specification converts the second input field and stores it in the second entry in the argument list, and so on through the end of *format-string*.

fscanf_unlocked() is functionally equivalent to fscanf() with the exception that it is not thread-safe. This function can safely be used in a multithreaded application if and only if it is called while the invoking thread owns the (FILE*) object, as is the case after a successful call to either the flockfile() or ftrylockfile() function.

scanf_unlocked() is functionally equivalent to scanf() with the exception that it is not thread-safe. This function can safely be used in a multithreaded application if and only if it is called while the invoking thread owns the (FILE*) object, as is the case after a successful call to either the flockfile() or ftrylockfile() function.

An *input field* is defined as:

- All characters until a white space character (space, tab, or newline) is encountered
- All characters until a character is encountered that cannot be converted according to the conversion specification
- All characters until the field *width* is reached.

If there are too many arguments for the conversion specifications, the extra arguments are evaluated but otherwise ignored. The results are undefined if there are not enough arguments for the conversion specifications.

Syntax of Conversion Specification for fscanf(), scanf(), and sscanf()

```
►►──%──┬───────┬──┬───────┬──┬──────┬──conversion specifier──────────────────►◄
       └─flags─┘  └─width─┘  ├─h────┤
                             ├─hh───┤
                             ├─l────┤
                             ├─ll───┤
                             ├─j────┤
                             ├─t────┤
                             ├─z────┤
                             ├─D────┤
                             ├─DD───┤
                             ├─H────┤
                             ├─L────┤
                             ├─v────┤
                             ├─vh───┤
                             ├─hv───┤
                             ├─vl───┤
                             ├─lv───┤
                             ├─vll──┤
                             ├─llv──┤
                             ├─vL───┤
                             └─Lv───┘
```

Each field of the conversion specification is a single character or a number signifying a particular format option. The *conversion specifier*, which appears after the last optional format field, determines whether the input field is interpreted as a character, a string, or a number. The simplest conversion specification contains only the percent sign and a *conversion specifier* (for example, `%s`).

Each field of the format specification is discussed in detail below.

Other than conversion specifiers, you should avoid using the percent sign (`%`), except to specify the percent sign: `%%`. Currently, the percent sign is treated as the start of a conversion specifier. Any unrecognized specifier is treated as an ordinary sequence of characters. If, in the future, z/OS XL C/C++ permits a new conversion specifier, it could match a section of your format string, be interpreted incorrectly, and result in undefined behavior. See Table 32 on page 630 for a list of conversion specifiers.

Flag characters: The *flag* characters in Table 31 are used for assignment-suppressing and separator for vector type input conversion. Notice that more than one *flag* can appear in a format specification. This is an optional field.

Table 31. Flag Characters for fscanf Family

Flag	Meaning	Default
*	Suppresses assignment of the next input field, which is interpreted as a field of the specified conversion specifier. The field is scanned but not stored.	

Table 31. Flag Characters for fscanf Family (continued)

Flag	Meaning	Default
, ; : _	Indicates the separator character between each two components in the vector data type conversion. Only one separator character can be specified.	The default value for the separator character is a space (' '), unless the c format is being used. For the c format, there is no separator at all if no separator character is specified.

width is a positive decimal integer controlling the maximum number of characters to be read. No more than *width* characters are converted and stored at the corresponding *argument*.

Fewer than *width* characters are read if a white space character (space, tab, or newline), or a character that cannot be converted according to the given format occurs before *width* is reached.

Optional prefix: The optional prefix characters used to indicate the size of the argument expected or size of a single value in a vector data type input conversion are explained:

Prefix	Meaning
h	Specifies that the d, i, o, u, x, X, or n conversion specifier applies to an argument with type pointer to short or unsigned short.
hh	Specifies that the d, i, o, u, x, X, or n conversion specifier applies to an argument with type pointer to signed char or unsigned char.
l	(ell) Specifies that the d, i, o, u, x, X, or n conversion specifier applies to an argument with type pointer to long or unsigned long; the following a, A, e, E, f, F, g, or G conversion specifier applies to an argument with type pointer to double; and the following c, s, or [conversion specifier applies to an argument with type pointer to wchar_t.
ll	(ell-ell) Specifies that the d, i, o, u, x, X, or n conversion specifier applies to an argument with type pointer to long long or unsigned long long.
j	Specifies that the d, i, o, u, x, X, or n conversion specifier applies to an argument with type pointer to intmax_t or uintmax_t.
t	Specifies that the d, i, o, u, x, X, or n conversion specifier applies to an argument with type pointer to ptrdiff_t or the corresponding unsigned type.
z	Specifies that the d, i, o, u, x, X, or n conversion specifier applies to an argument with type pointer to size_t or the corresponding signed integer type.
D	Specifies that the e, E, f, F, g, or G conversion specifier applies to an argument with type pointer to _Decimal64 float.
DD	Specifies that the e, E, f, F, g, or G conversion specifier applies to an argument with type pointer to _Decimal128 float.
H	Specifies that the e, E, f, F, g, or G conversion specifier applies to an argument with type pointer to _Decimal32 float.

L Specifies that the `a`, `A`, `e`, `E`, `f`, `F`, `g`, or `G` conversion specifier applies to an argument with type pointer to long double.

v Specifies that a following `c`, `d`, `i`, `u`, `o`, `x`, or `X` conversion specifier applies to an argument with type pointer to `vector signed char`, `vector unsigned char`, or `vector bool char`.

vh, hv Specifies that a following `d`, `i`, `u`, `o`, `x`, or `X` conversion specifier applies to an argument with type pointer to `vector signed short`, `vector unsigned short`, or `vector bool short`.

vl, lv Specifies that a following `d`, `i`, `u`, `o`, `x`, or `X` conversion specifier applies to an argument with type pointer to `vector signed int`, `vector unsigned int`, or `vector bool int`.

vll, llv Specifies that a following `d`, `i`, `u`, `o`, `x`, or `X` conversion specifier applies to an argument with type pointer to `vector signed long long`, `vector unsigned long long`, or `vector bool long long`.

vL, Lv specifies that a following `a`, `A`, `e`, `E`, `f`, `F`, `g`, or `G` conversion specifier applies to an argument with type pointer to `vector double`.

Conversion specifier: Table 32 explains the valid conversion specifiers and their meanings are in.

Table 32. Conversion Specifiers in fscanf Family

Conversion Specifier	Type of Input Expected	Type of Argument
d	Decimal integer	Pointer to `int`
o	Octal integer	Pointer to `unsigned int`
x X	Hexadecimal integer	Pointer to `unsigned int`
i	Decimal, hexadecimal, or octal integer	Pointer to `int`
u	Unsigned decimal integer	Pointer to `unsigned int`
e E f F g G	Floating-point value consisting of an optional sign (+ or -); a series of one or more decimal digits possibly containing a decimal-point; and an optional exponent (e or E) followed by a possibly signed integer value	Pointer to `float`
a A	Matches an optionally signed floating-point number, infinity, or NaN, whose format is the same as expected for the subject sequence of strtod(). In the absence of a size modifier, the application shall ensure that the corresponding argument is a pointer to float.	Pointer to `float`
D(n,p)	Fixed-point value consisting of an optional sign (+ or -); a series of one or more decimal digits possibly containing a decimal-point.	Pointer to decimal
c	Sequence of one or more characters as specified by field width; white space characters that are ordinarily skipped are read when `%c` is specified. No terminating null is added.	Pointer to `char` large enough for input field.

Table 32. Conversion Specifiers in fscanf Family (continued)

Conversion Specifier	Type of Input Expected	Type of Argument
C or lc	The input is a sequence of one or more multibyte characters as specified by the field width, beginning in the initial shift state. Each multibyte character in the sequence is converted to a wide character as if by a call to the mbrtowc() function. The conversion state described by the mbstate_t object is initialized to zero before the first multibyte character is converted. The corresponding argument is a pointer to the initial element of an array of `wchar_t` large enough to accept the resulting sequence of wide characters. No NULL wide character is added.	`C` or `lc` uses a pointer to `wchar_t`.
s	Like `c`, a sequence of bytes of type `char` (signed or unsigned), except that white space characters are not allowed, and a terminating null is always added.	Pointer to character array large enough for input field, plus a terminating NULL character (`\0`) that is automatically appended.
S or ls	A sequence of multibyte characters that begins and ends in the initial shift state. Each multibyte character in the sequence is converted to a wide character as if by a call to the mbrtowc() function, with the conversion state described by the mbstate_t object initialized to zero before the first multibyte character is converted. The corresponding argument is a pointer to the initial array of `wchar_t` large enough to accept the sequence and the terminating NULL wide character, which is added automatically.	`S` or `ls` uses a pointer to `wchar_t` string.
n	No input read from *stream* or buffer.	Pointer to `int`, into which is stored the number of characters successfully read from the *stream* or buffer up to that point in the call to either fscanf() or to scanf().
p	Pointer to `void` converted to series of characters. For the specific format of the input, see the individual system reference guides.	Pointer to `void`.

Table 32. Conversion Specifiers in fscanf Family (continued)

Conversion Specifier	Type of Input Expected	Type of Argument
[	A non-empty sequence of bytes to be matched against a set of expected bytes (the *scanset*), which form the conversion specification. White space characters that are ordinarily skipped are read when %[is specified. Consider the following situations: [^*bytes*]. In this case, the scanset contains all bytes that do not appear between the circumflex and the right square bracket. []abc] *or* [^]abc.] In both these cases the right square bracket is included in the scanset (in the first case:]abc and in the second case, *not*]abc) [a–z] In EBCDIC The – is in the scanset, the characters b through y are *not* in the scanset; in ASCII The – is *not* in the scanset, the characters b through y are. The code point for the square brackets ([and]) and the caret (^) vary among the EBCDIC encoded character sets. The default C locale expects these characters to use the code points for encoded character set Latin-1 / Open Systems 1047. Conversion proceeds one byte at a time: there is no conversion to wide characters.	Pointer to the initial byte of an array of char, signed char, or unsigned char large enough to accept the sequence and a terminating byte, which will be added automatically.
l[	If an l length modifier is present, input is a sequence of multibyte characters that begins and ends in the initial shift state. Each multibyte character in the sequence is converted to a wide character as if by a call to the mbrtowc() function, with the conversion state described by the mbstate_t object initialized to zero before the first multibyte character is converted. The corresponding argument is a pointer to the initial array of wchar_t large enough to accept the sequence and the terminating NULL wide character, which is added automatically.	l[uses a pointer to wchar_t string

When the LC_SYNTAX category is set using setlocale(), the format strings passed to the fscanf(), scanf(), or sscanf() functions must use the same encoded character set as is specified for the LC_SYNTAX category.

To read strings not delimited by space characters, substitute a set of characters in square brackets ([ ]) for the s (string) conversion specifier. The corresponding input field is read up to the first character that does not appear in the bracketed character set. If the first character in the set is a logical not (¬), the effect is reversed: the input field is read up to the first character that does appear in the rest of the character set.

To store a string without storing an ending NULL character (\0), use the specification %*a*c, where *a* is a decimal integer. In this instance, the c conversion specifier means that the argument is a pointer to a character array. The next *a* characters are read from the input stream into the specified location, and no NULL character is added.

The input for a *%x* conversion specifier is interpreted as a hexadecimal number.

All three functions, fscanf(), scanf(), and sscanf() scan each input field character by character. It might stop reading a particular input field either before it reaches a space character, when the specified *width* is reached, or when the next character cannot be converted as specified. When a conflict occurs between the specification and the input character, the next input field begins at the first unread character. The conflicting character, if there is one, is considered unread and is the first character of the next input field or the first character in subsequent read operations on the input stream.

Special behavior for XPG4.2:

- When the %n$ conversion specification is found, the value of the *input field* is converted according to the conversion specification and stored in the location specified by the nth argument in the argument list. Numbered arguments in the argument list can only be referenced once from *format-string*.
- The *format-string* can contain either form of the conversion specification, that is, % or %n$ but the two forms cannot be mixed within a single *format-string* except that %% or %* can be mixed with the %n$ form.

Floating-point and the fscanf family of formatted input functions: The fscanf family functions match `e`, `E`, `f`, `F`, `g` or `G` conversion specifiers to floating-point number substrings in the input stream. The fscanf family functions convert each input substring matched by an `e`, `E`, `f`, `F`, `g` or `G` conversion specifier to a float, double or long double value depending on a size modifier preceding the `e`, `E`, `f`, `F`, `g` or `G` conversion specifier.

The floating-point value produced is hexadecimal floating-point or IEEE Binary Floating-Point format depending on the floating-point mode of the thread invoking the fscanf family function. The fscanf family functions use __isBFP() to determine the floating-point mode of invoking threads.

Many z/OS XL C/C++ formatted input functions, including the fscanf family, recognize special infinity and NaN floating-point number input sequences when the invoking thread is in IEEE Binary Floating-Point mode as determined by __isBFP().

- The special sequence for infinity input is an optional plus or minus sign, then the character sequence INF, where the individual characters may be uppercase or lowercase, and then a white space character (space, tab, or newline), a NULL character (`\0`) or EOF.
- The special sequence for NaN input is an optional plus or minus sign, then the character sequence NANS for a signalling NaN or NANQ for a quiet NaN, where the individual characters may be uppercase or lowercase, then an optional NaN ordinal sequence, and then a a white space character (space, tab, or newline), a NULL character (`\0`) or EOF.

  For binary floating point NANs: A NaN ordinal sequence is a left-parenthesis character, “(”, followed by a digit sequence representing an integer n, where 1 <= n <= INT_MAX-1, followed by a right-parenthesis character, “)”. If the NaN ordinal sequence is omitted, NaN ordinal sequence (1) is assumed. The integer value, n, corresponding to a NaN ordinal sequence determines what IEEE Binary Floating-Point NaN fraction bits are produced by formatted input functions.

For a signalling NaN, these functions produce NaN fraction bits (left to right) by reversing the bits (right to left) of the even integer value 2*n.

For a quiet NaN they produce NaN fraction bits (left to right) by reversing the bits (right to left) of the odd integer value 2*n-1.

For decimal floating point NANs: A NaN ordinal sequence is a left parenthesis character, "(", followed by a decimal digit sequence of up to 6 digits for a _Decimal32 output number, up to 15 digits for a _Decimal64 output value, or up to 33 digits for a _Decimal128 output value, followed by a right parenthesis, ")". If the NaN ordinal sequence is omitted, NaN ordinal sequence "(0)" is assumed. If the NaN ordinal sequence is shorter than 6, 15, or 33 digits, it will be padded on the left with "0" digits so that the length becomes 6, 15, or, 33 digits for _Decimal32, _Decimal64, and _Decimal128 values respectively.

For decimal floating point numbers, the digits are not reversed, and both odd or even NaN ordinal sequences can be specified for either a Quiet NAN or Signalling NAN.

Vector types and the fscanf family of formatted input functions: The fscanf family functions support character, integer, and floating-point vector data types in the input conversion specification. The vector value to be scanned is in the following general form:

```
Value1Cvalue2C...CValueN
```

where C is a separator character optionally defined in flag characters, and there can be 2, 4, 8, or 16 components depending on the optional prefix v, vh, hv, vl, lv, vll, llv, vL, or Lv. Each value is scanned according to the conversion specifier. If the conversion fails in the middle of a vector value, the successful elements will be saved to the vector argument and the other elements in the vector argument will remain unchanged.

Usage note

To use IEEE decimal floating-point, the hardware must have the Decimal Floating-Point Facility installed.

Returned value

All three functions, fscanf(), scanf(), and sscanf() return the number of input items that were successfully matched and assigned. The returned value does not include conversions that were performed but not assigned (for example, suppressed assignments). The functions return EOF if there is an input failure before any conversion, or if EOF is reached before any conversion. Thus a returned value of 0 means that no fields were assigned: there was a matching failure before any conversion. Also, if there is an input failure, then the file error indicator is set, which is not the case for a matching failure.

The ferror() and feof() functions are used to distinguish between a read error and an EOF. Note that EOF is only reached when an attempt is made to read “past” the last byte of data. Reading up to and including the last byte of data does *not* turn on the EOF indicator.

Examples

CELEBF42

```
/* CELEBF42

   This example scans various types of data

 */
#include <stdio.h>

int main(void)
{
   int i;
   float fp;
   char c, s[81];

   printf("Enter an integer, a real number, a character "
          "and a string : \n");
   if (scanf("%d %f %c %s", &i, &fp, &c, s) != 4)
      printf("Not all of the fields were assigned\n");
   else
   {
      printf("integer = %d\n", i);
      printf("real number = %f\n", fp);
      printf("character = %c\n", c);
      printf("string = %s\n",s);
   }
}
```

Output

If input is: `12 2.5 a yes`, then output would be:

```
Enter an integer, a real number, a character and a string:
integer = 12
real number = 2.500000
character = a
string = yes
```

CELEBF43

```
/* CELEBF43
   This example converts a hexadecimal integer to a decimal integer.
   The while loop ends if the input value is not a hexadecimal integer.
 */
#include <stdio.h>

int main(void)
{
   int number;

   printf("Enter a hexadecimal number or anything else to quit:\n");
   while (scanf("%x",&number))
      {
      printf("Hexadecimal Number = %x\n",number);
      printf("Decimal Number     = %d\n",number);
      }
}
```

Output

If input is: `0x231 0xf5e 0x1 q`, then output would be:

```
Enter a hexadecimal number or anything else to quit:
Hexadecimal Number = 231
Decimal Number     = 561
Hexadecimal Number = f5e
Decimal Number     = 3934
Hexadecimal Number = 1
Decimal Number     = 1
```

CELEBF44

```
/* CELEBF44
   The next example illustrates the use of scanf() to input fixed-point
   decimal data types. This example works under C only, not C++.
 */
#include <stdio.h>
#include <decimal.h>

decimal(15,4) pd01;
decimal(10,2) pd02;
decimal(5,5) pd03;

int main(void) {
  printf("\nFirst time :-------------------------------\n");
  printf("Enter three fixed-point decimal number\n");
  printf("  (15,4) (10,2) (5,5)\n");
  if (scanf("%D(15,4) %D(10,2) %D(5,5)", &pd01, &pd02, &pd03) != 3) {
    printf("Error found in scanf\n");
  } else {
    printf("pd01 = %D(15,4)\n", pd01);
    printf("pd02 = %D(10,2)\n", pd02);
    printf("pd03 = %D(5,5)\n", pd03);
  }
  printf("\nSecond time :------------------------------\n");
  printf("Enter three fixed-point decimal number\n");
  printf("  (15,4) (10,2) (5,5)\n");
  if (scanf("%D(15,4) %D(10,2) %D(5,5)", &pd01, &pd02, &pd03) != 3) {
    printf("Error found in scanf\n");
  } else {
    printf("pd01 = %D(15,4)\n", pd01);
    printf("pd02 = %D(10,2)\n", pd02);
    printf("pd03 = %D(5,5)\n", pd03);
  }
  return(0);
}
```

Output

```
First time :-------------------------------
Enter three fixed-point decimal number
  (15,4) (10,2) (5,5)
12345678901.2345 -987.6 .24680
pd01 = 12345678901.2345
pd02 = -987.60
pd03 = 0.24680

Second time :------------------------------
Enter three fixed-point decimal number
  (15,4) (10,2) (5,5)
123456789013579.24680 123.4567890 987
pd01 = 12345678901.3579
pd02 = 123.45
pd03 = 0.98700
```

CELEBF46

```
/* CELEBF46
   The next example opens the file myfile.dat for reading and then scans
   this file for a string, a long integer value, a character, and a
   floating-point value.
 */
#include <stdio.h>
#define  MAX_LEN  80

int main(void)
{
   FILE *stream;
   long l;
   float fp;
   char s[MAX_LEN + 1];
   char c;

   stream = fopen("myfile.dat", "r");

   /* Put in various data. */
   fscanf(stream, "%s", &s[0]);
   fscanf(stream, "%ld", &l);
   fscanf(stream, "%c", &c);
   fscanf(stream, "%f", &fp);

   printf("string = %s\n", s);
   printf("long double = %ld\n", l);
   printf("char = %c\n", c);
   printf("float = %f\n", fp);
}
```

Output

If `myfile.dat` contains `abcdefghijklmnopqrstuvwxyz 343.2`, then the expected output is:

```
string = abcdefghijklmnopqrstuvwxyz
long double = 343
char = .
float = 2.000000
```

CELEBS32

```
/* CELEBS32
   This example uses sscanf() to read various data from the string
   tokenstring, and then displays the data.
 */
#include <stdio.h>
#define  SIZE  81

int main(void)
{
char *tokenstring = "15 12 14";
int i;
float fp;
char s[SIZE];
char c;

   /* Input various data                           */
   printf("No. of conversions=%d\n",
      sscanf(tokenstring, "%s %c%d%f", s, &c, &i, &fp));

   /* If there were no space between %s and %c,       */
   /* sscanf would read the first character following */
   /* the string, which is a blank space.            */

   /* Display the data */
   printf("string = %s\n",s);
   printf("character = %c\n",c);
   printf("integer = %d\n",i);
   printf("floating-point number = %f\n",fp);
}
```

Output

You would see this output from example CELEBS32.

```
No. of conversions = 4
string = 15
character = 1
integer = 2
floating-point number = 14.000000
```

CELEBF91

```
/* CELEBF91
   The following example illustrates the use of sscanf()
   to input vector data types.
 */
#include <stdio.h>

int main (void)
{
   vector signed char s8;
   vector unsigned short u16;
   vector signed int s32;
   vector unsigned long long u64;
   vector double d64;

   sscanf("ab defghijklm,op", "%vc", &s8);
   printf("s8 = %vc\n", s8);

   sscanf("a,b, ,d,e,f,g,h,i,j,k,l,m,!,o,p", "%,vc", &s8);
   printf("s8 = %,vc\n", s8);

   sscanf("1 2 3 4 5 6 7 8", "%vhu", &u16);
   printf("u16 = %vhu\n", u16);

   Sscanf(" 1, 2, 3,99", "%,2lvd", &s32);
   printf("s32 = %,2lvd\n", s32);

   sscanf("1 2", "%vllu", &u64);
   printf("u64 = %vllu\n", u64);

   sscanf("0.1 1.2", "%vLf\n", &d64);
   printf("d64 = %vLf\n", d64);
}
```

Output

```
s8 = ab defghijklm!op
s8 = a,b, ,d,e,f,g,h,i,j,k,l,m,!,o,p
u16 = 1 2 3 4 5 6 7 8
s32 =  1, 2, 3,99
u64 = 1 2
d64 = 0.100000 1.200000
```

Related information

- See the topic about internationalization of locales and character sets in *z/OS XL C/C++ Programming Guide*.
- “locale.h” on page 40
- “stdio.h” on page 68
- “fprintf(), printf(), sprintf() — Format and write data” on page 588
- “__isBFP() — Determine application floating-point format” on page 904
- “localtime(), localtime64() — Convert time and correct for local time” on page 984
- “setlocale() — Set locale” on page 1552

fseek() — Change file position

Standards

Standards / Extensions	C or C++	Dependencies
ISO C POSIX.1 XPG4 XPG4.2 C99 Single UNIX Specification, Version 3 Language Environment	both	

Format

```
#include <stdio.h>

int fseek(FILE *stream, long int offset, int origin);

#define _OPEN_SYS_UNLOCKED_EXT 1
#include <stdio.h>

int fseek_unlocked(FILE *stream, long int offset, int origin);
```

General description

The fseek() function changes the current file position associated with *stream* to a new location within the file. The next operation on the stream takes place at the new location. On a stream opened for update, the next operation can be either a reading or a writing operation.

The *origin* must be one of the following constants defined in stdio.h:

Origin Definition

SEEK_SET
Beginning of file

SEEK_CUR
Current position of file pointer

SEEK_END
End of file

If successful, the fseek() function clears the EOF indicator, even when *origin* is SEEK_END, and cancels the effect of any preceding ungetc() or ungetwc() function on the same stream.

If the call to the fseek() function or the fsetpos() function is not valid, the call is treated as a flush and the ungetc characters are discarded.

Behavior for binary streams: ANSI states that binary streams use relative byte offsets for both the ftell() and fseek() functions. Under z/OS XL C/C++, this is true except for record-oriented files that have variable length records. For these types of files, the default behavior is to use encoded offsets for the ftell() function and the fseek() function using an origin of SEEK_SET.

Encoded offsets restrict you to seeking only to those positions that are recorded by a previous ftell() function call or to position 0. If you want to use relative-byte offsets for these types of files, you can either open the file with the BYTESEEK fopen() function option or set the _EDC_BYTE_SEEK environment variable before opening.

With relative-byte offsets, you can calculate your own offsets. If the offset exceeds the EOF, your file is extended with NULLs, except for z/OS UNIX files, for which the file is only extended with NULLs if you subsequently write new data. This is true also under POSIX, using z/OS UNIX files, where the file is only extended with NULLs if you subsequently write new data.

Attempting to reposition to before the start of the file causes the fseek() function to fail.

Regardless of whether encoded or relative offsets are returned by the ftell() function, you can specify relative offsets when using SEEK_CUR and SEEK_END.

If the new position is before the start of the file, the fseek() function fails. If the relative offset is positioned beyond the EOF, the file is padded with NULLs, except in the case of POSIX, using z/OS UNIX files, where padding does not occur until a subsequent write of new data.

Behavior for text streams: For text streams, the ftell() function returns an encoded offset. When seeking with an origin of SEEK_SET, you are restricted to seeking only to 0 or to positions returned by a previous ftell() function call.

Attempting to calculate your own position is not supported, and might result in a non-valid position and the failure of the fseek() function.

When you are using SEEK_CUR or SEEK_END, the offset is a relative byte offset. Attempting to seek to before the start of the file or past the EOF results in failure.

Behavior for record I/O: For files opened for record I/O using the `type=record` open mode parameter, the ftell() function returns the relative record number. For the origins of SEEK_SET, SEEK_CUR, and SEEK_END, the offset is a relative record number.

Attempting to seek to before the first record or past the EOF results in failure.

Behavior for blocked I/O: For files opened for blocked I/O using the `type=blocked` open mode parameter, the ftell() function returns the relative block number. For the origins of SEEK_SET, SEEK_CUR, and SEEK_END, the offset is a relative block number.

Attempting to seek to before the first block or past the EOF results in failure.

Behavior for wide-oriented streams: All of the above restrictions apply for wide-oriented streams of any type.

Multivolume data sets performance: Using the fgetpos() and fsetpos() functions generally results in better repositioning performance compared to the ftell() and fseek() functions when working with multivolume data sets.

Large file support for MVS data sets, VSAM data sets, and z/OS UNIX files: For AMODE 31 C/C++ applications, the fseek() function accepts a signed 4-byte offset and therefore cannot be used to directly or relatively position to offsets beyond 2 GB - 1. To avoid repositioning limitations, AMODE 31 C/C++ applications should define the _LARGE_FILES feature test macro before any headers are included and replace the fseek() function with the fseeko() function. For AMODE 64 C/C++ applications, there are no restrictions on using the fseek() function with large files. The AMODE 64 version automatically accepts a signed 8-byte offset.

Usage notes

1. Repositioning within a wide-oriented file and performing updates is strongly discouraged because it is not possible to predict if your update will overwrite part of a multibyte string or character, thereby invalidating subsequent data. For example, you could inadvertently add data that overwrites a shift-out. The following data expects the shift-out to be there, so is not valid if it is treated as if in the initial shift state. Repositioning to the end of the file and adding new data is safe.
2. If you specify SEEK_CUR, any characters pushed back by the ungetc() or ungetwc() functions will have backed up the current position of the file pointer—which is the starting point of the seek. The seek will discard any pushed-back characters before repositioning, but the starting point will still be affected. For more information about calling the fseek() function after an ungetc() or ungetwc() function call, see “ungetc() — Push character onto input stream” on page 1958 and “ungetwc() — Push a wide character onto a stream” on page 1961.
3. The _EDC_COMPAT environment variable causes fseek() to ignore the effects of the ungetc() or ungetwc() functions. For more details, see the topic about environment variables in *z/OS XL C/C++ Programming Guide*.
4. The fseek_unlocked() function is functionally equivalent to the fseek() function with the exception that it is not threadsafe. The fseek() function can safely be used in a multithreaded application if, and only if, it is called while the invoking thread owns the (FILE*) object, as is the case after a successful call to either the flockfile() or ftrylockfile() function.

Returned value

If successful in moving the pointer, the fseek() function returns 0.

If unsuccessful, or on devices that cannot seek, such as terminals and printers, the fseek() function returns nonzero.

Special behavior for XPG4.2: If unsuccessful, the fseek() function returns -1 and sets errno to one of the following values:

Error Code
Description

EOVERFLOW
The resulting file offset would be a value which cannot be represented correctly in an object of type long.

Note: Environment variable _EDC_EOVERFLOW can be used to control behavior of the fseek() function with respect to detecting an EOVERFLOW condition for z/OS UNIX files. By default, the fseek() function will continue to be able to position beyond a location that the ftell() function can return. When _EDC_EOVERFLOW is set to YES, the fseek() function will check if the new position can be returned by the ftell() function.

ESPIPE
The underlying file type for the stream is a PIPE or a socket.

Example

```
/* This example opens a file myfile.dat for reading.
   After performing input operations (not shown), it moves the file
   pointer to the beginning of the file.
 */
#include <stdio.h>

int main(void)
{
   FILE *stream;
   int result;

   if (stream = fopen("myfile.dat", "r"))
   { /* successful */

   if (fseek(stream, 0L, SEEK_SET));  /* moves pointer to   */
                                   /* the beginning of the file */
   { /* if not equal to 0
             then error ...    */
   }
   else {
       /* fseek() successful  */
   }
}
```

Related information

- For information about wide-oriented streams, see *z/OS XL C/C++ Programming Guide*.
- For information about BYTESEEK or _EDC_BYTE_SEEK, see *z/OS XL C/C++ Programming Guide*.
- For additional usage information about the fseek() function with respect to MVS data sets, VSAM data sets, or z/OS UNIX files, see *z/OS XL C/C++ Programming Guide*.
- "stdio.h" on page 68
- "fseeko() — Change file position"
- "ftell() — Get current file position" on page 657
- "ungetc() — Push character onto input stream" on page 1958
- "ungetwc() — Push a wide character onto a stream" on page 1961

fseeko() — Change file position

Standards

Standards / Extensions	C or C++	Dependencies
Single UNIX Specification, Version 2 Single UNIX Specification, Version 3 Language Environment	both	OS/390 V2R10

Format

```
#define _XOPEN_SOURCE 500
#include <stdio.h>

int fseeko(FILE *stream, off_t offset, int origin);

#define _OPEN_SYS_UNLOCKED_EXT 1
#include <stdio.h>

int fseeko_unlocked(FILE *stream, off_t offset, int origin);
```

General description

The fseeko() function changes the current file position associated with *stream* to a new location within the file. The next operation on the stream takes place at the new location. On a stream opened for update, the next operation can be either a reading or a writing operation.

The *origin* must be one of the following constants defined in stdio.h:

Origin Definition

SEEK_SET
Beginning of file

SEEK_CUR
Current position of file pointer

SEEK_END
End of file

If successful, the fseeko() function clears the EOF indicator, even when the origin is SEEK_END, and cancels the effect of any preceding ungetc() or ungetwc() function on the same stream.

If the call to the fseeko() function or the fsetpos() function is not valid, the call is treated as a flush and the ungetc characters are discarded.

Behavior for binary streams: ANSI states that binary streams use relative byte offsets for both the ftello() and fseeko() functions. Under z/OS XL C/C++, this is true except for record-oriented files that have variable length records. For these types of files, the default behavior is to use encoded offsets for the ftello() function and the fseeko() function using an origin of SEEK_SET.

Encoded offsets restrict you to seeking only to those positions that are recorded by a previous ftello() function call or to position 0. If you want to use relative-byte offsets for these types of files, you can either open the file with the BYTESEEK fopen() function option or set the _EDC_BYTE_SEEK environment variable before opening.

With relative-byte offsets, you can calculate your own offsets. If the offset exceeds the EOF, your file is extended with NULLs, except for z/OS UNIX files, for which the file is only extended with NULLs if you subsequently write new data. This is true also under POSIX, using z/OS UNIX files, where the file is only extended with NULLs if you subsequently write new data.

Attempting to reposition to before the start of the file causes the fseeko() function to fail.

Regardless of whether encoded or relative offsets are returned by the ftello() function, you can specify relative offsets when using SEEK_CUR and SEEK_END.

If the new position is before the start of the file, the fseeko() function fails. If the relative offset is positioned beyond the EOF, the file is padded with NULLs, except in the case of POSIX, using z/OS UNIX files, where padding does not occur until a subsequent write of new data.

Behavior for text streams: For text streams, the ftello() function returns an encoded offset. When seeking with an origin of SEEK_SET, you are restricted to seeking only to 0 or to positions returned by a previous ftello() function call.

Attempting to calculate your own position is not supported, and might result in a non-valid position and the failure of the fseeko() function.

When you are using SEEK_CUR or SEEK_END, the offset is a relative byte offset. Attempting to seek to before the start of the file or past the EOF results in failure.

Behavior for record I/O: For files opened for record I/O using the `type=record` open mode parameter, the ftello() function returns the relative record number. For the origins of SEEK_SET, SEEK_CUR, and SEEK_END, the offset is a relative record number.

Attempting to seek to before the first record or past the EOF results in failure.

Behavior for blocked I/O: For files opened for blocked I/O using the `type=blocked` open mode parameter, the ftello() function returns the relative block number. For the origins of SEEK_SET, SEEK_CUR, and SEEK_END, the offset is a relative block number.

Attempting to seek to before the first block or past the EOF results in failure.

Behavior for wide-oriented streams: All of the above restrictions apply for wide-oriented streams of any type.

Multivolume data sets performance: Using the fgetpos() and fsetpos() functions generally results in better repositioning performance compared to the ftello() and fseeko() functions when working with multivolume data sets.

Large file support for MVS data sets, VSAM data sets, and z/OS UNIX files: For AMODE 31 C/C++ applications, the fseeko() function accepts a signed 4-byte offset and therefore cannot be used to directly or relatively position to offsets beyond 2 GB - 1. To avoid repositioning limitations, AMODE 31 C/C++ applications should define the _LARGE_FILES feature test macro before any headers are included. For AMODE 64 C/C++ applications, there are no restrictions on using the fseeko() function with large files. The AMODE 64 version automatically accepts a signed 8-byte offset.

Usage notes

1. Repositioning within a wide-oriented file and performing updates is strongly discouraged because it is not possible to predict if your update will overwrite part of a multibyte string or character, thereby invalidating subsequent data. For example, you could inadvertently add data that overwrites a shift-out. The following data expects the shift-out to be there, so is not valid if it is treated as if in the initial shift state. Repositioning to the end of the file and adding new data is safe.
2. If you specify SEEK_CUR, any characters pushed back by the ungetc() or ungetwc() functions will have backed up the current position of the file pointer—which is the starting point of the seek. The seek will discard any pushed-back characters before repositioning, but the starting point will still be affected. For more information about calling the fseeko() function after an

ungetc() or ungetwc() function call, see “ungetc() — Push character onto input stream” on page 1958 and “ungetwc() — Push a wide character onto a stream” on page 1961.

3. The _EDC_COMPAT environment variable causes fseeko() to ignore the effects of the ungetc() or ungetwc() functions. For more details, see the topic about environment variables in *z/OS XL C/C++ Programming Guide*.
4. The fseeko_unlocked() function is functionally equivalent to the fseeko() function with the exception that it is not threadsafe. The fseek() function can safely be used in a multithreaded application if, and only if, it is called while the invoking thread owns the (FILE*) object, as is the case after a successful call to either the flockfile() or ftrylockfile() function.

Returned value

If successful, fseeko() returns 0, which means it successfully moved the pointer.

If unsuccessful, fseeko() returns nonzero and sets errno to one of the following values.

On devices that cannot seek, such as terminals and printers, the fseeko() function returns nonzero.

Error Code
  Description

EBADF
  The file descriptor underlying stream is not an open file descriptor.

EOVERFLOW
  The current file offset cannot be represented correctly in an object of type off_t.

ESPIPE
  The file descriptor underlying stream is associated with a pipe or FIFO.

Example

```
/* This example opens a file myfile.dat for reading.
   After performing input operations (not shown), it moves the file
   pointer to the beginning of the file.
 */
#define _LARGE_FILES 1
#include <stdio.h>

int main(void)
{
   FILE *stream;
   int result;

   if (stream = fopen("/myfile.dat", "r"))

   { /* successful */

   if (fseeko(stream, 0LL, SEEK_SET));  /* moves pointer to   */
                                       /* the beginning of the file */
   { /* if not equal to 0
             then error ...    */
   }
   else {
       /* fseeko() successful  */
   }
}
```

Related information

- For information about wide-oriented streams, see *z/OS XL C/C++ Programming Guide*.
- For information about BYTESEEK or _EDC_BYTE_SEEK, see *z/OS XL C/C++ Programming Guide*.
- For additional usage information about the fseeko() function with respect to MVS data sets, VSAM data sets, or z/OS UNIX files, see *z/OS XL C/C++ Programming Guide*.
- "stdio.h" on page 68
- "fseek() — Change file position" on page 639
- "ftello() — Get current file position" on page 659
- "ungetc() — Push character onto input stream" on page 1958
- "ungetwc() — Push a wide character onto a stream" on page 1961

__fseterr() — Set stream in error

Standards

Standards / Extensions	C or C++	Dependencies
Language Environment	both	

Format

```
#include <stdio.h>
#include <stdio_ext.h>

void __fseterr(FILE *stream);
```

General description

The __fseterr() function sets the specified stream in error.

Returned value

The __fseterr() function returns no values.

An application wishing to check for error situations should set errno to 0, then call __fseterr(), and then check errno. If errno is nonzero, assume that an error has occurred.

Error Code
    Description

EBADF
    The stream specified by *stream* is not valid.

Example

CELEBF88

```
/* CELEBF88

   This example sets a stream in error.

*/

#include <stdio.h>
#include <stdio_ext.h>

void main() {
   FILE *f;
   char filename[FILENAME_MAX] = "myfile.dat";

   f = fopen(filename,"wb");
   if (f == NULL) {
      perror("fopen failed\n");
      return;
   }

   if (ferror(f)) printf("The error indicator is set for the open stream\n");
   else printf("The error indicator is not set for the open stream\n");

   __fseterr(f);

   if (ferror(f)) printf("The error indicator is set for the open stream\n");
   else printf("The error indicator is not set for the open stream\n");

   return;
}
```

Output

```
The error indicator is not set for the open stream
The error indicator is set for the open stream
```

Related information

- "stdio.h" on page 68
- "stdio_ext.h" on page 70
- "clearerr() — Reset error and end of file (EOF)" on page 284
- "ferror() — Test for read and write errors" on page 503

__fsetlocking() — Set locking type

Standards

Standards / Extensions	C or C++	Dependencies
Language Environment	both	None

Format

```
#include <stdio.h>
#include <stdio_ext.h>

int __fsetlocking(FILE *stream, int type);
```

General description

The __fsetlocking() function allows the type of locking on an open stream to be controlled or queried by the application.

If *type* is FSETLOCKING_INTERNAL, subsequent stdio functions perform implicit locking around every operation on the given stream. This is the default system behavior.

If *type* is FSETLOCKING_BYCALLER, subsequent stdio functions assume that the caller is responsible for maintaining the integrity of the stream in the face of access by multiple threads. If only one thread is accessing the stream, nothing further needs to be done. If multiple threads are accessing the stream, you can use the flockfile(), funlockfile(), and ftrylockfile() functions to provide the appropriate serialization.

If *type* is FSETLOCKING_QUERY, the __fsetlocking() function returns the current locking type of the stream without changing it.

Usage notes

1. The _fsetlocking() function acts upon FILE* objects. It is possible to have the same physical file represented by multiple FILE* objects that are not recognized as being equivalent. For example, fopen() opens a file in thread A and sets the locking type as FSETLOCKING_INTERNAL. Then fopen() opens the same file in another thread B and sets the locking type as FSETLOCKING_BYCALLER. If both threads begin to write to their FILE* objects, the results are unpredictable.
2. The __fsetlocking() function impacts the behavior of all other stream operation functions. Using __fsetlocking() to modify the locking type while any of these stream operation functions are executing might produce undesirable behaviors, including hang conditions. You must make sure that __fsetlocking() is used only when no other functions are acting upon the stream.

Returned value

The __fsetlocking() function returns the locking type in effect before the call to __fsetlocking().

The __fsetlocking() function returns -1 if the stream or the locking type is not valid and sets errno to nonzero.

Error Code
  Description

EBADF
  The stream specified by *stream* is not valid.

EINVAL
  The locking type is not valid.

Example

CELEBF85

```
/* CELEBF85

   This example sets LOCKING_TYPE as locking type to a stream.

   It checks that the fopen() function is successful and that
   locking type is setted to the stream and queried from the stream.

*/

#include <stdio.h>
#include <stdio_ext.h>

int main(void)
{
   FILE *stream;
   int locktype;
   int nwrite;
   int buflen;
   char buf[5]="1234";

   if((stream = fopen("myfile.dat", "w+b")) != NULL )
   {
      locktype = __fsetlocking(stream,FSETLOCKING_BYCALLER);
      locktype = __fsetlocking(stream,FSETLOCKING_QUERY);
      if(locktype == FSETLOCKING_BYCALLER){
         printf("__fsetlocking succeeds! \
         Lock type is FSETLOCKING_BYCALLER\n");
      }
      buflen = strlen(buf);
      flockfile(stream);
      nwrite = fwrite(buf, 1, buflen, stream);
      if(nwrite < buflen){
         printf("fwrite did not return expected number of bytes\n");
      }
      funlockfile(stream);
   }
   return(0);
}
```

Output

```
__fsetlocking succeeds! Lock type is FSETLOCKING_BYCALLER
```

Related information

- “stdio.h” on page 68
- “stdio_ext.h” on page 70
- “ftrylockfile() — stdio locking” on page 665
- “flockfile()— stdio locking” on page 547
- “funlockfile() — stdio unlocking” on page 668

fsetpos() — Set file position

Standards

Standards / Extensions	C or C++	Dependencies
ISO C POSIX.1 XPG4 XPG4.2 C99 Single UNIX Specification, Version 3 Language Environment	both	

Format

```
#include <stdio.h>

int fsetpos(FILE *stream, const fpos_t *pos);

#define _OPEN_SYS_UNLOCKED_EXT 1
#include <stdio.h>

int fsetpos_unlocked(FILE *stream, const fpos_t *pos);
```

General description

The fsetpos() function moves the file position associated with *stream* to a new location within the file according to the value of the object pointed to by *pos*. The value of *pos* must be obtained by a call to the fgetpos() function. If successful, the fsetpos() function clears the EOF indicator, and cancels the effect of any previous ungetc() or ungetwc() function on the same stream.

If the call to the fsetpos() function is not valid, the call is treated as a flush, and the ungetc characters are discarded.

The fsetpos() function handles the double-byte character set (DBCS) state information for wide-oriented files. An fsetpos() function call to a position that no longer exists results in an error.

For text streams, the DBCS shift state is recalculated from the start of the record, which has a performance implication. The fsetpos() function repositions to the start of a multibyte character.

For binary streams, the DBCS shift state is set to the state saved by the fsetpos() function. If the record has been updated in the meantime, the shift state might be incorrect.

After the fsetpos() function call, the next operation on a stream in update mode can be an input or output operation.

Multivolume data sets performance: Using the fgetpos() and fsetpos() functions generally results in better repositioning performance compared to the ftell() and fseek() functions when working with multivolume data sets.

Large file support for MVS data sets, VSAM data sets, and z/OS UNIX files: The fsetpos() function implicitly supports operating on large files. Defining the _LARGE_FILES feature test macro is not required to use this function on large files.

Usage notes

1. Repositioning within a wide-oriented file and performing updates is strongly discouraged because it is not possible to predict if the update will overwrite part of a multibyte string or character, thereby invalidating subsequent data. For example, you could inadvertently add data that overwrites a shift-out. The following data expects the shift-out to be there, so is not valid if it is treated as if in the initial shift state. Repositioning to the end of the file and adding new data is safe. For information about wide-oriented streams, see *z/OS XL C/C++ Programming Guide*.
2. The fsetpos_unlocked() function is functionally equivalent to the fsetpos() function with the exception that it is not threadsafe. The fsetpos() function can safely be used in a multithreaded application if, and only if, it is called while the invoking thread owns the (FILE*) object, as is the case after a successful call to either the flockfile() or ftrylockfile() function.

Returned value

If successful in changing the current position of the file, the fsetpos() function returns 0.

If unsuccessful, the fsetpos() function returns nonzero and sets errno.

Special behavior for XPG4.2: If unsuccessful, the fsetpos() function returns -1 and sets errno to one of the following values:

Error Code
  Description

ESPIPE
  The underlying file type for the stream is a PIPE or a socket.

Example

```
/* This example opens a file called myfile.dat for reading.
   After performing input operations (not shown), it moves the file
   pointer to the beginning of the file and rereads the first byte.
 */
#include <stdio.h>

int main(void)
{

   FILE *stream;
   int retcode;
   fpos_t pos, pos1, pos2, pos3;
   char ptr[20];  /* existing file 'myfile.dat' has 20 byte records */

   /* Open file, get position of file pointer, and read first record */

   stream = fopen("myfile.dat", "rb");
   fgetpos(stream,&pos);
   pos1 = pos;
   if (!fread(ptr,sizeof(ptr),1,stream))
       printf("fread error\n");

   /* Perform a number of read operations.  The value of 'pos'
      changes if 'pos' is passed to fgetpos()                  */
⋮
   /* Re-set pointer to start of file and re-read first record  */

   fsetpos(stream,&pos1);
   if (!fread(ptr,sizeof(ptr),1,stream))
       printf("fread error\n");

   fclose(stream);
}
```

Related information

- "stdio.h" on page 68
- "fgetpos() — Get file position" on page 528
- "ftell() — Get current file position" on page 657
- "rewind() — Set file position to beginning of file" on page 1445
- "ungetc() — Push character onto input stream" on page 1958
- "ungetwc() — Push a wide character onto a stream" on page 1961

fstat() — Get status information about a file

Standards

Standards / Extensions	C or C++	Dependencies
POSIX.1 XPG4 XPG4.2 Single UNIX Specification, Version 3	both	

Format

```
#define _POSIX_SOURCE
#include <sys/stat.h>

int fstat(int fildes, struct stat *info);
```

General description

Gets status information about the file specified by the open file descriptor *fildes* and stores it in the area of memory indicated by the *info* argument. The status information is returned in a `stat` structure, as defined in the sys/stat.h header file. The elements of this structure are described in “stat() — Get file information” on page 1720.

Note: Environment variable _EDC_EOVERFLOW can be used to control behavior of fstat() with respect to detecting an EOVERFLOW condition for z/OS UNIX files. By default, fstat() will not set EOVERFLOW when the file size can not be represented correctly in structure pointed to by info. When _EDC_EOVERFLOW is set to YES, fstat() will check for an overflow condition.

Large file support for z/OS UNIX files: Large z/OS UNIX files are supported automatically for AMODE 64 C/C++ applications. AMODE 31 C/C++ applications must be compiled with the option LANGLVL(LONGLONG) and define the _LARGE_FILES feature test macro before any headers are included to enable this function to operate on z/OS UNIX files that are larger than 2 GB in size. File size and offset fields are enlarged to 63 bits in width. Therefore, any other function operating on the file is required to define the _LARGE_FILES feature test macro as well.

Returned value

If successful, fstat() returns 0.

If unsuccessful, fstat() returns -1 and sets errno to one of the following values:

Error Code
Description

EBADF
fildes is not a valid open file descriptor.

EINVAL
info contains a NULL.

EIO **Added for XPG4.2:** An I/O error occurred while reading from the file system.

EOVERFLOW

The file size in bytes or the number of blocks allocated to the file or the file serial number cannot be represented correctly in the structure pointed to by info.

Note: The fstat() function might fail with error code EOVERFLOW if large file support is not enabled. The environment variable _EDC_EOVERFLOW controls this behavior. If _EDC_EOVERFLOW is set to YES the new behavior will take place. The default for _EDC_EOVERFLOW is NO.

Example

CELEBF47

```
/* CELEBF47

   This example gets status information for the file called temp.file.
```

```
 */
#define _POSIX_SOURCE
#include <fcntl.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <unistd.h>
#undef _POSIX_SOURCE
#include <stdio.h>
#include <time.h>

main() {
  char fn[]="temp.file";
  struct stat info;
  int fd;

  if ((fd = creat(fn, S_IWUSR)) < 0)
    perror("creat() error");
  else {
    if (fstat(fd, &info) != 0)
      perror("fstat() error");
    else {
      puts("fstat() returned:");
      printf("  inode:   %d\n",   (int) info.st_ino);
      printf(" dev id:   %d\n",   (int) info.st_dev);
      printf("   mode:   %08x\n",       info.st_mode);
      printf("  links:   %d\n",         info.st_nlink);
      printf("    uid:   %d\n",   (int) info.st_uid);
      printf("    gid:   %d\n",   (int) info.st_gid);
      printf("created:   %s",           ctime(&info.st_createtime));
    }
    close(fd);
    unlink(fn);
  }
}
```

Output

```
fstat() returned:
  inode:   3057
 dev id:   1
   mode:   03000080
  links:   1
    uid:   25
    gid:   500
created:   Fri Jun 16 16:03:16 2006
```

Related information

- “sys/stat.h” on page 75
- “sys/types.h” on page 75
- “fcntl() — Control open file descriptors” on page 474
- “lstat() — Get status of file or symbolic link” on page 1022
- “open() — Open a file” on page 1152
- “stat() — Get file information” on page 1720

fstatvfs() — Get file system information

Standards

Standards / Extensions	C or C++	Dependencies
XPG4.2 Single UNIX Specification, Version 3	both	

Format

```
#define _XOPEN_SOURCE_EXTENDED 1
#include <sys/statvfs.h>

int fstatvfs(int fildes, struct statvfs *fsinfo);
```

General description

The fstatvfs() function obtains information about the file system containing the file referenced by *fildes* and stores it in the area of memory pointed to by the *fsinfo* argument.

The information is returned in a `statvfs` structure, as defined in the sys/statvfs.h header file. The elements of this structure are described in "statvfs() — Get file system information" on page 1724. If fstatvfs() successfully determines this information, it stores it in the area indicated by the *fsinfo* argument. The size of the buffer determines how much information is stored; data that exceeds the size of the buffer is truncated.

Returned value

If successful, fstatvfs() returns 0.

If unsuccessful, fstatvfs() returns -1 and sets errno to one of the following values:

Error Code
Description

EBADF
fildes is not a valid open file descriptor.

EINTR
A signal was caught during the execution of the function.

EIO An I/O error has occurred while reading the file system.

Example

```
#include <sys/statvfs.h>
#include <stdio.h>

main()
{
  char fn[]="temp.file";
  int fd;
  struct statvfs buf;

  if ((fd = creat(fn,S_IWUSR)) < 0)
    perror("creat() error");
  else {
    if (fstatvfs(fd, &buf) == -1)
      perror("fstatvfs() error");
    else {
      printf("each block is %d bytes big\n", buf.f_bsize);
      printf("there are %d blocks available\n", buf.f_bavail);
      printf("out of a total of %d in bytes,\n", buf.f_blocks);
      printf("that's %.0f bytes free out of a total of %.0f\n",
           ((double)buf.f_bavail * buf.f_bsize),
           ((double)buf.f_blocks * buf.f_bsize));
    }
    close(fd);
    unlink(fn);
  }
}
```

Output

```
each block is 4096 bytes big
there are 2089 blocks available
out of a total of 2400 in bytes,
that's 8556544 bytes free out of a total of 9830400
```

Related information

- “sys/statvfs.h” on page 75
- “chmod() — Change the mode of a file or directory” on page 272
- “chown() — Change the owner or group of a file or directory” on page 275
- “creat() — Create a new file or rewrite an existing one” on page 340
- “dup() — Duplicate an open file descriptor” on page 404
- “fcntl() — Control open file descriptors” on page 474
- “link() — Create a link to a file” on page 970
- “mknod() — Make a directory or file” on page 1074
- “open() — Open a file” on page 1152
- “pipe() — Create an unnamed pipe” on page 1179
- “read() — Read from a file or socket” on page 1376
- “rexec() — Execute commands one at a time on a remote host” on page 1447
- “time(),time64() — Determine current UTC time” on page 1878
- “unlink() — Remove a directory entry” on page 1962
- “utime() — Set file access and modification times” on page 1967
- “write() — Write data on a file or socket” on page 2085

fsync() — Write changes to direct-access storage

Standards

Standards / Extensions	C or C++	Dependencies
POSIX.1 XPG4 XPG4.2 Single UNIX Specification, Version 3	both	

Format

```
#define _POSIX1_SOURCE 2
#include <unistd.h>

int fsync(int fildes);
```

General description

Transfers all data for the file indicated by the open file descriptor *fildes* to the storage device associated with *fildes*. fsync() does not return until the transfer has completed, or until an error is detected.

Returned value

If successful, fsync() returns 0.

If unsuccessful, fsync() returns -1 and sets errno to one of the following values:

Error Code
Description

EBADF
fildes is not a valid open file descriptor.

EINVAL
The file is not a regular file.

Example

CELEBF48

```
/* CELEBF48 */
#define _POSIX_SOURCE
#include <fcntl.h>
#include <sys/stat.h>
#include <sys/types.h>
#include <unistd.h>
#undef _POSIX_SOURCE
#include <stdio.h>
#include <stdlib.h>

#define mega_string_len 250000

main() {
  char *mega_string;
  int  fd, ret;
  char fn[]="fsync.file";

  if ((mega_string = (char*) malloc(mega_string_len)) == NULL)
    perror("malloc() error");
  else if ((fd = creat(fn, S_IWUSR)) < 0)
    perror("creat() error");
  else {
    memset(mega_string, 's', mega_string_len);
    if ((ret = write(fd, mega_string, mega_string_len)) == -1)
      perror("write() error");
    else {
      printf("write() wrote %d bytes\n", ret);
      if (fsync(fd) != 0)
        perror("fsync() error");
      else if ((ret = write(fd, mega_string, mega_string_len)) == -1)
        perror("write() error");
      else
        printf("write() wrote %d bytes\n", ret);
    }
    close(fd);
    unlink(fn);
  }
}
```

Output

```
write() wrote 250000 bytes
write() wrote 250000 bytes
```

Related information

- "unistd.h" on page 82
- "open() — Open a file" on page 1152
- "write() — Write data on a file or socket" on page 2085

ftell() — Get current file position

Standards

Standards / Extensions	C or C++	Dependencies
ISO C POSIX.1 XPG4 XPG4.2 C99 Single UNIX Specification, Version 3 Language Environment	both	

Format

```
#include <stdio.h>

long int ftell(FILE *stream);

#define _OPEN_SYS_UNLOCKED_EXT 1
#include <stdio.h>

long int ftell_unlocked(FILE *stream);
```

General description

The ftell() function obtains the current value of the file position indicator for the stream pointed to by *stream*.

Behavior for binary streams: ANSI states that the ftell() function returns relative byte offsets from the beginning of the file for binary files. Under z/OS XL C/C++, this is true except for record-oriented files that have variable length records. For these types of files, the ftell() function returns an encoded offset.

If you want to use relative-byte offsets for these types of files, you can either open the file with the BYTESEEK fopen() function option or set the _EDC_BYTE_SEEK environment variable before opening.

Behavior for text streams: The ftell() function returns an encoded offset for text streams.

Behavior for record I/O: For files opened for record I/O using the `type=record` open mode parameter, the ftell() function returns the relative record offset of the current file position from the beginning of the file. All offset values are given in terms of records.

Behavior for blocked I/O: For files opened for blocked I/O using the `type=blocked` open mode parameter, the ftell() function returns the relative block offset of the current file position from the beginning of the file. All offset values are given in terms of blocks.

Multivolume data sets performance: Using the fgetpos() and fsetpos() functions generally results in better repositioning performance compared to the ftell() and fseek() functions when working with multivolume data sets.

Large file support for MVS data sets, VSAM data sets, and z/OS UNIX files: For AMODE 31 C/C++ applications, the ftell() function accepts a signed 4-byte offset and therefore cannot be used to directly or relatively position to offsets beyond 2 GB - 1. To avoid repositioning limitations, AMODE 31 C/C++ applications should define the _LARGE_FILES feature test macro before any headers are included and replace the ftell() function with the ftello() function. For AMODE 64 C/C++ applications, there are no restrictions on using the ftell() function with large files. The AMODE 64 version automatically accepts a signed 8-byte offset.

Usage note

The ftell_unlocked() function is functionally equivalent to the ftell() function with the exception that it is not threadsafe. The ftell() function can safely be used in a multithreaded application if, and only if, it is called while the invoking thread owns the (FILE*) object, as is the case after a successful call to either the flockfile() or ftrylockfile() function.

Returned value

If successful, the ftell() function returns the calculated value.

If unsuccessful, the ftell() function returns -1 and sets errno to a positive value.

Special behavior for XPG4.2: If unsuccessful, the ftell() function returns -1 and sets errno to one of the following values:

Error Code
  Description

EOVERFLOW
  For ftell(), the current file offset cannot be represented correctly in an object of type long.

  Note: Environment variable _EDC_EOVERFLOW can be used to control behavior of the ftell() function with respect to detecting an EOVERFLOW condition for z/OS UNIX files. By default, the ftell() function will not set EOVERFLOW when the file offset cannot be represented correctly. When _EDC_EOVERFLOW is set to YES, the ftell() function will check for an overflow condition.

ESPIPE
  The underlying file type for the stream is a PIPE or a socket.

Example

```
/* This example opens the file myfile.dat for reading.
   The current file pointer position is stored in the variable pos.
 */
#include <stdio.h>

int main(void)
{
   FILE *stream
   long int pos;

   stream = fopen("myfile.dat", "rb");

   /* The value returned by ftell can be used by fseek()
      to set the file pointer if 'pos' is not -1       */
   if ((pos = ftell(stream)) != EOF)
      printf("Current position of file pointer found\n");
   fclose(stream);
}
```

Related information

- For information about BYTESEEK or _EDC_BYTE_SEEK, see *z/OS XL C/C++ Programming Guide*.
- For information about calling the ftell() function after an ungetc() or ungetwc() function call, see "ungetc() — Push character onto input stream" on page 1958 and "ungetwc() — Push a wide character onto a stream" on page 1961.
- For additional usage information about the ftell() function with respect to MVS data sets, VSAM data sets, or z/OS UNIX files, see *z/OS XL C/C++ Programming Guide*.
- "stdio.h" on page 68
- "fgetpos() — Get file position" on page 528
- "fopen() — Open a file" on page 565
- "fseek() — Change file position" on page 639
- "fsetpos() — Set file position" on page 649
- "ftello() — Get current file position"

ftello() — Get current file position

Standards

Standards / Extensions	C or C++	Dependencies
Single UNIX Specification, Version 2 Single UNIX Specification, Version 3 Language Environment	both	OS/390 V2R10

Format

```
#define _XOPEN_SOURCE 500
#include <stdio.h>

off_t ftello(FILE *stream);

#define _OPEN_SYS_UNLOCKED_EXT 1
#include <stdio.h>

off_t ftello_unlocked(FILE *stream);
```

General description

The ftello() function obtains the current value of the file position indicator for the stream pointed to by *stream*.

Behavior for binary streams: ANSI states that the ftello() function returns relative byte offsets from the beginning of the file for binary files. Under z/OS XL C/C++, this is true except for record-oriented files that have variable length records. For these types of files, the ftello() function returns an encoded offset.

If you want to use relative-byte offsets for these types of files, you can either open the file with the BYTESEEK fopen() function option or set the _EDC_BYTE_SEEK environment variable before opening.

Behavior for text streams: The ftello() function returns an encoded offset for text streams.

Behavior for record I/O: For files opened for record I/O using the `type=record` open mode parameter, the ftello() function returns the relative record offset of the current file position from the beginning of the file. All offset values are given in terms of records.

Behavior for blocked I/O: For files opened for blocked I/O using the `type=blocked` open mode parameter, the ftello() function returns the relative block offset of the current file position from the beginning of the file. All offset values are given in terms of blocks.

Multivolume data sets performance: Using the fgetpos() and fsetpos() functions generally results in better repositioning performance compared to the ftello() and fseeko() functions when working with multivolume data sets.

Large file support for MVS data sets, VSAM data sets, and z/OS UNIX files: For AMODE 31 C/C++ applications, the ftello() function accepts a signed 4-byte offset and therefore cannot be used to directly or relatively position to offsets beyond 2 GB - 1. To avoid repositioning limitations, AMODE 31 C/C++ applications should define the _LARGE_FILES feature test macro before any headers are included. For AMODE 64 C/C++ applications, there are no restrictions on using the ftello() function with large files. The AMODE 64 version automatically accepts a signed 8-byte offset.

Usage notes

1. The ftello_unlocked() function is functionally equivalent to the ftello() function with the exception that it is not threadsafe. The ftello() function can safely be used in a multithreaded application if, and only if, it is called while the invoking thread owns the (FILE*) object, as is the case after a successful call to either the flockfile() or ftrylockfile() function.

Returned value

If successful, the ftello() function returns the calculated value.

If unsuccessful, the ftello() function returns (off_t)-1 and sets errno to one of the following values:

Error Code
: **Description**

EBADF
: The file descriptor underlying stream is not an open file descriptor.

EOVERFLOW
: The current file offset cannot be represented correctly in an object of type off_t.

ESPIPE
: The file descriptor underlying stream is associated with a pipe or FIFO.

Example

```
/* This example opens the file myfile.dat for reading.
   The current file pointer position is stored in the variable
   pos.
 */
#define _LARGE_FILES 1
#include <stdio.h>

int main(void)
{
   FILE *stream
```

```
    off_t pos;

   stream = fopen("/myfile.dat", "rb");


    /* The value returned by ftello() can be used by fseeko()
       to set the file pointer if 'pos' is not -1       */

    if ((pos = ftello(stream)) != -1LL)
       printf("Current position of file pointer found\n");
    fclose(stream);
 }
```

Related information

- For information about BYTESEEK or _EDC_BYTE_SEEK, see *z/OS XL C/C++ Programming Guide*.
- For information about calling the ftello() function after an ungetc() or ungetwc() function call, see "ungetc() — Push character onto input stream" on page 1958 and "ungetwc() — Push a wide character onto a stream" on page 1961.
- For additional usage information about the ftello() function with respect to MVS data sets, VSAM data sets, or z/OS UNIX files, see *z/OS XL C/C++ Programming Guide*.
- "stdio.h" on page 68
- "fgetpos() — Get file position" on page 528
- "fopen() — Open a file" on page 565
- "fseek() — Change file position" on page 639
- "fsetpos() — Set file position" on page 649
- "ftell() — Get current file position" on page 657
- "ungetc() — Push character onto input stream" on page 1958
- "ungetwc() — Push a wide character onto a stream" on page 1961

ftime() — Set the date and time

Standards

Standards / Extensions	C or C++	Dependencies
XPG4.2 Single UNIX Specification, Version 3	both	

Format

```
#define _XOPEN_SOURCE_EXTENDED 1
#include <sys/timeb.h>

int ftime(struct timeb *tp);
```

General description

The ftime() function sets the time and millitm members of the timeb structure pointed to by *tp* to contain seconds and milliseconds, respectively, of the current time in seconds since 00:00:00 Coordinated Universal Time (UTC), January 1, 1970.

Note: The ftime() function has been moved to the Legacy Option group in Single UNIX Specification, Version 3 and may be withdrawn in a future version. The time() function is preferred for portability.

Returned value

If successful, ftime() returns 0.

If overflow occurs, ftime() returns -1.

[1]

Related information

- "limits.h" on page 39
- "sys/timeb.h" on page 75
- "ctime(), ctime64() — Convert time to character string" on page 359
- "ctime_r(), ctime64_r() — Convert time value to date and time character string" on page 362
- "time(),time64() — Determine current UTC time" on page 1878

ftok() — Generate an interprocess communication (IPC) key

Standards

Standards / Extensions	C or C++	Dependencies
XPG4.2 Single UNIX Specification, Version 3	both	

Format

```
#define _XOPEN_SOURCE_EXTENDED 1
#include <sys/ipc.h>

key_t ftok(const char *path, int id);
```

General description

The ftok() function returns a key based on *path* and *id* that is usable in subsequent calls to msgget(), semget(), and shmget(). The *path* argument must be the path name of an existing file that the process is able to stat().

The ftok() function returns the same key value for all paths that name the same file, when called with the same *id* value. If a different *id* value is given, or a different file is given, a different key is returned. Only the low-order 8-bits of *id* are significant, and must be nonzero.

Returned value

If successful, ftok() returns a key.

If unsuccessful, ftok() returns -1 and sets errno to one of the following values:

Error Code
    Description

EACCES
    Search permission is denied for a component of the path prefix.

[1] Overflow occurs when the current time in seconds since 00:00:00 UTC, January 1, 1970 exceeds the capacity of the *time* member of the timeb structure pointed to by *tp*. The *time* member is type time_t.

EINVAL
The low-order 8-bits of *id* are zero.

ELOOP
Too many symbolic links were encountered in resolving path.

ENAMETOOLONG
One of the following error conditions exists:
- The length of the *path* argument exceeds **PATH_MAX** or a path name component is longer than **NAME_MAX**.
- The path name resolution of a symbolic link produced an intermediate result whose length exceeds **PATH_MAX**.

ENOENT
A component of *path* does not name an existing file or *path* is an empty string.

ENOTDIR
A component of the path prefix is not a directory.

Related information

- "sys/ipc.h" on page 73
- "msgget() — Get message queue" on page 1105
- "semget() — Get a set of semaphores" on page 1485
- "shmget() — Get a shared memory segment" on page 1602
- "stat() — Get file information" on page 1720

ftruncate() — Truncate a file

Standards

Standards / Extensions	C or C++	Dependencies
POSIX.1a XPG4.2 Single UNIX Specification, Version 3	both	

Format

```
#define _POSIX1_SOURCE 2
#include <unistd.h>

int ftruncate(int fildes, off_t length);
```

General description

The ftruncate() function truncates the file indicated by the open file descriptor *fildes* to the indicated *length*. *fildes* must be a regular file that is open for writing. If the file size exceeds *length*, any extra data is discarded. If the file size is smaller than length, bytes between the old and new lengths are read as zeros. A change to the size of the file has no impact on the file offset.

Special behavior for XPG4.2: If the ftruncate() function would cause the file size to exceed the soft file size limit for the process, ftruncate() will fail and a SIGXFSZ signal will be generated for the process.

If successful, the ftruncate() function marks the `st_ctime` and `st_mtime` fields of the file.

If unsuccessful, the ftruncate() function leaves the file unchanged.

Large file support for z/OS UNIX files: Large z/OS UNIX files are supported automatically for AMODE 64 C/C++ applications. AMODE 31 C/C++ applications must be compiled with the option LANGLVL(LONGLONG) and define the _LARGE_FILES feature test macro before any headers are included to enable this function to operate on z/OS UNIX files that are larger than 2 GB in size. File size and offset fields are enlarged to 63 bits in width. Therefore, any other function operating on the file is required to define the _LARGE_FILES feature test macro as well.

Returned value

If successful, the ftruncate() function returns 0.

If unsuccessful, the ftruncate() function returns -1 and sets errno to one of the following values:

Error Code
Description

EBADF
fildes is not a valid open file descriptor.

EFBIG
The length argument was greater than the maximum file size.

EINTR
Added for XPG4.2: A signal was caught during execution.

EINVAL
fildes does not refer to a regular file, it is opened read-only, or the length specified is incorrect.

EIO **Added for XPG4.2:** An I/O error occurred while reading from or writing to a file system.

EROFS
The file resides on a read-only file system.

Example

CELEBF49

```
/* CELEBF49 */
#define _POSIX_SOURCE
#include <fcntl.h>
#include <sys/stat.h>
#include <sys/types.h>
#include <unistd.h>
#undef _POSIX_SOURCE
#include <stdio.h>
#include <stdlib.h>

#define string_len 1000

main() {
  char *mega_string;
  int  fd, ret;
  char fn[]="write.file";
  struct stat st;

  if ((mega_string = (char*) malloc(string_len)) == NULL)
```

```
    perror("malloc() error");
  else if ((fd = creat(fn, S_IWUSR)) < 0)
    perror("creat() error");
  else {
    memset(mega_string, '0', string_len);
    if ((ret = write(fd, mega_string, string_len)) == -1)
      perror("write() error");
    else {
      printf("write() wrote %d bytes\n", ret);
      fstat(fd, &st);
      printf("the file has %ld bytes\n", (long) st.st_size);
      if (ftruncate(fd, 1) != 0)
        perror("ftruncate() error");
      else {
        fstat(fd, &st);
        printf("the file has %ld bytes\n", (long) st.st_size);
      }
    }
    close(fd);
    unlink(fn);
  }
}
```

Output

```
write() wrote 1000 bytes
the file has 1000 bytes
the file has 1 bytes
```

Related information

- “unistd.h” on page 82
- “open() — Open a file” on page 1152
- “truncate() — Truncate a file to a specified length” on page 1917

ftrylockfile() — stdio locking

Standards

Standards / Extensions	C or C++	Dependencies
Single UNIX Specification, Version 3	both	z/OS V1R8

Format

```
#define _UNIX03_SOURCE
#include <stdio.h>

int ftrylockfile(FILE *file);
```

General description

This function provides explicit application-level locking of stdio (FILE*) objects. The flockfile() family of functions can be used by a thread to delineate a sequence of I/O statements that are executed as a unit.

If the (**FILE***) object specified by the ftrylockfile() function is available, ownership is granted to the thread for the (FILE*) object and the internal lock count is increased. If the thread has previously been granted ownership, the internal lock count is increased. If another thread has been granted ownership, ftrylockfile() does not grant ownership to the calling thread and returns a non-zero value. ftrylockfile() is a non-blocking version of flockfile().

The internal lock count allows matching calls to flockfile() (or successful calls to ftrylockfile()) and funlockfile() to be nested.

z/OS consideration: The flockfile() family of functions acts upon FILE * objects. It is possible to have the same physical file represented by multiple FILE * objects that are not recognized as being equivalent. For example, fopen() opens a file and open() opens the same file, and then fdopen() creates a FILE * object. In this case, locking the first FILE * does not prevent the second FILE * from also being locked and used.

Returned value

The ftrylockfile() function returns zero for success and non-zero to indicate that the lock cannot be acquired.

Error Code
Definition

EBADF
The input (**FILE ***) object is not valid.

EBUSY
The input (**FILE ***) object is locked by another thread.

Note: It is the application's responsibility to prevent deadlock (or looping). For example, deadlock (or looping) may occur if a (FILE *) object is closed, or a thread is terminated, before relinquishing all locked (FILE *) objects.

Related information

- "flockfile()— stdio locking" on page 547
- "funlockfile() — stdio unlocking" on page 668

ftw() — Traverse a file tree

Standards

Standards / Extensions	C or C++	Dependencies
XPG4 XPG4.2 Single UNIX Specification, Version 3	both	

Format

```
#define _XOPEN_SOURCE
#include <ftw.h>

int ftw(const char *path,
        int (*fn)(const char *, const struct stat *, int),
        int ndirs);
```

General description

The ftw() function recursively descends the directory hierarchy rooted in *path*. For each object in the hierarchy, ftw() calls the function pointed to by *fn*, passing it a pointer to a NULL-terminated string containing the name of the object, a pointer to a *stat* structure containing information about the object, and an integer. Possible values of the integer, defined in the <ftw.h> header, are:

FTW_D
for a directory

FTW_DNR
for a directory that cannot be read

FTW_F
for a file

FTW_SL
for a symbolic link

FTW_NS
for an object other than a symbolic link on which stat() could not be successfully executed. If the object is a symbolic link, and stat() failed, it is unspecified whether ftw() passes FTW_SL or FTW_NS to the user-supplied function.

If the integer is FTW_DNR, descendants of that directory will not be processed. If the integer is FTW_NS, the stat structure will contain undefined values. An example of an object that would cause FTW_NS to be passed to the function pointed to by *fn* would be a file in a directory with read but without execute (search) permission.

The ftw() function visits a directory before visiting any of its descendants.

The ftw() function uses at most one file descriptor for each level in the tree.

The argument *ndirs* should be in the range of 1 to OPEN_MAX.

The tree traversal continues until the tree is exhausted, an invocation of *fn* returns a nonzero value, or some other error, other than [EACCES], is detected within ftw().

The *ndirs* argument specifies the maximum number of directory streams or file descriptors or both available for use by ftw() while traversing the tree. When ftw() returns it closes any directory streams and file descriptors it uses not counting any opened by the application-supplied *fn* function.

Large file support for z/OS UNIX files: Large z/OS UNIX files are supported automatically for AMODE 64 C/C++ applications. AMODE 31 C/C++ applications must be compiled with the option LANGLVL(LONGLONG) and define the _LARGE_FILES feature test macro before any headers are included to enable this function to operate on z/OS UNIX files that are larger than 2 GB in size. File size and offset fields are enlarged to 63 bits in width. Therefore, any other function operating on the file is required to define the _LARGE_FILES feature test macro as well.

Returned value

If the tree is exhausted, ftw() returns 0. If the function pointed to by *fn* returns a nonzero value, ftw() stops its tree traversal and returns whatever value was returned by the function pointed to by *fn()*.

If ftw() detects an error, it returns -1 and sets errno to one of the following values. All other errnos returned by ftw() are unchanged.

Error Code
Description

EACCES
Search permission is denied for any component of *path* or read permission is denied for *path*.

EINVAL
The value of the *ndirs* argument is not valid.

ELOOP
Too many symbolic links were encountered.

ENAMETOOLONG
One of the following error conditions exists:

- Path name resolution of a symbolic link produced an intermediate result whose length exceeds PATH_MAX.
- The length of *path* exceeds PATH_MAX, or a path name component is longer than PATH_MAX.

ENOENT
A component of *path* does not name an existing file or *path* is an empty string.

ENOTDIR
A component of *path* is not a directory.

Related information

- “ftw.h” on page 32
- “longjmp() — Restore stack environment” on page 1005
- “lstat() — Get status of file or symbolic link” on page 1022
- “malloc() — Reserve storage block” on page 1031
- “nftw() — Traverse a file tree” on page 1143
- “opendir() — Open a directory” on page 1158
- “readdir() — Read an entry from a directory” on page 1382
- “siglongjmp() — Restore the stack environment and signal mask” on page 1640
- “stat() — Get file information” on page 1720

funlockfile() — stdio unlocking

Standards

Standards / Extensions	C or C++	Dependencies
Single UNIX Specification, Version 3	both	z/OS V1R8

Format

```
#define _UNIX03_SOURCE
#include <stdio.h>

void funlockfile(FILE *file);
```

General description

This function provides explicit application-level unlocking of stdio (FILE*) objects. The flockfile() family of functions can be used by a thread to delineate a sequence of I/O statements that are executed as a unit.

The funlockfile() function reduces the internal lock count. When the count is reduced to zero, the funlockfile() function relinquishes the ownership granted to the thread, of a (FILE *) object. If a call to funlockfile() is made by a thread which has not been granted ownership of a (FILE *) object, the call is ignored and the lock count is not reduced

The internal lock count allows matching calls to flockfile() (or successful calls to ftrylockfile()) and funlockfile() to be nested.

z/OS consideration: The flockfile() family of functions acts upon FILE * objects. It is possible to have the same physical file represented by multiple FILE * objects that are not recognized as being equivalent. For example, fopen() opens a file and open() opens the same file, and then fdopen() creates a FILE * object. In this case, locking the first FILE * does not prevent the second FILE * from also being locked and used.

Returned value

None.

Notes:

1. Because the funlockfile() function returns void, no error information can be returned. If an invalid (FILE *) object is input, it will be ignored.
2. It is the application's responsibility to prevent deadlock (or looping). For example, deadlock (or looping) may occur if a (FILE *) object is closed, or a thread is terminated, before relinquishing all locked (FILE *) objects.

Related information

- "flockfile()— stdio locking" on page 547
- "ftrylockfile() — stdio locking" on page 665

fupdate() — Update a VSAM record

Standards

Standards / Extensions	C or C++	Dependencies
Language Environment	both	

Format

```
#include <stdio.h>

size_t fupdate(const void *buffer, size_t size, FILE *stream);

#define _OPEN_SYS_UNLOCKED_EXT 1
#include <stdio.h>

size_t fupdate_unlocked(const void *buffer, size_t size, FILE *stream);
```

General description

Replaces the last record read from the VSAM cluster pointed to by *stream*, with the contents of *buffer* for a length of *size*. See "Performing VSAM I/O Operations" in *z/OS XL C/C++ Programming Guide* for details.

The fupdate() function can be used *only* with a VSAM data set opened in update mode (`rb+/r+b`, `ab+/a+b`, or `wb+/w+b`) with the `type=record` option.

The fupdate() function can only be used after an fread() call has been performed and before any other operation on that file pointer. For example, if you need to acquire the file position using ftell() or fgetpos(), you can do it either before the fread() or after the fupdate(). An fread() after an fupdate() retrieves the next updated record.

To avoid infringing on the user's name space, this nonstandard function has two names. One name is prefixed with two underscore characters, and one name is not. The name without the prefix underscore characters is exposed only when you use LANGLVL(EXTENDED).

To use this function, you must either invoke the function using its external entry point name (that is, the name that begins with two underscore characters), or compile with LANGLVL(EXTENDED). When you use LANGLVL(EXTENDED) any relevant information in the header is also exposed.

fupdate_unlocked() is functionally equivalent to fupdate() with the exception that it is not thread-safe. This function can safely be used in a multithreaded application if and only if it is called while the invoking thread owns the (FILE*) object, as is the case after a successful call to either the flockfile() or ftrylockfile() function.

KSDS or KSDS PATH: The size of the record can be changed by a call to fupdate(). If the size is greater than the existing record size but less than or equal to the maximum record length of the file, a call to fupdate() will lengthen the record up to the maximum record length of the file. If the size is greater than the maximum record length of the file, the record is truncated and errno is set. If the size is less than or equal to the existing record length, all size bytes of the record are written, and no padding or overlaying occurs. The records will be shortened and not partially updated.

ESDS, ESDS PATH, or RRDS: The size of a record cannot be changed by a call to fupdate(). If you call fupdate() with *size* smaller than the size of the existing record, *size* bytes of the record are updated; the remaining bytes are unchanged, and the record length remains unchanged.

The key of reference (the prime key if opened as a cluster, the alternative index key if opened as a path) cannot be changed by an update. If a data set is opened as a path, the prime key cannot be changed by an update. For RRDS files, the `buffer` must be an RRDS record structure, which includes an `rrds_key`.

Returned value

If successful, fupdate() returns the size of the updated record.

If the update operation is not successful, fupdate() returns 0.

Example

CELEBF50

```
/* CELEBF50 */
#include <stdio.h>
```

```
int main(void)
{
   FILE *stream;
   struct record { char name[20];
                   char address[40];
                   int age;
                 } buffer;
   int vsam_rc, numread;

   stream = fopen("DD:MYCLUS", "rb+,type=record");
   numread = fread(&buffer, 1, sizeof(buffer), stream);
   /*  ... Update fields in the record ...  */
   vsam_rc = fupdate(&buffer, sizeof(buffer), stream);
}
```

Related information

- “Performing VSAM I/O Operations” in *z/OS XL C/C++ Programming Guide*
- “stdio.h” on page 68
- “fdelrec() — Delete a VSAM record” on page 486
- “flocate() — Locate a VSAM record” on page 544

fwide() — Set stream orientation

Standards

Standards / Extensions	C or C++	Dependencies
ISO C Amendment ISO/ANSI C++ C99 Single UNIX Specification, Version 3 Language Environment	both	z/OS V1R7

Format

```
#define _MSE_PROTOS
#include <stdio.h>
#include <wchar.h>

int fwide(FILE *stream, int mode);

#define _OPEN_SYS_UNLOCKED_EXT 1
#include <wchar.h>

int fwide_unlocked(FILE *stream, int mode);
```

General description

fwide() determines the orientation of the stream pointed to by *stream*. If *mode* is greater than 0, the function attempts to make the stream wide-oriented. If *mode* is less than 0, the function attempts to make the stream byte-oriented. Otherwise, *mode* is 0 and the function does not alter the orientation of the stream, rather the function returns the current orientation of the stream.

If the orientation of the stream has already been determined, fwide() will not change it.

Streams opened as `type=record` or `type=blocked` do not have orientation.

VSAM data sets and CICS transient data queues do not have orientation. Use of fwide() against streams referring to VSAM data sets or CICS transient data queues will be unsuccessful.

An application wishing to check for error situations should set errno to 0, then call fwide(), then check errno. If errno is non-zero assume an error has occurred.

fwide_unlocked() is functionally equivalent to fwide() with the exception that it is not thread-safe. This function can safely be used in a multithreaded application if and only if it is called while the invoking thread owns the (FILE*) object, as is the case after a successful call to either the flockfile() or ftrylockfile() function.

Special considerations for C++: The interaction of fwide() and a C++ I/O stream is undefined.

Usage notes

1. The runtime library does not prevent using byte-oriented I/O functions on a wide-oriented stream, using wide-oriented I/O functions on a byte-oriented stream, or any other mixed orientation usage. The behavior of an application doing so is undefined. As a result, the orientation of a stream reported by fwide() might not be consistent with the I/O functions that are being used. The stream orientation first set using fwide() itself, or through the first I/O operation on the stream is what will be returned. For example, if fwide() is used to set the orientation as byte-oriented, but only wide-oriented I/O functions are used on the stream, the orientation of the stream remains byte-oriented even though no mixing of I/O functions has occurred.

Returned value

If successful, fwide() returns a value greater than 0 if the stream has wide-orientation after the call. It returns a value less than 0 if the stream has byte-orientation, or 0 if the stream has no orientation after the call.

When unsuccessful, fwide() returns 0 and sets errno to one of the following:

EBADF – The stream specified by stream was not valid.

EINVAL – The stream specified by stream was opened as `type=record` or `type=blocked`, or the stream refers to a VSAM data set or CICS transient data queue.

Related information

- “stdio.h” on page 68
- “wchar.h” on page 85

fwprintf(), swprintf(), wprintf() — Format and write wide characters

Standards

Standards / Extensions	C or C++	Dependencies
ISO C Amendment C99 Single UNIX Specification, Version 3 Language Environment	both	

Format

Non-XPG4:

```
#include <wchar.h>

int fwprintf(FILE * __restrict__ stream, const wchar_t * __restrict__ format, ...);
int swprintf(wchar_t * __restrict__ wcs, size_t n, const wchar_t * __restrict__ format, ...);
int wprintf(const wchar_t * __restrict__ format, ...);

#define _OPEN_SYS_UNLOCKED_EXT 1
#include <wchar.h>

int fwprintf_unlocked(FILE * __restrict__ stream, const wchar_t * __restrict__ format, ...);
int wprintf_unlocked(const wchar_t * __restrict__ format, ...);
```

XPG4:

```
#define _XOPEN_SOURCE
#define _MSE_PROTOS
#include <wchar.h>

int fwprintf(FILE * __restrict__ stream, const wchar_t * __restrict__ format, ...);
int swprintf(wchar_t *__restrict__ wcs, size_t n, const wchar_t * __restrict__ format, ...);
int wprintf(const wchar_t * __restrict__ format, ...);

#define _OPEN_SYS_UNLOCKED_EXT 1
#include <wchar.h>

int fwprintf_unlocked(FILE * __restrict__ stream, const wchar_t * __restrict__ format, ...);
int wprintf_unlocked(const wchar_t * __restrict__ format, ...);
```

General description

The fwprintf(), swprintf() and wprintf() functions are equivalent to fprintf(), sprintf() and printf(), respectively, except for the following:

- For swprintf(), the argument *wcs* specifies an array of type `wchar_t` into which the generated output is to be written, rather than an array of type char.
- The argument *format* specifies an array of type `wchar_t` that describes how subsequent arguments are converted for output, rather than an array of type char.
- %c without an `l` prefix means an int arg is to be converted to `wchar_t`, as if mbtowc() were called, and then written.
- %c with `l` prefix means a `wint_t` is converted to `wchar_t` and then written.
- %s without an `l` prefix means a character array containing a multibyte character sequence is to be converted to an array of `wchar_t` and then written. The conversion will take place as if mbrtowc() were called repeatedly.
- %s with `l` prefix means an array of `wchar_t` will be written. The array is written up to but not including the terminating NULL character, unless the precision specifies a shorter output.

For swprintf(), a NULL wide character is written at the end of the wide characters written; the NULL wide character is not counted as part of the returned sum. If copying takes place between objects that overlap, the behavior is undefined.

fwprintf_unlocked() family is functionally equivalent to fwprintf() family with the exception that it is not thread-safe. This function can safely be used in a multithreaded application if and only if it is called while the invoking thread owns the (FILE*) object, as is the case after a successful call to either the flockfile() or ftrylockfile() function.

Special behavior for XPG4: If you define any feature test macro specifying XPG4 behavior before the statement in your program source file to include the wchar header, then you must also define the _MSE_PROTOS feature test macro to make the declaration of the fwprintf(), swprintf() or wprintf() function in the wchar header available when you compile your program. Please see Table 2 on page 4 for a list of XPG4 and other feature test macros.

Note: The fwprintf() and wprintf() functions have a dependency on the level of the Enhanced ASCII Extensions. See "Enhanced ASCII support" on page 2113 for details.

Returned value

If successful, fwprintf(), wprintf(), and swprintf() return the number of wide characters written, not counting the terminating NULL wide character.

If unsuccessful, a negative value is returned.

If *n* or more wide characters were requested to be written, swprintf() returns a negative value and sets errno to indicate the error.

Related information

- "wchar.h" on page 85
- "fprintf(), printf(), sprintf() — Format and write data" on page 588
- "vfwprintf(), vswprintf(), vwprintf() — Format and write wide characters of a STDARG argument list" on page 1985

__fwritable() — Determine if a stream is open for writing

Standards

Standards / Extensions	C or C++	Dependencies
Language Environment	both	

Format

```
#include <stdio.h>
#include <stdio_ext.h>

int __fwritable(FILE *stream);
```

General description

The __fwritable() function determines if the specified stream has been opened for writing.

Returned value

The __fwritable() function returns nonzero if the stream is opened for writing. Otherwise, the __fwritable() function returns 0. If an error has occurred, __fwritable() returns 0 and sets errno to nonzero.

An application wishing to check for error situations should set errno to 0, then call __fwritable(), and then check errno. If errno is nonzero, assume that an error has occurred.

Error Code
  Description

EBADF
  The stream specified by *stream* is not valid.

Example

CELEBF89

```
/* CELEBF89

   This example writes and reads data to a file while querying the
   stream for information about data in the I/O buffer.

*/

#include <stdio.h>
#include <stdio_ext.h>

void main() {
   FILE *f;
   char filename[FILENAME_MAX] = "myfile.dat";
   char data[128] = "There are 34 bytes in this buffer\n";
   int datalen = strlen(data);
   size_t n = 0;

   f = fopen(filename,"wb+");
   if (f == NULL) {
      perror("fopen() failed\n");
      return;
   }

   if (__fwritable(f)) printf("Writing is allowed on the open stream\n");
   if (__freadable(f)) printf("Reading is allowed on the open stream\n");

   n = fputs(data,f);
   if (n == EOF) {
      perror("fputs() failed\n");
      return;
   }

   n = __fpending(f);
   printf("There are %d bytes in the buffer pending to be written\n", n);

   if (__fwriting(f)) printf("The last operation on the stream was a write\n");

   rewind(f);

   n = fgetc(f);

   n = __freadahead(f);
   printf("There are %d bytes remaining to be read from the buffer\n", n);

   if (__freading(f)) printf("The last operation on the stream was a read\n");

   return;
}
```

Output

```
Writing is allowed on the open stream
Reading is allowed on the open stream
There are 34 bytes in the buffer pending to be written
The last operation on the stream was a write
There are 33 bytes remaining to be read from the buffer
The last operation on the stream was a read
```

Related information

- "stdio.h" on page 68
- "stdio_ext.h" on page 70
- "fopen() — Open a file" on page 565

- "freopen() — Redirect an open file" on page 620
- "__freadable() — Determine if a stream is open for reading" on page 612
- "__freading() — Determine if last operation on stream is a read operation" on page 616
- "__fwriting() — Determine if last operation on stream is a write operation" on page 677

fwrite() — Write items

Standards

Standards / Extensions	C or C++	Dependencies
ISO C POSIX.1 XPG4 XPG4.2 C99 Single UNIX Specification, Version 3 Language Environment	both	

Format

```
#include <stdio.h>

size_t fwrite(const void * __restrict__buffer, size_t size, size_t count, FILE * __restrict__stream);

#define _OPEN_SYS_UNLOCKED_EXT 1
#include <stdio.h>

size_t fwrite_unlocked(const void * __restrict__buffer, size_t size,
                       size_t count, FILE * __restrict__stream);
```

General description

Writes up to *count* items of size *size* from the location pointed to by *buffer* to the stream pointed to by *stream*.

When you are using fwrite() for record I/O output, set *size* to 1 and *count* to the length of the record to be written. You can only write one record at a time when you are using record I/O. Any string longer than the record length is truncated at the record length. A flush or reposition is required before a subsequent read.

When you are using fwrite() for blocked I/O output, set *size* to 1 and *count* to the length of the block to be written. You can only write one block at a time when you are using blocked I/O. Any string longer than the block length is truncated at the block length.

Because fwrite() may buffer output before writing it out to the stream, data from prior fwrite() calls may be lost where a subsequent call to fwrite() causes a failure when the buffer is written to the stream.

fwrite() has the same restriction as any write operation for a read immediately following a write or a write immediately following a read. Between a write and a subsequent read, there must be an intervening flush or reposition. Between a read and a subsequent write, there must also be an intervening flush or reposition unless an EOF has been reached.

fwrite_unlocked() is functionally equivalent to fwrite() with the exception that it is not thread-safe. This function can safely be used in a multithreaded application if and only if it is called while the invoking thread owns the (FILE*) object, as is the case after a successful call to either the flockfile() or ftrylockfile() function.

Returned value

fwrite() returns the number of items that were successfully written.

This number can be smaller than *count* only if a write error occurred.

Example

CELEBF51

```
/* CELEBF51

   This example writes NUM long integers to a stream in binary
   format.
   It checks that the &fopen. function is successful and that
   100 items are written to the stream.

 */
#include <stdio.h>
#define  NUM  100

int main(void)
{
   FILE *stream;
   long list[NUM];
   int numwritten, number;

   if((stream = fopen("myfile.dat", "w+b")) != NULL )
   {
     for (number = 0; number < NUM; ++number)
       list[number] = number;
     numwritten = fwrite(list, sizeof(long), NUM, stream);
     printf("number of long characters written is %d\n",numwritten);
   }
   else
     printf("fopen error\n");
}
```

Related information

- “stdio.h” on page 68
- “fopen() — Open a file” on page 565
- “freopen() — Redirect an open file” on page 620
- “fread() — Read items” on page 610

__fwriting() — Determine if last operation on stream is a write operation

Standards

Standards / Extensions	C or C++	Dependencies
Language Environment	both	

Format

```
#include <stdio.h>
#include <stdio_ext.h>

int __fwriting(FILE *stream);
```

```
#define _OPEN_SYS_UNLOCKED 1
#include <stdio.h>
#include <stdio_ext.h>

int __fwriting_unlocked(FILE *stream);
```

General description

The __fwriting() function determines if the last operation on the specified stream is a write operation or if the specified stream is open for write-only or append-only.

The __fwrite_unlocked() function is equivalent to the __fwriting() function with the exception that it is not thread-safe. This function can be safely used in a multithreaded application if it is called while the invoking thread owns the (FILE *) object, such as after a successful call to either the flockfile() or ftrylockfile() function.

Returned value

The __fwriting() functions return nonzero when the last operation was a write operation or the stream is open for write-only or append-only. Otherwise, the __fwriting() function returns 0. If an error has occurred, the __fwriting() functions return 0 and set errno to nonzero.

An application wishing to check for error situations should set errno to 0, then call __fwriting(), and then check errno. If errno is nonzero, assume that an error has occurred.

Error Code
    Description

EBADF
    The stream specified by *stream* is not valid.

Example

CELEBF89

```
/* CELEBF89

   This example writes and reads data to a file while querying the
   stream for information about data in the I/O buffer.

*/

#include <stdio.h>
#include <stdio_ext.h>

void main() {
   FILE *f;
   char filename[FILENAME_MAX] = "myfile.dat";
   char data[128] = "There are 34 bytes in this buffer\n";
   int datalen = strlen(data);
   size_t n = 0;

   f = fopen(filename,"wb+");
   if (f == NULL) {
      perror("fopen() failed\n");
      return;
   }

   if (__fwritable(f)) printf("Writing is allowed on the open stream\n");
   if (__freadable(f)) printf("Reading is allowed on the open stream\n");
   n = fputs(data,f);
   if (n == EOF) {
      perror("fputs() failed\n");
      return;
   }

   n = __fpending(f);
   printf("There are %d bytes in the buffer pending to be written\n", n);

   if (__fwriting(f)) printf("The last operation on the stream was a write\n");

   rewind(f);

   n = fgetc(f);

   n = __freadahead(f);
   printf("There are %d bytes remaining to be read from the buffer\n", n);

   if (__freading(f)) printf("The last operation on the stream was a read\n");

   return;
}
```

Output

```
Writing is allowed on the open stream
Reading is allowed on the open stream
There are 34 bytes in the buffer pending to be written
The last operation on the stream was a write
There are 33 bytes remaining to be read from the buffer
The last operation on the stream was a read
```

Related information

- "stdio.h" on page 68
- "stdio_ext.h" on page 70
- "fopen() — Open a file" on page 565
- "freopen() — Redirect an open file" on page 620
- "__freadable() — Determine if a stream is open for reading" on page 612
- "__freading() — Determine if last operation on stream is a read operation" on page 616
- "__fwritable() — Determine if a stream is open for writing" on page 674

fwscanf(), swscanf(), wscanf() — Convert formatted wide-character input

Standards

Standards / Extensions	C or C++	Dependencies
ISO C Amendment ISO/ANSI C++ C99 Single UNIX Specification, Version 3 Language Environment	both	z/OS V1R7

Format

Non-XPG4:

```
#define _MSE_PROTOS
#include <stdio.h>
#include <wchar.h>

int fwscanf(FILE *__restrict__ stream,
     const wchar_t *__restrict__ format, ... );

int swscanf(const wchar_t * __restrict__ wcs,
     const wchar_t * __restrict__ format, ...);

int wscanf(const wchar_t *__restrict__ format, ... );

#define _OPEN_SYS_UNLOCKED_EXT 1
#include <wchar.h>

int fwscanf_unlocked(FILE *__restrict__ stream,
     const wchar_t *__restrict__ format, ... );
int wscanf_unlocked(const wchar_t *__restrict__ format, ... );
```

XPG4 and swscanf():

```
#define _XOPEN_SOURCE
#define _MSE_PROTOS
#include <wchar.h>

int swscanf(const wchar_t *wcs, const wchar_t *format, ...);
```

General description

The fwscanf(), swscanf(), and wscanf() functions are equivalent to fscanf(), scanf(), and sscanf() respectively, except for the following:

- The argument *wcs* specifies an array of type wchar_t from which the input is to be obtained, rather than an array of type char.
- The *format* argument specifies an array of type wchar_t that describes the admissible input sequences and how they are to be converted for assignment, rather than an array of type char.
- %c with no *l* prefix means one or more (depending on precision) wchar_t is converted to multibyte characters and copied to the character array pointed to by the corresponding argument.
- %c with the *l* prefix means one or more (depending on precision) wchar_t is copied to the array of wchar_t pointed to by the corresponding argument.
- %s with no *l* prefix means a sequence of non-white wchar_t will be converted and copied, including the terminating NULL character, to the character array pointed to by the corresponding argument.
- %s with the *l* prefix means an array of wchar_t will be copied, including the terminating NULL wide-character, to the array of wchar_t pointed to by the corresponding argument.
- %[with no *l* prefix means a sequence of non-white wchar_t will be converted and copied, including the terminating NULL character, to the character array pointed to by the corresponding argument.
- %[with the *l* prefix means an array of wchar_t will be copied, including the terminating NULL wide-character, to the array of wchar_t pointed to by the corresponding argument.

Note: Reaching the end of a wide-character string is equivalent to reaching the end of a char string for the fscanf() and scanf() functions. If copying takes place between objects that overlap, the behavior is undefined.

fwscanf_unlocked() family is functionally equivalent to fwscanf() family with the exception that they are not thread-safe. These functions can safely be used in a multithreaded application if and only if they are called while the invoking thread owns the (FILE*) object, as is the case after a successful call to either the flockfile() or ftrylockfile() function.

Special behavior for XPG4 and swscanf(): If you define any feature test macro specifying XPG4 behavior before the statement in your program source file to include the wchar header, then you must also define the _MSE_PROTOS feature test macro to make the declaration of the swscanf() function in the wchar header available when you compile your program. Please see Table 2 on page 4 for a list of XPG4 and other feature test macros.

Returned value

If successful, they either return the number of input items assigned, which can be fewer than provided for, or 0 in the event of an early matching failure. If an input failure occurs before any conversion, EOF is returned.

Related information

- “stdio.h” on page 68
- “wchar.h” on page 85
- “fscanf(), scanf(), sscanf() — Read and format data” on page 625

gai_strerror() — Address and name information error description

Standards

Standards / Extensions	C or C++	Dependencies
RFC2553 Single UNIX Specification, Version 3	both	z/OS V1R4

Format

```
#define _OPEN_SYS_SOCK_IPV6
#include <netdb.h>

char *gai_strerror(int ecode);
```

SUSV3:

```
#define _POSIX_C_SOURCE 200112L
#include <netdb.h>
const char *gai_strerror(int ecode);
```

General description

The gai_strerror() function returns a pointer to a text string describing the error value returned by a failure return from either the getaddrinfo() or getnameinfo() function. If the *ecode* is not one of the EAI_xxx values from the <netdb.h> header, then gai_strerror() returns a pointer to a string indicating an unknown error.

Subsequent calls to gai_strerror() will overwrite the buffer containing the text string.

Returned value

When successful, gai_strerror() returns a pointer to a string describing the error. Upon failure, gai_strerror() will return NULL and set errno to one of the following:

Error Code
Description

ENOMEM
Insufficient memory to allocate buffer for text string describing the error.

Related information

- "getaddrinfo() — Get address information" on page 684
- "getnameinfo() — Get name information" on page 738
- "netdb.h" on page 49

gamma() — Calculate gamma function

Standards

Standards / Extensions	C or C++	Dependencies
SAA XPG4 XPG4.2	both	

Format

```
#include <math.h>

double gamma(double x);
```

Compiler option: LANGLVL(SAA), LANGLVL(SAAL2), or LANGLVL(EXTENDED)

General description

gamma() provides the same function as lgamma(), including the use of *signgam*. Use of lgamma() instead of gamma() is suggested by XPG4.2.

Note:

This function is kept for historical reasons. It was part of the Legacy Feature in Single UNIX Specification, Version 2, but has been withdrawn and is not supported as part of Single UNIX Specification, Version 3. New applications should use lgamma() instead of gamma().

If it is necessary to continue using this function in an application written for Single UNIX Specification, Version 3, define the feature test macro _UNIX03_WITHDRAWN before including any standard system headers. The macro exposes all interfaces and symbols removed in Single UNIX Specification, Version 3.

Related information

- "math.h" on page 44
- "lgamma(), lgammaf(), lgammal() — Log gamma function" on page 964
- "__signgam() — Return signgam reference" on page 1648

gcvt() — Convert double to string

Standards

Standards / Extensions	C or C++	Dependencies
XPG4.2 Single UNIX Specification, Version 3	both	

Format

```
#define _XOPEN_SOURCE_EXTENDED 1
#include <stdlib.h>

char *gcvt(double x, int ndigit, char *buf);
```

General description

The gcvt() function converts double floating-point argument values to floating-point output strings. The gcvt() function has been extended to determine the floating-point format (hexadecimal floating-point or IEEE Binary Floating-Point) of double argument values by using __isBFP().

z/OS XL C/C++ formatted output functions, including the gcvt() function, convert IEEE Binary Floating-Point infinity and NaN argument values to special infinity and NaN floating-point number output sequences. See "*fprintf* Family of Formatted Output Functions" on fprintf(), printf(), sprintf() — Format and write data for a description of the special infinity and NaN output sequences.

The gcvt() function converts *x* to a NULL-terminated string (similar to the %g format of "fprintf(), printf(), sprintf() — Format and write data" on page 588) in the array pointed to by *buf* and returns *buf*. It produces *ndigit* significant digits (limited to an unspecified value determined by the precision of a double) in %f if possible, or %e (scientific notation) otherwise. A minus sign is included in the returned string if *value* is less than 0. A radix character is included in the returned string if *value* is not a whole number. Trailing zeros are suppressed where *value* is not a whole number. The radix character is determined by the current locale. If "setlocale() — Set locale" on page 1552 has not been called successfully, the default locale, "POSIX", is used. The default locale specifies a period (.) as the radix character. The LC_NUMERIC category determines the value of the radix character within the current locale.

Note: This function has been moved to the Legacy Option group in Single UNIX Specification, Version 3 and may be withdrawn in a future version. The sprintf() function is preferred for portability.

Returned value

If successful, gcvt() returns the character equivalent of *x* as specified above.

If unsuccessful, gcvt() returns NULL.

Related information

- "stdlib.h" on page 70
- "ecvt() — Convert double to string" on page 418
- "fcvt() — Convert double to string" on page 485
- "__isBFP() — Determine application floating-point format" on page 904

getaddrinfo() — Get address information

Standards

Standards / Extensions	C or C++	Dependencies
RFC2553	both	z/OS V1R4

Format

```
#define _OPEN_SYS_SOCK_IPV6
#include <sys/socket.h>
#include <netdb.h>

int getaddrinfo(const char *nodename,
                const char *servname,
                const struct addrinfo *hints,
                struct addrinfo **res);
```

SUSV3:

```
#define _POSIX_C_SOURCE 200112L
#include <sys/socket.h>
#include <netdb.h>
int getaddrinfo(const char *__restrict__ nodename,
           const char *__restrict__ servname,
           const struct addrinfo *__restrict__ hints,
           struct addrinfo **__restrict__ res);
```

General description

The getaddrinfo() function translates the name of a service location (for example, a host name) and/or service name and returns a set of socket addresses and associated information to be used in creating a socket with which to address the specified service.

The *nodename* and *servname* arguments are either pointers to null-terminated strings or null pointers. One or both of these two arguments must be specified as a non-null pointer.

The format of a valid name depends on the protocol family or families. If a specific family is not given and the name could be interpreted as valid within multiple supported families, the function attempts to resolve the name in all supported families. When no errors are detected, all successful results will be returned.

If the *nodename* argument is not null, it can be a descriptive name or it can be an address string. If the specified address family is AF_INET, AF_INET6, or AF_UNSPEC, valid descriptive names include host names. If the specified address family is AF_INET or AF_UNSPEC, address strings using standard dot notation as specified in inet_addr() are valid. If the specified address family is AF_INET6 or AF_UNSPEC, standard IPv6 text forms described in inet_pton() are valid. In addition, scope information can be appended to the descriptive name or the address string using the format *nodename%scope* information. Scope information can be either an interface name or the numeric representation of an interface index suitable for use on this system.

If *nodename* is not null, the requested service location is named by *nodename*; otherwise, the requested service location is local to the caller.

If *servname* is null, the call returns network-level addresses for the specified *nodename*. If *servname* is not null, it is a null-terminated character string identifying the requested service. This can be either a descriptive name or a numeric representation suitable for use with the address family or families. If the specified address family is AF_INET, AF_INET6, or AF_UNSPEC, the service can be specified as a string specifying a decimal port number.

If the argument *hints* is not null, it refers to a structure containing input values that may direct the operation by providing options and by limiting the returned information to a specific socket type, address family and/or protocol. In the hints structure every member other than *ai_flags*, *ai_family*, *ai_socktype*, and *ai_protocol* must be zero or a null pointer. A value of AF_UNSPEC for *ai_family* means that the caller will accept any protocol family. A value of zero for *ai_socktype* means that the caller will accept any socket type. A value of zero for ai_protocol means that the caller will accept any protocol. If hints is a null pointer, the behavior must be as if it referred to a structure containing the value zero for the *ai_flags*, *ai_socktype*, and *ai_protocol* fields, and AF_UNSPEC for the *ai_family* field.

The *ai_flags* member to which the hints argument points can be set to 0 or be the bitwise inclusive OR of one or more of the following values:
- AI_PASSIVE
- AI_CANONNAME
- AI_NUMERICHOST
- AI_NUMERICSERV
- AI_V4MAPPED
- AI_ALL
- AI_ADDRCONFIG
- AI_EXTFLAGS

If the AI_PASSIVE bit is set in the *ai_flags* member of the hints structure, then the caller plans to use the returned socket address structure in a call to bind(). In this case, if the *nodename* argument is a null pointer, then the IP address portion of the socket address structure will be set to INADDR_ANY for an IPv4 address or IN6ADDR_ANY_INIT for an IPv6 address. If the AI_PASSIVE bit is not set in the *ai_flags* member of the hints structure, then the returned socket address structure will be ready for a call to connect() (for a connection-oriented protocol) or either connect(), sendto(), or sendmsg() (for a connectionless protocol). In this case, if the *nodename* argument is a null pointer, then the IP address portion of the socket address structure will be set to the loopback address.

If the AI_CANONNAME bit is set in the *ai_flags* member of the hints structure, then upon successful return the *ai_canonname* member of the first addrinfo structure in the linked list will point to a null-terminated string containing the canonical name of the specified nodename.

If the AI_NUMERICHOST bit is set in the *ai_flags* member of the hints structure, then a non-null *nodename* string must be a numeric host address string. Otherwise an error code of EAI_NONAME is returned. This flag prevents any type of name resolution service (for example, the DNS) from being called.

If the AI_NUMERICSERV flag is specified then a non-null *servname* string must be a numeric port string. Otherwise an error code EAI_NONAME is returned. This flag prevents any type of name resolution service (for example, NIS+ from being invoked.

If the AI_V4MAPPED flag is specified along with the AF field with the value of AF_INET6, or a value of AF_UNSPEC when IPv6 is supported on the system, then the caller will accept IPv4-mapped IPv6 addresses. When the AI_ALL flag is not also specified and no IPv6 addresses are found, then a query is made for IPv4 addresses. If any IPv4 addresses are found, they are returned as IPv4-mapped IPv6 addresses.

If the AF field does not have a value of AF_INET6 or the AF field contains AF_UNSPEC but IPv6 is not supported on the system, this flag is ignored.

When the AF field has a value of AF_INET6 and AI_ALL is set, the AI_V4MAPPED flag must also be set to indicate that the caller will accept all addresses (IPv6 and IPv4-mapped IPv6 addresses). When the AF field has a value of AF_UNSPEC when the system supports IPv6 and AI_ALL is set, the caller accepts IPv6 addresses and either IPv4 (if AI_V4MAPPED is not set) or IPv4-mapped IPv6 (if AI_V4MAPPED is set) addresses. A query is first made for IPv6 addresses and if successful, the IPv6 addresses are returned. Another query is then made for IPv4 addresses and any found are returned as IPv4 addresses (if AI_V4MAPPED was not set) or as IPv4-mapped IPv6 addresses (if AI_V4MAPPED was set). If the AF field does not have the value of AF_INET6, or the value of AF_UNSPEC when the system supports IPv6, the flag is ignored.

If the AI_ADDRCONFIG flag is specified then a query for IPv6 address records should occur only if the node has at least one IPv6 source address configured. A query for IPv4 address records will always occur, whether or not any IPv4 addresses are configured. The loopback address is not considered for this case as valid as a configured sources address.

If the AI_EXTFLAGS flag is specified then getaddrinfo() will look for the values stored in the extended flags field called *ai_eflags* in the addrinfo structure. The flags stored in the *ai_eflags* field are only meaningful if the AI_EXTFLAGS flag is set in the *ai_flags* field of the addrinfo data structure. By default, AI_EXTFLAGS is not set in the *ai_flags* field. If AI_EXTFLAGS is set in the *ai_flags* field, and the *ai_eflags* extended flags field is 0 (zero) or undefined, then AI_EXTFLAGS is ignored. The *ai_eflags* field can be set to any of the following:

- IPV6_PREFER_SRC_HOME - prefer home address as source
- IPV6_PREFER_SRC_COA - prefer care-of address as source
- IPV6_PREFER_SRC_TMP - prefer temporary address as source
- IPV6_PREFER_SRC_PUBLIC - prefer public address as source
- IPV6_PREFER_SRC_CGA - prefer CGA address as source
- IPV6_PREFER_SRC_NONCGA - prefer a non-CGA address as source

These flags can be combined into a flag set to express complex address preferences, but some can result in a contradictory flag set. For example, the following flags are mutually exclusive:

- IPV6_PREFER_SRC_HOME and IPV6_PREFER_SRC_COA
- IPV6_PREFER_SRC_TMP and IPV6_PREFER_SRC_PUBLIC
- IPV6_PREFER_SRC_CGA and IPV6_PREFER_SRC_NONCGA

All of the information returned by getaddrinfo() is dynamically allocated: the addrinfo structures, and the socket address structures and canonical node name strings pointed to by the addrinfo structures. To return this information to the system the function freeaddrinfo() is called.

Usage notes

1. If the caller handles only TCP and not UDP, for example, then the *ai_protocol* member of the hints structure should be set to IPPROTO_TCP when getaddrinfo() is called.
2. If the caller handles only IPV4 and not IPv6, then the *ai_family* member of the hints structure should be set to AF_INET when getaddrinfo() is called.
3. Scope information is only pertinent to IPv6 link-local addresses. It is ignored for resolved IPv4 addresses and IPv6 addresses that are not link-local addresses.

Returned value

When successful, getaddrinfo() returns 0 and a pointer to a linked list of one or more addrinfo structures through the res argument. The caller can process each addrinfo structure in this list by following the ai_next pointer, until a null pointer is encountered. In each returned addrinfo structure the three members *ai_family*, *ai_socktype*, and *ai_protocol* are the corresponding arguments for a call to the socket() function. In each addrinfo structure the *ai_addr* member points to a filled-in socket address structure whose length is specified by the *ai_addrlen* member. Upon failure, getaddrinfo() returns a non-zero error code. The error codes are as follows:

Error Code
Description

EAI_AGAIN
The name specified by the Node_Name or Service_Name parameter could not be resolved within the configured time interval, or the resolver address space has not been started. The request can be retried later.

EAI_BADEXTFLAGS
The extended flags parameter had an incorrect setting.

EAI_BADFLAGS
The flags parameter had an incorrect setting.

EAI_FAIL
An unrecoverable error occurred.

EAI_FAMILY
The family parameter had an incorrect setting.

EAI_MEMORY
A memory allocation failure occurred during an attempt to acquire an Addr_Info structure.

EAI_NONAME
One of the following conditions occurred:

1. The name does not resolve for the specified parameters. At least one of the Name or Service operands must be specified.
2. The request name parameter is valid, but it does not have a record at the name server.

EAI_SERVICE
The service that was passed was not recognized for the specified socket type.

EAI_SOCKTYPE
The intended socket type was not recognized.

EAI_SYSTEM
A system error occurred.

For more information about the above return codes, see *z/OS Communications Server: IP and SNA Codes*.

Related information

- "netdb.h" on page 49
- "sys/socket.h" on page 74
- "freeaddrinfo() — Free addrinfo storage" on page 620
- "gai_strerror() — Address and name information error description" on page 681
- "getnameinfo() — Get name information" on page 738

getc(), getchar() — Read a character

Standards

Standards / Extensions	C or C++	Dependencies
ISO C POSIX.1 XPG4 XPG4.2 C99 Single UNIX Specification, Version 3	both	

Format

```
#include <stdio.h>

int getc(FILE *stream);
int getchar(void);
```

General description

Reads a single character from the current *stream* position and advances the *stream* position to the next character. The getchar() function is identical to getc(stdin).

The getc() and fgetc() functions are identical. However, getc() and getchar() are provided in a highly efficient macro form. For performance purposes, it is recommended that the macro forms be used rather than the functional forms or fgetc(). By default, stdio.h provides the macro versions of these functions.

However, to get the functional forms, do one or more of the following:
- For C only: do *not* include stdio.h.
- Specify `#undef`, for example, `#undef getc`
- Surround the call statement by parentheses, for example, `(getc)`

getc() and getchar() are not supported for files opened with `type=record` or `type=blocked`.

getc() and getchar() have the same restriction as any read operation for a read immediately following a write or a write immediately following a read. Between a write and a subsequent read, there must be an intervening flush or reposition. Between a read and a subsequent write, there must also be an intervening flush or reposition unless an EOF has been reached.

If the application is not multithreaded, then setting the _ALL_SOURCE_NO_THREADS feature test macro may improve performance of the application, because it allows use of the inline version of this function.

Special behavior for POSIX: In a multithreaded C application that uses POSIX(ON), in the presence of the feature test macro, _OPEN_THREADS, these macros are in an #undef status because they are not thread-safe.

Note: Because the getc() macro reevaluates its input argument more than once, you should never pass a stream argument that is an expression with side effects.

Returned value

getc() and getchar() return the character read.

A returned value of EOF indicates either an error or an EOF condition. If a read error occurs, the error indicator is set. If an EOF is encountered, the EOF indicator is set.

Use ferror() or feof() to determine whether an error or an EOF condition occurred. Note that EOF is only reached when an attempt is made to read past the last byte of data. Reading up to and including the last byte of data does *not* turn on the EOF indicator.

Example

CELEBG02

```
/* CELEBG02

   This example gets a line of input from the stdin stream.
   You can also use getc(stdin) instead of &getchar. in the for
   statement to get a line of input from stdin.

 */
#include <stdio.h>

#define LINE 80

int main(void)
{
  char buffer[LINE+1];
  int i;
  int ch;

  printf( "Please enter string\n" );

  /* Keep reading until either:
     1. the length of LINE is exceeded  or
     2. the input character is EOF  or
     3. the input character is a new-line character
   */

  for ( i = 0; ( i < LINE ) && (( ch = getchar()) != EOF) &&
               ( ch !='\n' ); ++i )
    buffer[i] = ch;

  buffer[i] = '\0';  /* a string should always end with '\0' ! */

  printf( "The string is %s\n", buffer );
}
```

Output

```
Please enter string
hello world
The string is hello world
```

Related information

- "stdio.h" on page 68
- "fgetc() — Read a character" on page 526
- "gets() — Read a string" on page 769
- "putc(), putchar() — Write a character" on page 1347
- "ungetc() — Push character onto input stream" on page 1958

getc_unlocked(), getchar_unlocked(), putc_unlocked(), putchar_unlocked() — Stdio with explicit client locking

Standards

Standards / Extensions	C or C++	Dependencies
Single UNIX Specification, Version 3	both	z/OS V1R8

Format

```
#define _UNIX03_SOURCE
#include <stdio.h>

int getc_unlocked(FILE *stream);
int getchar_unlocked(void);
int putc_unlocked(int c, FILE *stream);
int putchar_unlocked(int c);
```

General description

Versions of the functions getc(), getchar(), putc(), and putchar() respectively named getc_unlocked(), getchar_unlocked(), putc_unlocked(), and putchar_unlocked() are functionally equivalent to the original versions, with the exception that they are not thread-safe. These functions may safely be used in a multi-threaded program if and only if they are called while the invoking thread owns the (FILE*) object, as is the case after a successful call to the flockfile() or ftrylockfile() functions.

getc_unlocked, getchar_unlocked, putc_unlocked, putchar_unlocked are provided in a highly efficient macro form. For performance purposes, it is recommended that the macro forms be used rather than the functional forms. By default, stdio.h provides the macro versions of these functions.

However, to get the functional forms, do one or more of the following:
- Surround the call statement by parentheses, for example, (getc_unlocked)
- Specify #undef, for example, #undef getc_unlocked
- For C only: do *not* include stdio.h.

getc_unlocked, getchar_unlocked, putc_unlocked, putchar_unlocked are not supported for files that are opened with type=record or type=blocked.

getc_unlocked, getchar_unlocked, putc_unlocked, putchar_unlocked have the same restrictions as any read or write operation for a read immediately following a write or a write immediately following a read. Between a write and a subsequent read, there must be an intervening flush or reposition. Between a read and a subsequent write, there must also be an intervening flush or reposition unless an EOF has been reached.

Note: Because the macro forms of these functions reevaluate their input arguments more than once, you must not pass an argument that is an expression with side effects.

Returned value

See “getc(), getchar() — Read a character” on page 688 and “putc(), putchar() — Write a character” on page 1347.

Related information

getclientid() — Get the identifier for the calling application

Standards

Standards / Extensions	C or C++	Dependencies
z/OS UNIX	both	

Format

```
#define _OPEN_SYS_SOCK_EXT
#include <sys/socket.h>
#include <sys/types.h>

int getclientid(int domain, struct clientid *clientid);
```

General description

The getclientid() function call returns the identifier by which the calling application is known to the TCP/IP address space. The *clientid* can be used in the givesocket() and takesocket() calls. However, this function is supplied for use by existing programs that depend on the address space name returned. Even for these programs it is recommended that the name be saved for its later use and the __getclientid() function be issued to reconstruct the clientid structure for use by givesocket() and takesocket().

Parameter
Description

domain The address domain requested.

clientid The pointer to a *clientid* structure to be filled.

The clientid structure is filled in by the call and returned as follows:

The clientid structure:

```
struct clientid {
   int domain;
   union  {
     char name[8];
     struct {
       int NameUpper;
       pid_t pid;
     } c_pid;
   } c_name;
```

```
          char subtaskname[8];

          struct  {
            char type;
            union  {
               char specific[19];
               struct  {
                 char unused[3];
                 int SockToken;
               } c_close;
            } c_func;
          } c_reserved;
       };
```

Element
Description

domain
The input *domain* value returned in the domain field of the clientid structure.

c_name.name
The application program's address space name, left-justified and padded with blanks.

subtaskname
The calling program's task identifier.

c_reserved
Specifies binary zeros.

Returned value

If successful, getclientid() returns 0.

If unsuccessful, getclientid() returns -1 and sets errno to one of the following values:

Error Code
Description

EFAULT
Using the *clientid* parameter as specified would result in an attempt to access storage outside the caller's address space, or storage not modifiable by the caller.

Related information

- “sys/socket.h” on page 74
- “sys/types.h” on page 75
- “__getclientid() — Get the PID identifier for the calling application”

__getclientid() — Get the PID identifier for the calling application

Standards

Standards / Extensions	C or C++	Dependencies
z/OS UNIX	both	

Format

```
#define _OPEN_SYS_SOCK_EXT
#include <sys/socket.h>
#include <sys/types.h>

int __getclientid(int domain, struct clientid *clientid);
```

General description

The __getclientid() function call returns the process identifier (PID) by which the calling application is known to the TCP/IP address space. The *clientid* is used in the givesocket() and takesocket() calls. Use the __getclientid() function call to transfer sockets between the caller and the selected application. The __getclientid() function provides improved performance and integrity over the getclientid() function for applications that use the output of __getclientid() as input clientids for givesocket() and takesocket().

Parameter
Description

domain The address domain requested.

clientid The pointer to a *clientid* structure to be filled.

The clientid structure:

```
      struct clientid {
         int domain;
         union  {
           char name[8];
           struct {
             int NameUpper;
             pid_t pid;
           } c_pid;
         } c_name;
         char subtaskname[8];

         struct  {
           char type;
           union  {
              char specific[19];
              struct  {
                char unused[3];
                int SockToken;
              } c_close;
           } c_func;
         } c_reserved;
      };
```

Element
Description

domain The input *domain* value returned in the domain field of the clientid structure.

c_pid.pid
Is the label in the *clientid* structure that is filled in by the function call to the PID of the requester (caller of __getclientid()). It should be left as set because it is used by the takesocket() and givesocket() functions.

subtaskname
Blanks

c_reserved
Binary zeros

Returned value

If successful, __getclientid() returns 0.

If unsuccessful, __getclientid() returns -1 and sets errno to one of the following values:

Error Code
Description

EFAULT
Using the *clientid* parameter as specified would result in an attempt to access storage outside the caller's address space, or storage not modifiable by the caller.

Related information

- "sys/socket.h" on page 74
- "sys/types.h" on page 75
- "getclientid() — Get the identifier for the calling application" on page 691
- "givesocket() — Make the specified socket available" on page 804
- "takesocket() — Acquire a socket from another program" on page 1812

getcontext() — Get user context

Standards

Standards / Extensions	C or C++	Dependencies
XPG4.2 Single UNIX Specification, Version 3	both	POSIX(ON)

Format

```
#define _XOPEN_SOURCE_EXTENDED 1
#include <ucontext.h>

int getcontext(ucontext_t *ucp);
```

General description

The getcontext() function initializes the structure pointed to by *ucp* to the current user context of the calling process. The **ucontext_t** type that *ucp* points to defines the user context and includes the contents of the calling process's machine registers, the signal mask, and the current execution stack. A subsequent call to setcontext() restores the saved context and returns control to a point in the program corresponding to the getcontext() call. Execution resumes as if the getcontext() call had just returned. The return value from getcontext() is the same regardless of whether the return is from the initial invocation or using a call to setcontext().

The context created by getcontext() may be modified by the makecontext() function. Refer to makecontext for details.

getcontext() is similar in some respects to sigsetjmp() (and setjmp() and _setjmp()). The getcontext()–setcontext() pair, the sigsetjmp()–siglongjmp() pair, the setjmp()–longjmp() pair, and the _setjmp()–_longjmp() pair cannot be intermixed. A context saved by getcontext() should be restored only by setcontext().

Note: Some compatibility exists with siglongjmp(), so it is possible to use siglongjmp() from a signal handler to restore a context created with getcontext(), but it is not recommended.

Portable applications should not modify or access the **uc_mcontext** member of **ucontext_t**. A portable application cannot assume that context includes any process-wide static data, possibly including errno. Users manipulating contexts should take care to handle these explicitly when required.

This function is supported only in a POSIX program.

The **<ucontext.h>** header file defines the **ucontext_t** type as a structure that includes the following members:

```
mcontext_t       uc_mcontext     A machine-specific representation
                                 of the saved context.
ucontext_t      *uc_link         Pointer to the context that will
                                 be resumed when this context returns.
sigset_t         uc_sigmask      The set of signals that are blocked
                                 when this context is active.
stack_t          uc_stack        The stack used by this context.
```

Special behavior for C++: If getcontext() and setcontext() are used to transfer control in a z/OS XL C++ program, the behavior in terms of the destruction of automatic objects is undefined. This applies to both z/OS XL C++ and z/OS XL C/C++ ILC modules. The use of getcontext() and setcontext() in conjunction with try(), catch(), and throw() is also undefined.

Do not issue getcontext() in a C++ constructor or destructor, since the saved context would not be usable in a subsequent setcontext() or swapcontext() after the constructor or destructor returns.

Special behavior for XPLINK-compiled C/C++: Restrictions concerning setjmp.h and ucontext.h:

1. All XPLINK programs compiled with the V2R10 or later C compilers that are to run with Language Environment V2R10 or later libraries and use the **jmp_buf**, **sigjmp_buf** or **ucontext_t** types must not be compiled with C headers from Language Environment V2R9 or earlier.
2. Non-XPLINK functions compiled with any level of Language Environment headers must not define **jmp_buf**, **sigjmp_buf** or **ucontext_t** data items and pass them to XPLINK functions that call getcontext(), longjmp(), _longjmp(), setjmp(), _setjmp(), setcontext(), sigsetjmp(), or swapcontext() with these passed-in data items.
3. When __XPLINK__ is defined, the Language Environment V2R10 and later headers define a larger **jmp_buf**, **sigjmp_buf** or **ucontext_t**area that is required by setjmp(), getcontext(), and related functions when they are called from an XPLINK routine. If __XPLINK__ is not defined, the Language Environment V2R10 and later headers define a shorter **jmp_buf**, **sigjmp_buf** or **ucontext_t** area. The Language Environment headers before V2R10 also define the shorter version of these data areas. If an XPLINK function calls setjmp(), getcontext() or similar functions with a short **jmp_buf**, **sigjmp_buf** or **ucontext_t** area, a storage overlay or program check may occur when the C library tries to store past the end of the passed-in (too short) data area.

Returned value

If successful, getcontext() returns 0.

If unsuccessful, getcontext() returns -1.

There are no errno values defined.

Example

This example saves the context in main with the getcontext() statement. It then returns to that statement from the function `func` using the setcontext() statement. Since getcontext() always returns 0 if successful, the program uses the variable `x` to determine if getcontext() returns as a result of setcontext() or not.

```
/* This example shows the usage of getcontext() and setcontext().    */

#define _XOPEN_SOURCE_EXTENDED 1
#include <stdio.h>
#include <ucontext.h>

void func(void);

int  x = 0;
ucontext_t context, *cp = &context;

int main(void) {

  getcontext(cp);
  if (!x) {
    printf("getcontext has been called\n");
    func();
  }
  else {
    printf("setcontext has been called\n");
  }

}

void func(void) {

  x++;
  setcontext(cp);

}
```

Output

```
getcontext has been called
setcontext has been called
```

Related information

- "ucontext.h" on page 82
- "makecontext() — Modify user context" on page 1028
- "setcontext() — Restore user context" on page 1524
- "setjmp() — Preserve stack environment" on page 1546
- "_setjmp() — Set jump point for a nonlocal goto" on page 1549
- "sigaction() — Examine or change a signal action" on page 1611
- "sigsetjmp() — Save stack environment and signal mask" on page 1659
- "swapcontext() — Save and restore user context" on page 1792

__get_cpuid() — Retrieves the system CPUID

Standards

Standards / Extensions	C or C++	Dependencies
z/OS UNIX	both	OS/390 V2R10

Format

```
#define _OPEN_SYS_EXT 1
#include <sys/ps.h>

int __get_cpuid(char *buff);
```

General description

Retrieves the current CPU ID in the form of a string containing the readable part of the serial number concatenated with the model number. The variable *buff* is a character string of 11 bytes in length. It is a work area to build the unique cpuid.

Returned value

Always returns the serial and model number.

Related information

- "sys/ps.h" on page 74

getcwd() — Get path name of the working directory

Standards

Standards / Extensions	C or C++	Dependencies
POSIX.1 XPG4 XPG4.2 Single UNIX Specification, Version 3	both	

Format

```
#define _POSIX_SOURCE
#include <unistd.h>

char *getcwd(char *buffer, size_t size);
```

General description

Determines the path name of the working directory and stores it in *buffer*.

size The number of characters in the *buffer* area.

buffer The name of the buffer that will be used to hold the path name of the working directory. *buffer* must be big enough to hold the working directory name, plus a terminating NULL to mark the end of the name.

Returned value

If successful, getcwd() returns a pointer to the buffer.

If unsuccessful, getcwd() returns a NULL pointer and sets errno to one of the following values:

Error Code
: **Description**

EACCES
The process did not have read or search permission on some component of the working directory's path name.

EINVAL
size is less than or equal to zero.

EIO An input/output error occurred.

ENOENT
A component of the working directory's path name does not exist.

ENOTDIR
A directory component of the working directory's path name is not really a directory.

ERANGE
size is greater than 0, but less than the length of the working directory's path name, plus 1 for the terminating NULL.

Example

CELEBG03

```
/* CELEBG03

   This example determines the working directory.

 */
#define _POSIX_SOURCE
#include <unistd.h>
#undef _POSIX_SOURCE
#include <stdio.h>

main() {
  char cwd[256];

  if (chdir("/tmp") != 0)
    perror("chdir() error()");
  else {
    if (getcwd(cwd, sizeof(cwd)) == NULL)
      perror("getcwd() error");
    else
      printf("current working directory is: %s\n", cwd);
  }
}
```

Output

```
current working directory is: /tmp
```

Related information

- “unistd.h” on page 82
- “chdir() — Change the working directory” on page 267

getdate(), getdate64() — Convert user format date and time

Standards

Standards / Extensions	C or C++	Dependencies
XPG4.2 Single UNIX Specification, Version 3 Language Environment	both	

Format

```
#define _XOPEN_SOURCE_EXTENDED 1
#include <time.h>

struct tm *getdate(const char *string);

extern int getdate_err;
#define _LARGE_TIME_API
#include <time.h>

struct tm *getdate64(const char *string);
```

General description

The getdate() function converts definable date and/or time specifications pointed to by *string* into a tm structure. The tm structure declaration is in the header <time.h>.

Templates are used to parse and interpret the input string. The templates are contained in text files created by the process and identified using the environment variable DATEMSK. The DATEMSK variable should be set to indicate the full path name of the file that contains the templates. The first line in the template that matches the input specification is used for the interpretation and conversion into the internal time format.

The following field descriptors are supported:

%% same as %.

%a abbreviated weekday name

%A full weekday name

%b abbreviated month name

%B full month name

%c locale's appropriate date and time representation

%C Century number [00,99]; leading zeros are permitted but not required. Used in conjuction with %y

%d day of month (01-31; the leading 0 is optional)

%D date as %m/%d/%y

%e same as %d

%h same as %b

%H hour (00-23; the leading 0 is optional)

%I hour (01-12; the leading 0 is optional)

%m month number (00-11; the leading 0 is optional)

%M minute (00-59; the leading 0 is optional)

%n same as `\n`

%p locale's equivalent of either AM or PM

%r locale's 12 hour time representation. In the POSIX locale this is equivalent to %I:%M:%S %p

%R time as %H:%M

%S Seconds [00,60]. The range goes to 60 (rather than stopping at 59) to allow positive leap seconds to be expressed. Since leap seconds cannot be predicted by any algorithm, leap second data must come from some external source.

%t same as `\t` (tab)

%T time as %H:%M:%S

%w weekday number (0-6; 0 indicates Sunday)

%x locale's date representation. In the POSIX locale this is equivalent to %m/%d/%y.

%X locale's time representation. In the POSIX locale this is equivalent to %H:%M:%S.

%y year within century. When a century is not otherwise specified, values in the range 69-99 refer to years in the twentieth century (1969 to 1999 inclusive); values in the range 00-68 refer to years in the twenty-first century (2000 to 2068 inclusive). 31-bit supports values in the range 00-37 and 64-bit supports values in the range 00-68.

%Y

year as ccyy (1969-9999)

31-bit the upper bound for %Y is 2037.

64-bit the upper bound for %Y is 9999.

%Z time zone name or no characters if no time zone exists. If the time zone supplied for %Z is not the time zone getdate() expects, a non-valid input specification error will result. The getdate() function calculates an expected time zone based on time and date information supplied to it.

The match between the template and input specification performed by getdate() is case insensitive.

The month and weekday names can consist of any combination of uppercase or lowercase letters. The process can request that the input date and time specification be in a specific language by setting the `LC_TIME` category (see setlocale()).

Leading 0's are not necessary for the descriptors that allow leading 0's. However, at most two digits are allowed for those descriptors, including leading 0's. Extra white space in either the template file or in *string* is ignored.

The field descriptors %c, %x, and %X will not be supported if they include unsupported field descriptors.

The following rules apply for converting the input specification into a tm structure:

- If only weekday is given, today is assumed if the given day is equal to the current day and next week if it is less,
- If only the month is given, the current month is assumed if the given month is equal to the current month and next year if it is less and no year is given (the first day of the month is assumed if no day is given),
- If no hour, minute, and second are given, the current hour, minute and second are assumed,
- If no date is given, today is assumed if the given hour is greater than the current hour and tomorrow is assumed if it is less.

Notes:

1. When converting an input specification into a tm structure, the getdate() function assumes current time and date values for missing fields as indicated in the previous list. The function treats these values as local time, based on the customization of time zone data.
2. When neither the TZ (POSIX) nor _TZ (non-POSIX) environment variable is defined, the current locale is interrogated for time zone information. If neither TZ nor _TZ is defined and LC_TOD time zone information is not present in the current locale, a default value is applied to local time. POSIX programs simply default to Coordinated Universal Time (UTC), while non-POSIX programs establish an offset from UTC based on the setting of the system clock. For more information about customizing a time zone to work with local time, see "Customizing a time zone" in *z/OS XL C/C++ Programming Guide*.

If a field descriptor in the template file is not one of the supported field descriptors, then the following behaviors exist when the descriptor is encountered:

1. In an ASCII application, it is ignored (treated as non-matching) and processing continues to the next template.
2. In an EBCDIC application, it causes the function to return unsuccessful with getdate_err being set to the value 8.

The function getdate64() will behave exactly like getdate() except it will convert definable date or time specifications pointed to by string into a *tm* structure of calendar times beyond 03:14:07 UTC on January 19, 2038 with a limit of 23:59:59 UTC on December 31, 9999.

Returned value

If successful, getdate() returns a pointer to a tm structure.

If unsuccessful, getdate() returns a NULL pointer and sets the external variable `getdate_err` to a value indicating the error.

The tm structure to which getdate() returns a pointer is not shared with any other functions. Also, the getdate() function produces a tm structure unique to the thread on which it runs.

As is true for all external variables, C/370 allocates storage for the `getdate_err` external variable in writable static storage which is shared among all threads. Thus, `getdate_err` is not intrinsically "thread-safe".

C/370 allocates storage on a per thread basis for an analog of `getdate_err`. The __gderr() function returns a pointer to this storage. It is recommended that multithread applications and applications running from a DLL use the __gderr()

function rather than getdate_err if getdate() returns a NULL pointer to determine in a thread-safe manner why getdate() was unsuccessful.

The __gderr() is defined as follows:

```
#include <time.h>

int *__gderr(void);
```

The __gderr() function returns a pointer to the thread-specific value of getdate_err. The getdate64() function affects the same pointer to the thread-specific value of getdate_err as the getdate() function does.

The following is a list of getdate_err settings and their description:

1 The DATEMSK environment variable is NULL or undefined.

2 The template file cannot be opened for reading.

3 Failed to get file status information.

4 The template file is not a regular file.

5 An error was encountered while reading the template file.

6 Memory allocation failed (not enough memory available).

7 No line in the template file matches the input specification.

8 Non-valid input specification. For example, February 31; or a time that can not be represented in a time_t (representing the time is seconds since Epoch - midnight, January 1, 1970 (UTC)).

9 Unable to determine current time.

Note: This value is unique for z/OS UNIX services.

Related information

- "time.h" on page 79

getdtablesize() — Get the file descriptor table size

Standards

Standards / Extensions	C or C++	Dependencies
XPG4.2	both	

Format

```
#define _XOPEN_SOURCE_EXTENDED 1
#include <unistd.h>

int getdtablesize(void);
```

General description

The getdtablesize() function is equivalent to getrlimit() with the RLIMIT_NOFILE option.

Note:

This function is kept for historical reasons. It was part of the Legacy Feature in Single UNIX Specification, Version 2, but has been withdrawn and is not supported as part of Single UNIX Specification, Version 3. New applications should use getrlimit() instead of getdtablesize().

If it is necessary to continue using this function in an application written for Single UNIX Specification, Version 3, define the feature test macro _UNIX03_WITHDRAWN before including any standard system headers. The macro exposes all interfaces and symbols removed in Single UNIX Specification, Version 3.

Returned value

getdtablesize() returns the current soft limit as if obtained from a call to getrlimit().

There are no errno values defined.

Related information

- "unistd.h" on page 82
- "close() — Close a file" on page 288
- "getrlimit() — Get current or maximum resource consumption" on page 765
- "open() — Open a file" on page 1152
- "select(), pselect() — Monitor activity on files or sockets and message queues" on page 1470
- "setrlimit() — Control maximum resource consumption" on page 1573

getegid() — Get the effective group ID

Standards

Standards / Extensions	C or C++	Dependencies
POSIX.1 XPG4 XPG4.2 Single UNIX Specification, Version 3	both	

Format

```
#define _POSIX_SOURCE
#include <unistd.h>

gid_t getegid(void);
```

General description

Finds the effective group ID (GID) of the calling process.

Returned value

Returns the effective group ID (GID). It is always successful.

There are no documented errno values.

Example

CELEBG04

```
/* CELEBG04

   This example finds the group ID.

 */
#define _POSIX_SOURCE
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>

main() {
  printf("my group id is %d\n", (int) getgid());
}
```

Output

```
my group id is 500
```

Related information

- "sys/types.h" on page 75
- "unistd.h" on page 82
- "getgid() — Get the real group ID" on page 708
- "setegid() — Set the effective group ID" on page 1527
- "setgid() — Set the group ID" on page 1537

getenv() — Get value of environment variables

Standards

Standards / Extensions	C or C++	Dependencies
ISO C POSIX.1 XPG4 XPG4.2 C99 Single UNIX Specification, Version 3	both	

Format

```
#include <stdlib.h>

char *getenv(const char *varname);
```

General description

Searches the table of environment variables for an entry corresponding to *varname* and returns a pointer to a buffer containing the current string value of *varname*.

Special behavior for POSIX: Under POSIX, the value of the `char **environ` pointer is honored and used by getenv(). You can declare and use this pointer. Under POSIX(OFF) this is not the case: the table start cannot be modified.

Returned value

If successful, getenv() returns a pointer to a buffer containing the current string value of *varname*. You should copy the string that is returned because a subsequent call to getenv() will overwrite it.

If the *varname* is not found, getenv() returns a NULL pointer. The returned value is NULL if the given variable is not currently defined.

Example

CELEBG05

```
/* CELEBG05

   In this example, *pathvar points to the value of the PATH
   environment variable.
   In a POSIX environment, this variable would be from the CENV
   group ID.

 */
#include <stdlib.h>
#include <stdio.h>

int main(void)
{
   char *pathvar;

   pathvar = getenv("PATH");
   printf("pathvar=%s",pathvar);
}
```

Related information

- "Using Environment Variables" in *z/OS XL C/C++ Programming Guide*
- "stdlib.h" on page 70
- "clearenv() — Clear environment variables" on page 281
- "__getenv() — Get an environment variable"
- "setenv() — Add, delete, and change environment variables" on page 1528
- "putenv() — Change or add an environment variable" on page 1348

__getenv() — Get an environment variable

Standards

Standards / Extensions	C or C++	Dependencies
z/OS UNIX	both	

Format

```
#include <stdlib.h>

char  *__getenv(const char *varname);
```

General description

__getenv() returns a unique character pointer for each environmental variable. For single-threaded applications, this eliminates the need to copy the string returned by previous __getenv() calls.

This function should not be used by multithreaded applications. Updates to the environmental variable on another thread may invalidate the address returned by __getenv() before the application copies the returned value.

The format of an environment variable is made up of three parts that are combined to form:

name=value

Where:

1. The first part, *name*, is a character string that represents the name of the environment variable. It is this part of the environment variable that __getenv() tries to match with *varname*.
2. The second part, =, is a separator character (since the equal sign is used as a separator character it cannot appear in the *name*).
3. The third part, *value*, is a NULL-terminated character string that represents the value that the environment variable, *name*, is set to. This is the part of the environment variable that __getenv() returns a pointer to.

There are several ways to establish a set of environment variables.

- Set at program initialization time from the Language Environment runtime option ENVAR.
- Set at program initialization time from a data set.
- If the program was invoked with a system() call, they can be inherited from the calling enclave.
- In the z/OS UNIX environment they can also be inherited from the parent process if the program was invoked with one of the exec functions.
- During the running of a program they can be set with the setenv() function or the putenv() function.

For a list of the environment variables that z/OS UNIX services support, see the chapter "Using Environment Variables" in *z/OS XL C/C++ Programming Guide*.

Special behavior for POSIX: Under POSIX, the value of the `char **environ` pointer is honored and used by getenv(). You can declare and use this pointer. Under POSIX(OFF) this is not the case: the table start cannot be modified.

Note: The __getenv() function has a dependency on the level of the Enhanced ASCII Extensions. See "Enhanced ASCII support" on page 2113 for details.

Returned value

If successful, __getenv() returns a pointer to the string containing the value of the environment variable specified by *varname*.

If unsuccessful, __getenv() returns a NULL pointer. The returned value is NULL if the given variable is not currently defined or if the system does not support environment variables.

Related information

- "Using Environment Variables" in *z/OS XL C/C++ Programming Guide*
- "stdlib.h" on page 70
- "clearenv() — Clear environment variables" on page 281
- "getenv() — Get value of environment variables" on page 704
- "putenv() — Change or add an environment variable" on page 1348
- "setenv() — Add, delete, and change environment variables" on page 1528

geteuid() — Get the effective user ID

Standards

Standards / Extensions	C or C++	Dependencies
POSIX.1 XPG4 XPG4.2 Single UNIX Specification, Version 3	both	

Format

```
#define _POSIX_SOURCE
#include <unistd.h>

uid_t geteuid(void);
```

General description

Finds the effective user ID (UID) of the calling process.

Returned value

Returns the effective user ID of the calling process. It is always successful.

There are no documented errno values.

Example

CELEBG06

```
/* CELEBG06

   This example returns information for your user ID.

 */
#define _POSIX_SOURCE
#include <sys/types.h>
#include <pwd.h>
#include <unistd.h>

main() {
  struct passwd *p;
  uid_t  uid;

  if ((p = getpwuid(uid = geteuid())) == NULL)
    perror("getpwuid() error");
  else {
    puts("getpwuid() returned the following info for your userid:");
    printf("  pw_name  : %s\n",       p->pw_name);
    printf("  pw_uid   : %d\n", (int) p->pw_uid);
    printf("  pw_gid   : %d\n", (int) p->pw_gid);
    printf("  pw_dir   : %s\n",       p->pw_dir);
    printf("  pw_shell : %s\n",       p->pw_shell);
  }
}
```

Output

getpwuid() returns the following information for your user ID:

```
pw_name  : MVSUSR1
pw_uid   : 25
pw_gid   : 500
pw_dir   : /u/mvsusr1
pw_shell : /bin/sh
```

Related information

- "sys/types.h" on page 75
- "unistd.h" on page 82
- "getuid() — Get the real user ID" on page 791
- "seteuid() — Set the effective user ID" on page 1531
- "setreuid() — Set real and effective user IDs" on page 1571
- "setuid() — Set the effective user ID" on page 1592

getgid() — Get the real group ID

Standards

Standards / Extensions	C or C++	Dependencies
POSIX.1 XPG4 XPG4.2 Single UNIX Specification, Version 3	both	

Format

```
#define _POSIX_SOURCE
#include <unistd.h>

gid_t getgid(void);
```

General description

Finds the real group ID (GID) of the calling process.

Returned value

Returns the real group ID of the calling process. It is always successful.

There are no documented errno values.

Example

CELEBG07

```
/* CELEBG07

   This example gets the real group ID.

 */
#define _POSIX_SOURCE
#include <sys/types.h>
#include <stdio.h>
#include <unistd.h>

main() {
  printf("my group id is %d\n", (int) getgid());
}
```

Output

```
my group id is 500
```

Related information

- “sys/types.h” on page 75
- “unistd.h” on page 82
- “getegid() — Get the effective group ID” on page 703
- “geteuid() — Get the effective user ID” on page 707
- “getuid() — Get the real user ID” on page 791
- “setgid() — Set the group ID” on page 1537

getgrent() — Get group database entry

The information for this function is included in “endgrent() — Group database entry functions” on page 420.

getgrgid() — Access the group database by ID

Standards

Standards / Extensions	C or C++	Dependencies
POSIX.1 XPG4 XPG4.2 Single UNIX Specification, Version 3	both	

Format

```
#define _POSIX_SOURCE
#include <grp.h>

struct group *getgrgid(gid_t gid);
```

General description

Provides information about the group specified by *gid* and its members.

Returned value

If successful, getgrgid() returns a pointer to a group structure containing an entry from the group database with the specified *gid*. The return value may point to static data that is overwritten by each call. This group structure, defined in the grp.h header file, contains the following members:

gr_name
The name of the group

gr_gid The numerical group ID (GID)

gr_mem A NULL-terminated vector of pointers to the individual member names

If unsuccessful, getgrgid() returns a NULL pointer.

There are no documented errno values.

Example

CELEBG08

```
/* CELEBG08

   This example provides the root GID and group name.
```

```
 */
#define _POSIX_SOURCE
#include <sys/types.h>
#include <grp.h>
#include <stdio.h>
#include <sys/stat.h>   /*FIX: used to be <stat.h>*/

main() {
  struct stat info;
  struct group *grp;

  if (stat("/", &info) < 0)
    perror("stat() error");
  else {
    printf("The root is owned by gid %d\n", info.st_gid);
    if ((grp = getgrgid(info.st_gid)) == NULL)
      perror("getgrgid() error");
    else
      printf("This group name is %s\n", grp->gr_name);
  }
}
```

Output

```
The root is owned by gid 500
This group name is SYS1
```

Related information

- "grp.h" on page 32
- "sys/types.h" on page 75
- "endgrent() — Group database entry functions" on page 420
- "getgrgid_r() — Get group database entry for a group ID"
- "getgrnam() — Access the group database by name" on page 711
- "getgrnam_r() — Search group database for a name" on page 713
- "getlogin() — Get the user login name" on page 731
- "getlogin_r() — Get login name" on page 732

getgrgid_r() — Get group database entry for a group ID

Standards

Standards / Extensions	C or C++	Dependencies
Single UNIX Specification, Version 2 Single UNIX Specification, Version 3 z/OS UNIX	both	OS/390 V2R8

Format

```
#define _XOPEN_SOURCE 500
#include <sys/types.h>
#include <grp.h>

int getgrgid_r(gid_t gid, struct group *grp, char *buffer,
size_t bufsize, struct group **result);

#define _POSIX_SOURCE
#include <grp.h>

int __getgrgid1(gid_t gid, struct group *grp, char *buffer,
size_t bufsize, struct group **result);
```

General description

The getgrgid_r() function updates the group structure pointed to by *grp* and stores a pointer to that structure at the location pointed to by *result*. The structure contains an entry from the group database with a matching *gid*. Storage referenced by the group structure is allocated from the memory provided with the *buffer* parameter, which is *bufsize* bytes in size. A NULL pointer is returned at the location pointed to by *result* on error or if the requested entry is not found.

The __getgrgid1() function is identical to getgrgid_r() except that it does not return the individual group member names. Element gr_mem of the group structure pointed to by *grp* will be set to NULL.

Returned value

If successful, getgrgid_r() or __getgrgid1() returns 0.

If unsuccessful, getgrgid_r() or __getgrgid1() returns an error number and sets errno to one of the following values:

Error Code
Description

EINVAL
One of the input arguments was not valid. Arguments *grp*, *buffer*, and *result* must not be NULL. Argument *bufsize* must not be 0.

ERANGE
Insufficient storage was supplied in *buffer* and *bufsize* to contain the data to be referenced by the resulting group structure.

EMVSSAFEXTRERR
The system authorization facility (SAF) RACROUTE EXTRACT service had an error.

EMVSSAF2ERR
The SAF Get GMAP service had an error.

Related information

- “grp.h” on page 32
- “sys/types.h” on page 75
- “endgrent() — Group database entry functions” on page 420
- “getgrgid() — Access the group database by ID” on page 709
- “getgrnam() — Access the group database by name”
- “getgrnam_r() — Search group database for a name” on page 713

getgrnam() — Access the group database by name

Standards

Standards / Extensions	C or C++	Dependencies
POSIX.1 XPG4 XPG4.2 Single UNIX Specification, Version 3	both	

Format

```
#define _POSIX_SOURCE
#include <grp.h>

struct group *getgrnam(const char *name);
```

General description

Accesses the group structure containing an entry from the group database with the specified *name*.

Returned value

If successful, getgrnam() returns a pointer to a group structure. The return value may point to static data that is overwritten by each call.

The group structure, defined in the grp.h header file, contains the following members:

gr_name
The name of the group

gr_gid The numerical group ID (GID)

gr_mem A NULL-terminated vector of pointers to the individual member names.

If unsuccessful or if the requested entry is not found, getgrnam() returns a NULL pointer.

There are no documented errno values.

Example

CELEBG09

```
/* CELEBG09

   This example provides the members of a group.

 */
#define _POSIX_SOURCE
#include <sys/types.h>
#include <grp.h>
#include <stdio.h>

main() {
  struct group *grp;
  char   grpname[]="USERS", **curr;

  if ((grp = getgrnam(grpname)) == NULL)
    perror("getgrnam() error");
  else {
    printf("The following are members of group %s:\n", grpname);
    for (curr=grp->gr_mem; (*curr) != NULL; curr++)
      printf("  %s\n", *curr);
  }
}
```

Output

```
The following are members of group USERS:
  MVSUSR1
  MVSUSR2
  MVSUSR3
  MVSUSR4
```

```
MVSUSR5
MVSUSR6
MVSUSR7
MVSUSR8
MVSUSR9
```

Related information

- "grp.h" on page 32
- "sys/types.h" on page 75
- "endgrent() — Group database entry functions" on page 420
- "getgrgid() — Access the group database by ID" on page 709
- "getgrgid_r() — Get group database entry for a group ID" on page 710
- "getgrnam_r() — Search group database for a name"

getgrnam_r() — Search group database for a name

Standards

Standards / Extensions	C or C++	Dependencies
Single UNIX Specification, Version 2 Single UNIX Specification, Version 3 z/OS UNIX	both	OS/390 V2R8

Format

```
#define _XOPEN_SOURCE 500
#include <sys/types.h>
#include <grp.h>

int getgrnam_r(const char *name, struct group *grp, char *buffer,
size_t bufsize, struct group **result);

#define _POSIX_SOURCE
#include <grp.h>

int __getgrnam1(const char *name, struct group *grp, char *buffer,
size_t bufsize, struct group **result);
```

General description

The getgrnam_r() function updates the group structure pointed to by *grp* and stores a pointer to that structure at the location pointed to by *result*. The structure contains an entry from the group database with a matching name. Storage referenced by the group structure is allocated from the memory provided with the *buffer* parameter, which is *bufsize* bytes in size. A NULL pointer is returned at the location pointed to by *result* on error or if the requested entry is not found.

The __getgrnam1() function is identical to getgrnam_r() except that it does not return the individual group member names. Element gr_mem within the group structure pointed to by *grp* will be set to NULL.

Returned value

If successful, getgrnam_r() or __getgrnam1() returns 0.

If unsuccessful, getgrnam_r() or _getgrnam1() returns an error number and sets errno to one of the following values:

Error Code
Description

EINVAL
One of the input arguments was not valid. Arguments *grp*, *buffer*, and *result* must not be NULL. Argument *bufsize* must not be 0.

ERANGE
Insufficient storage was supplied in *buffer* and *bufsize* to contain the data to be referenced by the resulting group structure.

EMVSSAFEXTRERR
The system authorization facility (SAF) RACROUTE EXTRACT service had an error.

EMVSSAF2ERR
The SAF Get GMAP service had an error.

Related information

- "grp.h" on page 32
- "sys/types.h" on page 75
- "endgrent() — Group database entry functions" on page 420
- "getgrgid() — Access the group database by ID" on page 709
- "getgrgid_r() — Get group database entry for a group ID" on page 710
- "getgrnam() — Access the group database by name" on page 711

getgroups() — Get a list of supplementary group IDs

Standards

Standards / Extensions	C or C++	Dependencies
POSIX.1 XPG4 XPG4.2 Single UNIX Specification, Version 3	both	

Format

```
#define _POSIX_SOURCE
#include <unistd.h>

int getgroups(int size, gid_t list[]);
```

General description

Stores the supplementary group IDs of the calling process in the *list* array. *size* gives the number of `gid_t` elements that can be stored in the *list* array.

Returned value

If successful, getgroups() returns the number of supplementary group IDs that it puts into *list*. This value is always greater than or equal to 1 and less than or equal to the value of **NGROUPS_MAX** (which is defined in the limits.h header file).

If *size* is zero, getgroups() returns the total number of supplementary group IDs for the process. getgroups() does not try to store group IDs in *list*.

If unsuccessful, getgroups() returns -1 and sets errno to one of the following values.

Error Code
Description

EINVAL
size was not equal to 0 and is less than the total number of supplementary group IDs for the process. *list* may or may not contain a subset of the supplementary group IDs for the process.

Example

CELEBG10

```
/* CELEBG10

   This example provides a list of the supplementary group IDs.

 */
#define _POSIX_SOURCE
#include <sys/types.h>
#include <grp.h>
#include <stdio.h>
#include <unistd.h>

#define dim(x) (sizeof(x)/sizeof(x[0]))

main() {
  gid_t gids[500];
  struct group *grp;
  int count, curr;

  if ((count = getgroups(dim(gids), gids)) == -1)
    perror("getgroups() error");
  else {
    puts("The following is the list of my supplementary groups:");
    for (curr=0; curr<count; curr++) {
      if ((grp = getgrgid(gids[curr])) == NULL)
        perror("getgrgid() error");
      else
        printf("  %8s (%d)\n", grp->gr_name, (int) gids[curr]);
    }
  }
}
```

Output

```
The following is the list of my supplementary groups:
      SYS1 (500)
     KINGS (512)
    NOBLES (513)
   KNIGHTS (514)
   WIZARDS (515)
   SCRIBES (516)
   JESTERS (517)
  PEASANTS (518)
```

Related information

- “sys/types.h” on page 75
- “unistd.h” on page 82
- “getegid() — Get the effective group ID” on page 703
- “getgid() — Get the real group ID” on page 708
- “getgrnam() — Access the group database by name” on page 711
- “setgid() — Set the group ID” on page 1537

getgroupsbyname() — Get supplementary group IDs by user name

Standards

Standards / Extensions	C or C++	Dependencies
z/OS UNIX	both	

Format

```
#define _POSIX_SOURCE
#include <unistd.h>

int getgroupsbyname(char username[], int size, gid_t list[]);
```

General description

Stores the supplementary group IDs of the specified *username* in the array, *list*. *size* gives the number of `gid_t` elements that can be stored in the array, *list*.

Returned value

If successful, getgroupsbyname() returns the number of supplementary group IDs that it puts into *list*. This value is always greater than or equal to one, and less than or equal to the value of **NGROUPS_MAX**.

If *size* is zero, getgroupsbyname() returns the total number of supplementary group IDs for the process. getgroupsbyname() does not try to store group IDs in *list*.

If unsuccessful, getgroupsbyname() returns -1 and sets errno to one of the following values.

Error Code
Description

EINVAL
size was less than or equal to the total number of supplementary group IDs for the process. *list* may or may not contain a subset of the supplementary group IDs for the process.

Example

CELEBG11

```
/* CELEBG11

   This example provides a list of the supplementary group IDs for
   MVSUSR1.

 */
#define _POSIX_SOURCE
#include <sys/types.h>
#include <grp.h>
#include <stdio.h>
#include <unistd.h>

#define dim(x) (sizeof(x)/sizeof(x[0]))

main() {
  gid_t gids[500];
  struct group *grp;
  int count, curr;
  char user[]="MVSUSR1";

  if ((count = getgroupsbyname(user, dim(gids), gids)) == -1)
    perror("getgroups() error");
  else {
    printf("The following is the list of %s's supplementary groups:\n",
           user);
    for (curr=0; curr<count; curr++) {
      if ((grp = getgrgid(gids[curr])) == NULL)
        perror("getgrgid() error");
      else
        printf("  %8s (%d)\n", grp->gr_name, (int) gids[curr]);
    }
  }
}
```

Output

```
The following is the list of MVSUSR1's supplementary groups:
      SYS1 (500)
     USERS (523)
```

Related information

- "unistd.h" on page 82
- "getegid() — Get the effective group ID" on page 703
- "getgid() — Get the real group ID" on page 708
- "getgroups() — Get a list of supplementary group IDs" on page 714
- "setgid() — Set the group ID" on page 1537

gethostbyaddr() — Get a host entry by address

Standards

Standards / Extensions	C or C++	Dependencies
XPG4.2 Single UNIX Specification, Version 3	both	

Format

X/Open:

```
#define _XOPEN_SOURCE_EXTENDED 1
#include <netdb.h>
extern int h_errno;

struct hostent *gethostbyaddr(const void *address, size_t len, int type);
```

Berkeley sockets:

```
#define _OE_SOCKETS
#include <netdb.h>

struct hostent *gethostbyaddr(char *address, int address_len, int domain);
```

General description

The gethostbyaddr() call tries to resolve the host address through a name server, if one is present. gethostbyaddr() searches the local host tables until a matching host address is found or an EOF marker is reached.

Parameter
Description

address The pointer to a structure containing the address of the host. (An unsigned long for AF_INET.)

address_len
The size of *address* in bytes.

domain The address domain supported (AF_INET).

If you want gethostbyaddr() to bypass the name server and instead resolve the host address using the local host tables, you must define the RESOLVE_VIA_LOOKUP symbol before including any sockets-related include files in your source program.

You can use the **X_ADDR** environment variable to specify different local host tables and override those supplied by the z/OS global resolver during initialization.

Note: For more information on these local host tables or the environment variables, see *z/OS Communications Server: IP Configuration Guide, SC27-3650*.

The gethostbyaddr() call returns a pointer to a `hostent` structure for the host address specified on the call.

gethostent(), gethostbyaddr(), and gethostbyname() all use the same static area to return the `hostent` structure. This static area is only valid until the next one of these functions is called on the same thread.

The **netdb.h** include file defines the `hostent` structure and contains the following elements:

Element
Description

h_addr_list
A pointer to a NULL-terminated list of host network addresses.

h_addrtype
The type of address returned; currently, it is always set to AF_INET.

h_aliases
A zero-terminated array of alternative names for the host.

h_length
The length of the address in bytes.

h_name
The official name of the host.

The following function (X/Open sockets only) is defined in **netdb.h** and should be used by multithreaded applications when attempting to reference *h_errno* return on error:

```
int *__h_errno(void);
```

Also use this function when you invoke gethostbyaddr() in a DLL.

This function returns a pointer to a thread-specific value for the *h_errno* variable.

Special behavior for C++: To use this function with C++, you must use the _XOPEN_SOURCE_EXTENDED 1 feature test macro.

Note: The gethostbyaddr() and gethostbyname() functions have been moved to obsolescence in Single UNIX Specification, Version 3 and may be withdrawn in a future version. The getaddrinfo() and getnameinfo() functions are preferred for portability.

Returned value

The return value points to static data that is overwritten by subsequent calls. A pointer to a `hostent` structure indicates success. A NULL pointer indicates an error or End Of File (EOF).

If unsuccessful in X/Open, gethostbyaddr() sets *h_errno* to indicate the error as follows:

Error Code
Description

HOST_NOT_FOUND
No such host is known.

NO_DATA
The server recognized the request and the name but no address is available. Another type of request to the name server might return an answer.

NO_RECOVERY
An unexpected server failure occurred from which there is no recovery.

TRY_AGAIN
A temporary error such as no response from a server, indicating the information is not available now but may be at a later time.

Related information

- "netdb.h" on page 49
- "endhostent() — Close the host information data set" on page 421
- "gethostbyname() — Get a host entry by name"
- "gethostent() — Get the next host entry" on page 721
- "sethostent() — Open the host information data set" on page 1539

gethostbyname() — Get a host entry by name

Standards

Standards / Extensions	C or C++	Dependencies
XPG4.2 Single UNIX Specification, Version 3	both	

Format

X/Open:

```
#define _XOPEN_SOURCE_EXTENDED 1
#include <netdb.h>
extern int h_errno;

struct hostent *gethostbyname(const char *name);
```

Berkeley sockets:

```
#define _OE_SOCKETS
#include <netdb.h>

struct hostent *gethostbyname(char *name);
```

General description

The gethostbyname() call tries to resolve the host name through a name server, if one is present. If a name server is not present, gethostbyname() searches the local host tables until a matching host name is found or an EOF marker is reached.

Parameter
Description

name The name of the host.

The gethostbyname() call returns a pointer to a `hostent` structure for the host name specified on the call.

gethostent(), gethostbyaddr(), and gethostbyname() all use the same static area to return the `hostent` structure. This static area is only valid until the next one of these functions is called on the same thread.

If you want gethostbyname() to bypass the name server and instead resolve the host name using the local host tables, you must define the RESOLVE_VIA_LOOKUP symbol before including any sockets-related include files in your source program.

If the name server is not present or the RESOLVE_VIA_LOOKUP option is in effect, you can use the **X_SITE** environment variable to specify different local host tables and override those supplied by the z/OS global resolver during initialization.

Note: For more information on these local host tables or the environment variables, see *z/OS Communications Server: IP Configuration Guide, SC27-3650*.

gethostent(), gethostbyaddr(), and gethostbyname() all use the same static area to return the HOSTENT structure. This static area is only valid until the next one of these functions is called on the same thread.

The **netdb.h** include file defines the `hostent` structure and contains the following elements:

Element
Description

h_addr_list
A pointer to a NULL-terminated list of host network addresses.

h_addrtype
The type of address returned; currently, it is always set to AF_INET.

h_aliases
A zero-terminated array of alternative names for the host.

h_length
The length of the address in bytes.

h_name
The official name of the host.

The following function (X/Open sockets only) is defined in **netdb.h** and should be used by multithreaded applications when attempting to reference *h_errno* return on error:

```
int *__h_errno(void);
```

Also use this function when you invoke gethostbyname() in a DLL. This function returns a pointer to a thread-specific value for the *h_errno* variable.

Special Behavior for C++: To use this function with C++, you must use the _XOPEN_SOURCE_EXTENDED 1 feature test macro.

Note: The gethostbyaddr() and gethostbyname() functions have been moved to obsolescence in Single UNIX Specification, Version 3 and may be withdrawn in a future version. The getaddrinfo() and getnameinfo() functions are preferred for portability.

Returned value

The return value points to static data that is overwritten by subsequent calls. A pointer to a `hostent` structure indicates success. A NULL pointer indicates an error or End Of File (EOF).

If unsuccessful in X/Open, gethostbyname() sets *h_errno* to one of the following values:

Error Code
  Description

HOST_NOT_FOUND
  No such host is known.

NO_DATA
  The server recognized the request and the name but no address is available. Another type of request to the name server might return an answer.

NO_RECOVERY
  An unexpected server failure occurred from which there is no recovery.

TRY_AGAIN
  A temporary error such as no response from a server, indicating the information is not available now but may be at a later time.

Related information

- "netdb.h" on page 49
- "endhostent() — Close the host information data set" on page 421
- "gethostbyaddr() — Get a host entry by address" on page 717
- "gethostent() — Get the next host entry"
- "gethostname() — Get the name of the host processor" on page 724
- "sethostent() — Open the host information data set" on page 1539

gethostent() — Get the next host entry

Standards

Standards / Extensions	C or C++	Dependencies
XPG4.2 Single UNIX Specification, Version 3	both	

Format

X/Open:

```
#define _XOPEN_SOURCE_EXTENDED 1
#include <netdb.h>

struct hostent *gethostent(void);
```

Berkeley sockets:

```
#define _OE_SOCKETS
#include <netdb.h>

struct hostent *gethostent(void);
```

General description

The gethostent() call reads the next line of the local host tables.

The gethostent() call returns a pointer to the next entry in the local host tables. gethostent() uses the local host tables to get aliases.

You can use the **X_SITE** environment to specify different local host tables and override those supplied by the z/OS resolver during initialization.

Note: For more information on these local host tables or the environment variables, see *z/OS Communications Server: IP Configuration Guide, SC27-3650* .

gethostent(), gethostbyaddr(), and gethostbyname() all use the same static area to return the `hostent` structure. This static area is only valid until the next one of these functions is called on the same thread.

The **netdb.h** include file defines the `hostent` structure and contains the following elements:

Element
Description

h_addrtype
The type of address returned; currently, it is always set to AF_INET.

h_addr A pointer to the network address of the host.

h_aliases
A zero-terminated array of alternative names for host.

h_length
The length of the address in bytes.

h_name
The official name of the host.

Special behavior for C++: To use this function with C++, you must use the _XOPEN_SOURCE_EXTENDED 1 feature test macro.

Returned value

If successful, gethostent() returns a pointer to a `hostent` structure. The return value points to data that is overwritten by subsequent calls returning the same data structure.

If unsuccessful, gethostent() returns a NULL pointer, indicating an error or End Of File (EOF).

Related information

- "netdb.h" on page 49
- "gethostbyaddr() — Get a host entry by address" on page 717
- "gethostbyname() — Get a host entry by name" on page 719
- "sethostent() — Open the host information data set" on page 1539

gethostid() — Get the unique identifier of the current host

Standards

Standards / Extensions	C or C++	Dependencies
XPG4.2 Single UNIX Specification, Version 3	both	

Format

X/Open:

```
#define _XOPEN_SOURCE_EXTENDED 1
#include <unistd.h>

long gethostid(void);
```

Berkeley sockets:

```
#define _OE_SOCKETS
#include <unistd.h>

int gethostid();
```

General description

The gethostid() call gets the unique 32-bit identifier for the current host. This value is the default home Internet address.

Special behavior for C++: To use this function with C++, you must use the _XOPEN_SOURCE_EXTENDED 1 feature test macro.

Returned value

If successful, gethostid() returns the 32-bit identifier of the current host, which should be unique across all hosts.

If unsuccessful, gethostid() returns -1 and stores the error value in errno. For return codes, see *z/OS UNIX System Services Messages and Codes*.

Related information

- "unistd.h" on page 82
- "gethostname() — Get the name of the host processor" on page 724

gethostname() — Get the name of the host processor

Standards

Standards / Extensions	C or C++	Dependencies
XPG4.2 Single UNIX Specification, Version 3	both	

Format

X/Open:

```
#define _XOPEN_SOURCE_EXTENDED 1
#include <unistd.h>

int gethostname(char *name, size_t namelen);
```

Berkeley sockets:

```
#define _OE_SOCKETS
#include <unistd.h>

int gethostname(char *name, int namelen);
```

General description

The gethostname() call returns the name of the host processor that the program is running on. Up to *namelen* characters are copied into the name array. The returned name is NULL-terminated unless there is insufficient room in the name array.

Parameter
Description

name The character array to be filled with the host name.

namelen
The length of *name*.

Special behavior for C++: To use this function with C++, you must use the _XOPEN_SOURCE_EXTENDED 1 feature test macro.

Returned value

If successful, gethostname() returns 0.

If unsuccessful, gethostname() returns -1 and sets errno to one of the following values:

Error Code
Description

EFAULT
Using *name* and *namelen* would result in an attempt to copy the address into a portion of the caller's address space to which data cannot be written.

EMVSPARM
Incorrect parameters were passed to the service.

Related information

- “unistd.h” on page 82
- “gethostbyname() — Get a host entry by name” on page 719
- “gethostid() — Get the unique identifier of the current host” on page 723

getibmopt() — Get IBM TCP/IP image

Standards

Standards / Extensions	C or C++	Dependencies
z/OS UNIX	both	

Format

```
#define _OPEN_SYS_SOCK_EXT
#include <sys/socket.h>

int getibmopt(int cmd, struct ibm_gettcpinfo *bfrp);
```

General description

The getibmopt() function call returns -1 with errno EOPNOTSUPP to indicate that this function is not currently supported.

Parameter
Description

cmd The value in domain must be AF_INET.

bfrp The pointer to an ibm_gettcpinfo structure.

Returned value

getibmopt() always returns -1, indicating that this function is not currently supported.

Error Code
Description

EOPNOTSUPP
This function is not supported.

Related information

- “sys/socket.h” on page 74

getibmsockopt() — Get IBM specific options associated with a socket

Standards

Standards / Extensions	C or C++	Dependencies
z/OS UNIX	both	

Format

```
#define _OPEN_SYS_SOCK_EXT
#include <sys/socket.h>

int getibmsockopt(int s, int level, int optname, char *optval, int *optlen);
```

General description

Like getsockopt(), the getibmsockopt() gets the options associated with a socket in the AF_INET or AF_INET6 domain. Only SOL_SOCKET is supported. This call is for options specific to the IBM implementation of sockets. Currently, only the

SOL_SOCKET level and the socket options SO_NONBLOCKLOCAL and SO_IGNOREINCOMINGPUSH are supported.

Parameter
Description

s The socket descriptor.

level The level for which the option is set.

optname
The name of a specified socket option.

optval The pointer to option data.

optlen The pointer to the length of the option data.

The fields b_num_UNITs_sent and b_num_UNITs_received represent cumulative totals for this socket since the time the application was started.

For SO_NONBLOCKLOCAL, *optval* should point to an integer. getibmsockopt() returns 0 in *optval* if the socket is in blocking mode, and returns 1 in *optval* if the socket is in nonblocking mode.

For SO_IGNOREINCOMINGPUSH, *optval* should point to an integer. getibmsockopt() returns 0 in *optval* if the option is not set, and returns 1 in *optval* if the option is set.

Returned value

If successful, getibmsockopt() returns 0.

If unsuccessful, getibmsockopt() returns -1 and sets errno to one of the following values:

Error Code
Description

EBADF
The s parameter is not a valid socket descriptor.

EFAULT
Using optval and optlen parameters would result in an attempt to access storage outside the caller's address space.

ENOPROTOOPT
The optname parameter is unrecognized, or the level parameter is not SOL_SOCKET.

Related information

- “sys/socket.h” on page 74
- “setibmsockopt() — Set IBM specific options associated with a socket” on page 1542

__getipc() — Query interprocess communications

Standards

Standards / Extensions	C or C++	Dependencies
z/OS UNIX	both	

Format

```
#define _XOPEN_SOURCE
#include <sys/__getipc.h>

int __getipc(int token_id, IPCQPROC *bufptr, size_t buflng, int cmd);
```

General description

The __getipc() function provides means for obtaining information about the status of interprocess communications (IPC) resources, message queues, semaphores, shared memory, and map service memory.

The argument *token_id* is a number that identifies the relative position of an IPC member in the system or specifies a message queue ID, semaphore ID, or shared memory ID. Zero represents the first IPC member ID in the system. On the first call to __getipc(), pass the a *token_id* of zero; the function will return the token that identifies the next IPC resource to which the caller has access. Use this token on the next call to __getipc().

The argument *bufptr* is the address where the data is to be stored.

The argument *buflen* is the length of the buffer.

The argument *cmd* specifies one of the following commands:

IPCQALL
Retrieve the next shared memory, semaphore, or message queue

IPCQMSG
Retrieve the next message member

IPCQSEM
Retrieve the next semaphore member

IPCQSHM
Retrieve the next shared memory member

IPCQMAP
Retrieve the next map service memory currently allocated

IPCQOVER
Overview of system variables. Ignores the value of the first argument *token_id*.

Returned value

If successful, __getipc() returns 0.

If unsuccessful, __getipc() returns -1 and sets errno to one of the following values:

Error Code
Description

EACCES
Operation permission (read) is denied to the calling process for the member ID specified by *token_id*.

EFAULT
The argument *bufptr* contains an non-valid address.

EINVAL
The member ID specified in the argument *token_id* is not valid for the command specified, or the argument *cmd* is not a valid command.

Related information

- "sys/__getipc.h" on page 73
- "sys/ipc.h" on page 73
- "msgctl() — Message control operations" on page 1104
- "msgget() — Get message queue" on page 1105
- "msgrcv() — Message receive operation" on page 1108
- "msgsnd() — Message send operations" on page 1113
- "msgxrcv() — Extended message receive operation" on page 1115
- "semctl() — Semaphore control operations" on page 1482
- "semget() — Get a set of semaphores" on page 1485
- "semop() — Semaphore operations" on page 1488
- "shmat() — Shared memory attach operation" on page 1598
- "shmctl() — Shared memory control operations" on page 1600
- "shmdt() — Shared memory detach operation" on page 1601
- "shmget() — Get a shared memory segment" on page 1602

getipv4sourcefilter() — Get source filter

Standards

Standards / Extensions	C or C++	Dependencies
RFC3678	both	z/OS V1.9

Format

```
#define _OPEN_SYS_SOCK_EXT3
#include <netinet/in.h>

int getipv4sourcefilter(int s, struct in_addr interface,struct in_addr group,
                        uint32_t *fmode, uint32_t *numsrc, struct in_addr *slist);
```

General description

This function allows applications to get a previously set multicast filtering state for a tuple consisting of socket, interface, and multicast group values.

A multicast filter is described by a filter mode, which is MCAST_INCLUDE or MCAST_EXCLUDE, and a list of source addresses which are filtered.

This function is IPv4-specific, must be used only on AF_INET sockets with an open socket of type SOCK_DGRAM or SOCK_RAW.

If the function is unable to obtain the required storage, control will not return to the caller. Instead the application will terminate due to an out of memory condition (if the reserve stack is available and the caller is not XPLINK), or it will terminate with an abend indicating that storage could not be obtained.

Argument
Description

s
Identifies the socket.

interface
Holds the local IP address of the interface.

group Holds the IP multicast address of the group.

fmode Points to an integer that will contain the filter mode on a successful return. The value of this field will be either MCAST_INCLUDE or MCAST_EXCLUDE, which are likewise defined in <netinet/in.h>.

numsrc
It is a pointer that on input, points to the number of source addresses that will fit in the slist array. On return, points to the total number of sources associated with the filter.

slist Points to buffer into which an array of IP addresses of included or excluded (depending on the filter mode) sources will be written. If numsrc was 0 on input, a NULL pointer may be supplied.

Returned value

If successful, the function returns 0. Otherwise, it returns -1 and sets errno to one of the following values.

errno Description

EADDRNOTAVAIL
The tuple consisting of socket, interface, and multicast group values does not exist, or the specified interface address is incorrect for this host, or the specified interface address is not multicast capable.

EBADF
s is not a valid socket descriptor.

EINVAL
Interface or group is not a valid IPv4 address, or the socket s has already requested multicast setsockopt options.

EPROTOTYPE
The socket s is not of type SOCK_DGRAM or SOCK_RAW.

Related information

- “netinet/in.h” on page 53
- “setipv4sourcefilter() — Set source filter” on page 1543

getitimer() — Get value of an interval timer

Standards

Standards / Extensions	C or C++	Dependencies
XPG4.2 Single UNIX Specification, Version 3	both	

Format

```
#define _XOPEN_SOURCE_EXTENDED 1
#include <sys/time.h>

int getitimer(int which, struct itimerval *value);
```

General description

getitimer() gets the current value of an (previously set) interval timer. An interval timer is a timer which sends a signal after each repetition (interval) of time.

The *which* argument indicates what kind of time is being controlled. Values for *which* are:

ITIMER_REAL
This timer is marking real (clock) time. A SIGALRM signal is generated after each interval of time.

Note: alarm() also sets the real interval timer.

ITIMER_VIRTUAL
This timer is marking process virtual time. Process virtual time is the amount of time spent while executing in the process, and can be thought of as a CPU timer. A SIGVTALRM signal is generated after each interval of time.

ITIMER_PROF
This timer is marking process virtual time plus time spent while the system is running on behalf of the process. A SIGPROF signal is generated after each interval of time.

Note: In a multithreaded environment, each of the above timers is specific to a thread of execution for both the generation of the time interval and the measurement of time. For example, an ITIMER_VIRTUAL timer will mark execution time for just the thread, not the entire process.

The *value* argument is a pointer to a structure containing:

`it_interval`
timer interval

`it_value`
current timer value (time remaining)

Each of these fields is a timeval structure, and contains:

`tv_sec` seconds since January 1, 1970 (UTC)

`tv_usec`
microseconds

Returned value

If successful, getitimer() returns 0, and *value* points to the itimerval structure.

If unsuccessful, getitimer() returns -1 and sets errno to one of the following values:

Error Code
Description

EINVAL
which is not a valid timer type.

Related information

- "sys/time.h" on page 75
- "alarm() — Set an alarm" on page 156
- "gettimeofday(), gettimeofday64() — Get date and time" on page 790
- "sleep() — Suspend execution of a thread" on page 1678
- "setitimer() — Set value of an interval timer" on page 1544
- "ualarm() — Set the interval timer" on page 1939
- "usleep() — Suspend execution for an interval" on page 1966

getlogin() — Get the user login name

Standards

Standards / Extensions	C or C++	Dependencies
POSIX.1 XPG4 XPG4.2 Single UNIX Specification, Version 3	both	

Format

_POSIX_SOURCE:

```
#define _POSIX_SOURCE
#include <unistd.h>

char *getlogin(void);
```

_XOPEN_SOURCE:

```
#define _XOPEN_SOURCE
#include <unistd.h>

char *getlogin(void);
```

General description

Finds the name that the login process associated with the current terminal. This string is stored in a static data area and, therefore, may be overwritten with every call to getlogin().

Special behavior for _POSIX_SOURCE: If called from a batch program, a TSO command, or a shell command, getlogin() returns the MVS user name associated with the program. With z/OS UNIX services, this name is a TSO/E user ID. When _POSIX_SOURCE is defined and _XOPEN_SOURCE is not defined, then getlogin() is the same as __getlogin1().

Special behavior for XPG4.2: You must have a TTY at file descriptor 0, 1, or 2, and the TTY must be recorded in the /etc/utmpx database. Someone must have logged in using the TTY. Also, the program must be invoked from a shell session, and file descriptors 0, 1, and 2 are not all redirected.

If getlogin() cannot determine the login name, you can call getuid() to get the user ID of the process, and then call getpwuid() to get a login name associated with that user ID. getpwuid() always returns the `passwd struct` for the same user, even if multiple users have the same UID.

Returned value

If successful, getlogin() returns a pointer to a string that has the login name for the current terminal.

Special behavior for _POSIX_SOURCE: If unsuccessful, getlogin() returns the NULL pointer.

There are no documented errno values.

Special behavior for XPG4.2: If unsuccessful, getlogin() returns a NULL pointer and sets errno to one of the following values:

Error Code
  Description

EMFILE
  OPEN_MAX file descriptors are currently open in the calling process.

ENFILE
  The maximum allowable number of files is currently open in the system.

ENXIO
  The calling process has no controlling terminal.

Example

CELEBG12

```
/* CELEBG12

   This example gets the user login name.

 */
#define _POSIX_SOURCE
#include <stdio.h>
#include <unistd.h>

main() {
  char *user;

  if ((user = __getlogin1()) == NULL)
    perror("__getlogin1() error");
  else printf("__getlogin1() returned %s\n", user);
}
```

Output

```
getlogin() returned MEGA
```

Related information

- "unistd.h" on page 82
- "getlogin_r() — Get login name"
- "__getlogin1() — Get the user login name" on page 733
- "getpwuid() — Access the user database by user ID" on page 763
- "getpwuid_r() — Search user database for a user ID" on page 764
- "getuid() — Get the real user ID" on page 791

getlogin_r() — Get login name

Standards

Standards / Extensions	C or C++	Dependencies
Single UNIX Specification, Version 2 Single UNIX Specification, Version 3	both	OS/390 V2R8

Format

```
#define _XOPEN_SOURCE 500
#include <unistd.h>

int getlogin_r(char *name, size_t namesize);
```

General description

The getlogin_r() function puts the name associated by the login activity with the control terminal of the current process in the character array pointed to by *name*. The array is *namesize* characters long and should have space for the name and the terminating NULL character. The maximum size of the login name is **LOGIN_NAME_MAX**.

If getlogin_r() is successful, *name* points to the name the user used at login, even if there are several login names with the same user ID.

Returned value

If successful, getlogin_r() returns 0.

If unsuccessful, getlogin_r() sets errno to one of the following values:

Error Code
Description

ERANGE
The value of *namesize* is smaller than the length of the string to be returned including the terminating NULL character.

Related information

- "unistd.h" on page 82
- "getlogin() — Get the user login name" on page 731
- "__getlogin1() — Get the user login name"
- "getpwuid() — Access the user database by user ID" on page 763
- "getpwuid_r() — Search user database for a user ID" on page 764
- "getuid() — Get the real user ID" on page 791

__getlogin1() — Get the user login name

Standards

Standards / Extensions	C or C++	Dependencies
POSIX.1 XPG4 XPG4.2	both	

Format

_POSIX_SOURCE:

```
#define _POSIX_SOURCE
#include <unistd.h>

char *__getlogin1(void);
```

General description

Finds the name that the login process associated with the current terminal. If called from batch, __getlogin1() finds the name associated with the batch program. With z/OS UNIX services, this name is a TSO/E user ID unless the USERIDALISTABLE is in use. If the USERIDALISTABLE is setup and a z/OS UNIX alias name exists for a given MVS(TSO, batch user, etc.) userid, then that will be returned.

If __getlogin1() cannot determine the login name, you can call getuid() to get the user ID of the process, and then call getpwuid() to get a login name associated with that user ID. getpwuid() always returns the `passwd struct` for the same user, even if multiple users have the same UID.

Returned value

If successful, __getlogin1() returns a pointer to a string that has the login name for the current terminal.

If unsuccessful, __getlogin1() returns the NULL pointer.

There are no documented errno values.

Example

CELEBG12

```
/* CELEBG12

   This example gets the user login name.

 */
#define _POSIX_SOURCE
#include <stdio.h>
#include <unistd.h>

main() {
  char *user;

  if ((user = __getlogin1()) == NULL)
    perror("__getlogin1() error");
  else printf("__getlogin1() returned %s\n", user);
}
```

Output

```
getlogin() returned MEGA
```

Related information

- "unistd.h" on page 82
- "getlogin() — Get the user login name" on page 731
- "getpwuid() — Access the user database by user ID" on page 763
- "getuid() — Get the real user ID" on page 791

getmccoll() — Get next collating element from string

Standards

Standards / Extensions	C or C++	Dependencies
Language Environment	both	

Format

```
#include <collate.h>

collel_t getmccoll(char **src);
```

General description

If the object pointed to by *src* is not a NULL pointer, the getmccoll() library function determines the longest sequence of bytes in the array pointed to by *src* that constitute a valid multicharacter collating element. It then produces the value of type `collel_t` corresponding to that collating element. The object pointed to by *src* is assigned the address just past the last byte of the multicharacter collating element processed.

Returned value

If successful, getmccoll() returns the value of type `collel_t` that represents the collating element found.

If the object pointed to by *src* is a NULL pointer, or if it points to NULL character, getmccoll() returns 0.

Related information

- "collate.h" on page 19
- "cclass() — Return characters in a character class" on page 243
- "collequiv() — Return a list of equivalent collating elements" on page 295
- "collorder() — Return list of collating elements" on page 297
- "collrange() — Calculate the range list of collating elements" on page 298
- "colltostr() — Return a string for a collating element" on page 299
- "getwmccoll() — Get next collating element from wide string" on page 803
- "ismccollel() — Identify a multicharacter collating element" on page 914
- "maxcoll() — Return maximum collating element" on page 1039
- "strtocoll() — Return collating element for string" on page 1763

getmsg(), getpmsg() — Receive next message from a STREAMS file

Standards

Standards / Extensions	C or C++	Dependencies
XPG4.2 Single UNIX Specification, Version 3	both	

Format

```
#define _XOPEN_SOURCE_EXTENDED 1
#include <stropts.h>

int getmsg(int fildes, struct strbuf *ctlptr,
           struct strbuf *dataptr, int *flagsp);
int getpmsg(int fildes, struct strbuf *ctlptr,
            struct strbuf *dataptr, int *bandp, int *flagsp);
```

General description

The getmsg() function retrieves the contents of a message located at the head of the STREAM head read queue associated with a STREAMS file and places the contents into one or more buffers. The message contains either a data part, a control part, or both. The data and control parts of the message are placed into separate buffers, as described below. The semantics of each part is defined by the originator of the message.

The getpmsg() function does the same thing as getmsg(), but provides finer control over the priority of the messages received. Except where noted, all requirements on getmsg() also pertain to getpmsg().

The *fildes* argument specifies a file descriptor referencing a STREAMS-based file.

The *ctlptr* and *dataptr* arguments each point to a `strbuf` structure, in which the **buf** member points to a buffer in which the data or control information is to be placed, and the **maxlen** member indicates the maximum number of bytes this buffer can hold. On return, the **len** member contains the number of bytes of data or control information actually received. The **len** member is set to 0 if there is a zero-length control or data part and **len** is set to -1 if no data or control information is present in the message.

When getmsg() is called, *flagsp* should point to an integer that indicates the type of message the process is able to receive. This is described further below.

The *ctlptr* argument is used to hold the control part of the message, and *dataptr* is used to hold the data part of the message. If *ctlptr* (or *dataptr*) is a NULL pointer or the **maxlen** member is -1, the control (or data) part of the message is not processed and is left on the STREAM head read queue. If the *ctlptr* (or *dataptr*) is not a NULL pointer, **len** is set to -1. If the **maxlen** member is set to 0 and there is a zero-length control (or data) part, that zero-length part is removed from the read queue and **len** is set to 0. If the **maxlen** member is set to 0 and there are more than 0 bytes of control (or data) information, that information is left on the read queue and **len** is set to 0. If the **maxlen** member in *ctlptr* (or *dataptr*) is less than the control (or data) part of the message, **maxlen** bytes are retrieved. In this case, the remainder of the message is left on the STREAM head read queue and a nonzero return value is provided.

By default, getmsg() processes the first available message on the STREAM head read queue. However, a process may choose to retrieve only high-priority messages by setting the integer pointed to by *flagsp* to `RS_HIPRI`. In this case, getmsg() and getpmsg() will only process the next message if it is a high-priority message. When the integer pointed to by *flagsp* is 0, any message will be retrieved. In this case, on return, the integer pointed to by *flagsp* will be set to `RS_HIPRI` if a high-priority message was retrieved, or 0 otherwise.

For getpmsg(), the flags are different. The *flagsp* argument points to a bitmask with the following mutually-exclusive flags defined: `MSG_HIPRI`, `MSG_BAND`, and `MSG_ANY`. Like getmsg(), getpmsg() processes the first available message on the STREAM head read queue. A process may choose to retrieve only high-priority messages by setting the integer pointed to by *flagsp* to `MSG_HIPRI` and the integer pointed to by *bandp* to 0. In this case, getpmsg() will only process the next message if it is a high-priority message. In a similar manner, a process may choose to retrieve a message from a particular priority band by setting the integer pointed to by *flagsp* to `MSG_BAND` and the integer pointed to by *bandp* to the priority band of interest. In this case, getpmsg() will only process the next message if it is in a priority band equal to, or greater than, the integer pointed to by *bandp*, or if it is a high-priority message. If a process just wants to get the first message off the queue, the integer pointed to by *flagsp* should be set to `MSG_ANY` and the integer pointed to by *bandp* should be set to 0. On return, if the message retrieved was a high-priority message, the integer pointed to by *flagsp* will be set to `MSG_HIPRI` and the integer pointed to by *bandp* will be set to 0. Otherwise, the integer pointed to by *flagsp* will be set to `MSG_BAND` and the integer pointed to by *bandp* will be set to the priority band of the message.

If O_NONBLOCK is not set, getmsg() and getpmsg() will block until a message of the type specified by *flagsp* is available at the front of the STREAM head read queue. If O_NONBLOCK is set and a message of the specified type is not present at the front of the read queue, getmsg() and getpmsg() fail and set errno to EAGAIN.

If a hang-up occurs on the STREAM from which messages are to be retrieved, getmsg() and getpmsg() continue to operate normally, as described above, until the STREAM head read queue is empty. Thereafter, they return 0 in the *len* members of *ctlptr* and *dataptr*.

The following symbolic constants are defined under _XOPEN_SOURCE_EXTENDED 1 in <stropts.h>.

MSG_ANY
Receive any message.

MSG_BAND
Receive message from specified band.

MSG_HIPRI
Send/Receive high priority message.

MORECTL
More control information is left in message.

MOREDATA
More data is left in message.

Returned value

If successful, getmsg() and getpmsg() return a nonnegative value. A value of 0 indicates that a full message was read successfully. A return value of MORECTL indicates that more control information is waiting for retrieval. A return value of MOREDATA indicates that more data is waiting for retrieval. A return value of the bitwise logical OR of MORECTL and MOREDATA indicates that both types of information remain. Subsequent getmsg() and getpmsg() calls retrieve the remainder of the message. However, if a message of higher priority has come in on the STREAM head read queue, the next call to getmsg() or getpmsg() retrieves that higher-priority message before retrieving the remainder of the previous message.

If unsuccessful, getmsg() and getpmsg() return -1 and set errno to one of the following values.

Note: z/OS UNIX services do not supply any STREAMS devices or pseudodevices. It is impossible for getmsg() and getpmsg() to get a message from a STREAMS file. It will always return -1 with errno set to indicate the failure. See "open() — Open a file" on page 1152 for more information.

Error Code
Description

EAGAIN
The O_NONBLOCK flag is set and no messages are available.

EBADF
The *fildes* argument is not a valid file descriptor open for reading.

EBADMSG
The queued message to be read is not valid for getmsg() or getpmsg() or a pending file descriptor is at the STREAM head.

EINTR
A signal was caught during getmsg() or getpmsg()

EINVAL
An illegal value was specified by *flagsp*, or the STREAM or multiplexer referenced by *fildes* is linked (directly or indirectly) downstream from a multiplexer.

ENOSTR
A STREAM is not associated with *fildes*.

In addition, getmsg() and getpmsg() will fail if the STREAM head had processed an asynchronous error before the call. In this case, the value of errno does not reflect the result of getmsg() or getpmsg() but reflects the prior error.

Related information

- “stropts.h” on page 72
- “poll() — Monitor activity on file descriptors and message queues” on page 1185
- “putmsg(), putpmsg() — Send a message on a STREAM” on page 1350
- “read() — Read from a file or socket” on page 1376
- “write() — Write data on a file or socket” on page 2085

getnameinfo() — Get name information

Standards

Standards / Extensions	C or C++	Dependencies
RFC2553 Single UNIX Specification, Version 3	both	z/OS V1R4

Format

```
#define _OPEN_SYS_SOCK_IPV6
#include <sys/socket.h>
#include <netdb.h>

int getnameinfo(cons struct sockaddr *sa, socklen_t salen,
                char *host, socklen_t hostlen,
                char *serv, socklen_t servlen,
                int flags);
```

SUSV3

```
#define _POSIX_C_SOURCE 200112L
#include <sys/socket.h>
#include <netdb.h>

int getnameinfo(cons struct sockaddr *__restrict__ sa, socklen_t salen,
        char *__restrict__ host, socklen_t hostlen, char *__restrict__ serv,
          socklen_t servlen, int flags);
```

General description

The getnameinfo() function translates a socket address to a node name and service location. The getnameinfo() function looks up an IP address and port number provided by the caller in the DNS and system-specific database, and returns text strings for both in buffers provided by the caller.

The *sa* argument points to either a sockaddr_in structure (for IPv4) or a sockaddr_in6 structure (for IPv6) that holds the IP address and port number. The sockaddr_in6 structure may also contain a zone index value, if the IPv6 address represented by this sockaddr_in6 structure is a link-local address. The *salen* argument gives the length of the sockaddr_in or sockaddr_in6 structure.

If the socket address structure contains an IPv4-mapped IPv6 address or an IPv4-compatible IPv6 address, the embedded IPv4 address is extracted and the the lookup is performed on the IPv4 address.

Note: The IPv6 unspecified address ("::") and the IPv6 loopback address ("::1") are not IPv4-compatible addresses. If the address is the IPv6 unspecified address, a lookup is not performed, and the EAI_NONAME error code is returned.

The node name associated with the IP address is returned in the buffer pointed to by the host argument. The caller provides the size of this buffer in the *hostlen* argument. The caller specifies not to return the node name by specifying a zero value for *hostlen* or a null host argument. If the node's name cannot be located, the numeric form of the node's address is returned instead of its name. If a zone index value was present in the sockaddr_in6 structure, the numeric form of the zone index, or the interface name associated with the zone index, is appended to the node name returned, using the format node name%scope information.

If the size of the buffer specified in the *hostlen* argument is insufficient to contain the entire node name, or node name and scope information combination, up to *hostlen* characters will be copied into the buffer as a null terminated string.

The service name associated with the port number is returned in the buffer pointed to by the *serv* argument, and the *servlen* argument gives the length of this buffer. The caller specifies not to the service name by specifying a zero value for *servlen* or a null *serv* argument. If the service's name cannot be located, the numeric of the service address (for example, its port number) will be returned instead of its name.

If the size of the buffer specified in the *servlen* argument is insufficient to contain the entire service name, up to *servlen* characters will be copied into the buffer as a null terminated string.

The final argument, *flags*, is a flag that changes the default actions of this function. By default the fully-qualified domain name (FQDN) for the host is returned.

If the flag bit NI_NOFQDN is set, only the node name portion of the FQDN is returned for local hosts.

If the flag bit NI_NUMERICHOST is set, the numeric form of the host's address is returned instead of its name.

If the flag bit NI_NAMEREQD is set, an error is returned if the host's name cannot be located.

If the flag bit NI_NUMERICSERV is set, the numeric form of the service address is returned (for example, its port number) instead of its name.

If the flag bit NI_NUMERICSCOPE is set, the numeric form of the scope identifier is returned (for example, zone index) instead of its name. This flag is ignored if the *sa* argument is not an IPv6 address.

If the flag bit NI_DGRAM is set, this specifies that the service is a datagram service, and causes getservbyport() to be called with a second argument of "udp" instead of its default of "tcp". This flag is required for the few ports (for example, [512,514]) that have different services for UDP and TCP.

Note: The three NI_NUMERICxxx flags are required to support the "-n" flag that many commands provide.

Special behavior for SUSv3: Starting with z/OS V1.9, environment variable _EDC_SUSV3 can be used to control the behavior of getnameinfo() with respect to detecting if the buffer pointed to by the host or serv argument is too small to contain the entire resolved name. The function will fail and return EAI_OVERFLOW. By default, getnameinfo() will truncate the values pointed to by host or serv and return successfully. When _EDC_SUSV3 is set to 1, getnameinfo() will check for insufficient size buffers to contain the resolved name.

Returned value

Upon successful completion, getnameinfo() returns the node and service names, if requested, in the buffers provided. The returned names are always null-terminated strings.

A zero return value for getnameinfo() indicates successful completion; a non-zero return value indicates failure. The possible values for the failures are listed as follows.

Error Code
    Description

EAI_AGAIN
    The specified host address could not be resolved within the configured time interval, or the resolver address space has not been started. The request can be retried later.

EAI_BADFLAGS
    The flags parameter had an incorrect value.

EAI_FAIL
    An unrecoverable error occurred.

EAI_FAMILY
    The address family was not recognized, or the address length was not valid for the specified family.

EAI_MEMORY
    A memory allocation failure occurred.

EAI_NONAME
    The name does not resolve for the supplied parameter. One of the following occurred:

1. NI_NAMEREQD is set, and the host name cannot be located.
2. Both host name and service name were null.
3. The requested address is valid, but it does not have a record at the name server.

EAI_OVERFLOW
    An argument buffer overflowed. The buffer specified for the host name or the service name was not sufficient to contain the entire resolved name, and the caller previously specified _EDC_SUSV3=1, indicating that truncation was not permitted.

EAI_SYSTEM
An unrecoverable error occurred.

For more information on the above error codes, refer to *z/OS Communications Server: IP and SNA Codes*.

Related information

- "gai_strerror() — Address and name information error description" on page 681
- "getaddrinfo() — Get address information" on page 684
- "getservbyname() — Get a server entry by name" on page 770
- "getservbyport() — Get a service entry by port" on page 772
- "inet_ntop() — Convert Internet address format from binary to text" on page 863
- "socket() — Create a socket" on page 1688
- "netdb.h" on page 49
- "sys/socket.h" on page 74

getnetbyaddr() — Get a network entry by address

Standards

Standards / Extensions	C or C++	Dependencies
XPG4.2 Single UNIX Specification, Version 3	both	

Format

XPG4.2:

```
#define _XOPEN_SOURCE_EXTENDED 1
#include <netdb.h>

struct netent *getnetbyaddr(ip_addr_t net, int type);
```

SUSV3:

```
#define _POSIX_C_SOURCE 200112L
#include <netdb.h>
struct netent  *getnetbyaddr(uint32_t net, int type);
```

Berkeley sockets:

```
#define _OE_SOCKETS
#include <sys/types.h>
#include <netdb.h>

struct netent *getnetbyaddr(unsigned long net, int type);
```

General description

The getnetbyaddr() call searches the *tcpip*.HOSTS.ADDRINFO data set for the specified network address.

Parameter	**Description**
net	The network address.
type	The address domain supported (AF_INET).

If the name server is not present or the RESOLVE_VIA_LOOKUP option is in effect, you can use the **X_ADDR** environment variable to specify a data set other than *tcpip*.HOSTS.ADDRINFO.

Note: For more information on these data sets and environment variables, *tcpip*.HOSTS.LOCAL, *tcpip*.HOSTS.ADDRINFO, and *tcpip*.HOSTS.SITEINFO, see *z/OS Communications Server: IP Configuration Guide, SC27-3650*.

getnetbyaddr(), getnetbyname(), and getnetent() all use the same static area to return the `netent` structure. This static area is only valid until the next one of these functions is called on the same thread.

The `netent` structure is defined in the **netdb.h** include file and contains the following elements:

Element
Description

n_addrtype
The type of network address returned. The call always sets this value to AF_INET.

n_aliases
An array, terminated with a NULL pointer, of alternative names for the network.

n_name
The official name of the network.

n_net The network number, returned in host byte order.

Special behavior for C++: To use this function with C++, you must use the _XOPEN_SOURCE_EXTENDED 1 feature test macro.

Returned value

If successful, getnetbyaddr() returns a pointer to a `netent` structure. The return value points to data that is overwritten by subsequent calls returning the same data structure.

If unsuccessful, getnetbyaddr() returns a NULL pointer, indicating an error or End Of File (EOF).

Related information

- "netdb.h" on page 49
- "endhostent() — Close the host information data set" on page 421
- "endnetent() — Close network information data sets" on page 422
- "getnetbyname() — Get a network entry by name"
- "getnetent() — Get the next network entry" on page 744
- "setnetent() — Open the network information data set" on page 1562

getnetbyname() — Get a network entry by name

Standards

Standards / Extensions	C or C++	Dependencies
XPG4.2 Single UNIX Specification, Version 3	both	

Format

X/Open:

```
#define _XOPEN_SOURCE_EXTENDED 1
#include <netdb.h>

struct netent *getnetbyname(const char *name);
```

Berkeley sockets:

```
#define _OE_SOCKETS
#include <netdb.h>

struct netent *getnetbyname(name);
```

General description

The getnetbyname() call searches the *tcpip*.HOSTS.SITEINFO data set for the specified network name.

Parameter
Description

name The pointer to a network name.

You can use the **X_SITE** environment variable to specify a data set other than *tcpip*.HOSTS.SITEINFO.

Note: For more information on these data sets and environment variables, *tcpip*.HOSTS.LOCAL, *tcpip*.HOSTS.SITEINFO, and *tcpip*.HOSTS.SITEINFO, see *z/OS Communications Server: IP Configuration Guide, SC27-3650*.

The getnetbyname() call returns a pointer to a `netent` structure for the network name specified on the call. getnetbyaddr(), getnetbyname(), and getnetent() all use the same static area to return the `netent` structure. This static area is only valid until the next one of these functions is called on the same thread.

The `netent` structure is defined in the **netdb.h** include file and contains the following elements:

Element
Description

n_addrtype
The type of network address returned. The call always sets this value to AF_INET.

n_aliases
An array, terminated with a NULL pointer, of alternative names for the network.

n_name
The official name of the network.

n_net The network number, returned in host byte order.

Special behavior for C++: To use this function with C++, you must use the _XOPEN_SOURCE_EXTENDED 1 feature test macro.

Returned value

If successful, getnetbyname() returns a pointer to a `netent` structure. The return value points to static data that is overwritten by subsequent calls.

If unsuccessful, getnetbyname() returns a NULL pointer, indicating an error or End Of File (EOF).

Related information

- "netdb.h" on page 49
- "endhostent() — Close the host information data set" on page 421
- "endnetent() — Close network information data sets" on page 422
- "getnetbyaddr() — Get a network entry by address" on page 741
- "getnetent() — Get the next network entry"
- "setnetent() — Open the network information data set" on page 1562

getnetent() — Get the next network entry

Standards

Standards / Extensions	C or C++	Dependencies
XPG4.2 Single UNIX Specification, Version 3	both	

Format

X/Open:

```
#define _XOPEN_SOURCE_EXTENDED 1
#include <netdb.h>

struct netent *getnetent(void);
```

Berkeley sockets:

```
#define _OE_SOCKETS
#include <netdb.h>

struct netent *getnetent(void);
```

General description

The getnetent() call reads the next entry of the *tcpip*.HOSTS.ADDRINFO data set.

You can use the **X_ADDR** environment variable to specify a data set other than *tcpip*.HOSTS.ADDRINFO.

Note: For more information on these data sets and environment variables, *tcpip*.HOSTS.LOCAL, *tcpip*.HOSTS.ADDRINFO, and *tcpip*.HOSTS.SITEINFO, see *z/OS Communications Server: IP Configuration Guide, SC27-3650*.

The getnetent() call returns a pointer to the next entry in the *tcpip*.HOSTS.SITEINFO data set.

getnetbyaddr(), getnetbyname(), and getnetent() all use the same static area to return the `netent` structure. This static area is only valid until the next one of these functions is called on the same thread.

The netent structure is defined in the **netdb.h** include file and contains the following elements:

Element
Description

n_addrtype
The type of network address returned. The call always sets this value to AF_INET.

n_aliases
An array, terminated with a NULL pointer, of alternative names for the network.

n_name
The official name of the network.

n_net The network number, returned in host byte order.

Special behavior for C++: To use this function with C++, you must use the _XOPEN_SOURCE_EXTENDED 1 feature test macro.

Returned value

If successful, getnetent() returns a pointer to a netent structure. The return value points to data that is overwritten by subsequent calls returning the same data structure.

If unsuccessful, getnetent() returns a NULL pointer, indicating an error or End Of File (EOF).

Related information

- "netdb.h" on page 49
- "endhostent() — Close the host information data set" on page 421
- "endnetent() — Close network information data sets" on page 422
- "gethostbyaddr() — Get a host entry by address" on page 717
- "gethostbyname() — Get a host entry by name" on page 719
- "setnetent() — Open the network information data set" on page 1562

getopt() — Command option parsing

Standards

Standards / Extensions	C or C++	Dependencies
XPG4 XPG4.2 Single UNIX Specification, Version 3	both	

Format

```
#define _XOPEN_SOURCE
#include <stdio.h>
int getopt(int argc, char *const argv[], const char *optsting);

extern char *optarg;
extern int optind, opterr, optopt;
```

SUSV3:

```
#define _XOPEN_SOURCE 600
#include <unistd.h>

int getopt(int argc, char *const argv[], const char *optsting);

extern char *optarg;
extern int optind, opterr, optopt;
```

General description

The getopt() function is a command-line parser that can be used by applications that follow Utility Syntax Guidelines 3, 4, 5, 6, 7, 9 and 10 in *X/Open CAE Specification, System Interface Definitions, Issue 4, Version 2* Section 10.2 , Utility Syntax Guidelines. The getopt() function provides the identical functionality described in the X/Open CAE Specification System Interfaces and Headers, Issue 4, Version 2 for the getopt() function with the following extensions:

- If the external variable *optind* is set to zero, the getopt() function treats this as an indication to restart the scan at the first byte of argv[1].

If getopt() encounters an option character that is not contained in *optstring*, it returns the question-mark (?) character. If it detects a missing option-argument, it returns the colon character (:) if the first character of *optstring* was a colon, or a question-mark character (?) otherwise. In either case, getopt() sets the variable *optopt* to the option character that caused the error. If the application has not set the variable *opterr* to 0 and the first character of *optstring* is not a colon, getopt() also prints a diagnostic message to stderr in the format specified for the getopts utility.

Because the getopt() function returns thread-specific data the getopt() function can be used safely from a multithreaded application.

Returned value

If successful, getopt() returns the value of the next option character from *argv* that matches a character in optstring.

A colon (:) is returned if getopt() detects a missing argument and the first character of optstring was a colon (:).

A question-mark (?) is returned if getopt() encounters an option character not in optstring or detects a missing argument and the first character of optstring was not a colon (:).

Otherwise getopt() returns -1 when all command line arguments have been parsed or an unexpected error is encountered in the command line.

getopt() sets the external variables optind, optarg and optopt as described in the X/Open CAE Specification System Interfaces and Headers, Issue 4, Version 2 for the getopt() function.

The following functions defined in <stdio.h> should be used by multithreaded applications when attempting to reference or change the optind, optopt, optarg and opterr external variables:

```
int *__opindf(void);

int *__opoptf(void);
```

```
char **__opargf(void);

int *__operrf(void);
```

Also use these functions when you invoke getopt() in a DLL. These functions return a pointer to a thread-specific value for each variable.

getopt() does not return any errno values.

If getopt() detects a missing argument or an option character not in optstring it will write an error message to stderr describing the option character in error and the invoking program.

Related information

- “stdio.h” on page 68
- “getsubopt() — Parse suboption arguments” on page 787

getpagesize() — Get the current page size

Standards

Standards / Extensions	C or C++	Dependencies
XPG4.2	both	

Format

```
#define _XOPEN_SOURCE_EXTENDED 1
#include <unistd.h>

int getpagesize(void);
```

General description

The getpagesize() function returns the current page size. The getpagesize() function is equivalent to sysconf(_SC_PAGE_SIZE) and sysconf(_SC_PAGESIZE).

Note:

This function is kept for historical reasons. It was part of the Legacy Feature in Single UNIX Specification, Version 2, but has been withdrawn and is not supported as part of Single UNIX Specification, Version 3. New applications should use sysconf(_SC_PAGESIZE) instead of getpagesize().

If it is necessary to continue using this function in an application written for Single UNIX Specification, Version 3, define the feature test macro _UNIX03_WITHDRAWN before including any standard system headers. The macro exposes all interfaces and symbols removed in Single UNIX Specification, Version 3.

Returned value

getpagesize() returns the current page size.

Related information

- “unistd.h” on page 82
- “sysconf() — Determine system configuration options” on page 1798

getpass() — Read a string of characters without echo

Standards

Standards / Extensions	C or C++	Dependencies
XPG4 XPG4.2	both	

Format

```
#define _XOPEN_SOURCE
#include <unistd.h>

char *getpass(const char *prompt);
```

General description

The getpass() function opens the process's controlling terminal, writes to that device the NULL-terminated string *prompt*, disables echoing, reads a string of characters up to the next newline character or EOF, restores the terminal state and closes the terminal.

getpass() only works in an environment where either a controlling terminal exists, or stdin and stderr refer to tty devices. Specifically, it does not work in a TSO environment.

Note:

This function is kept for historical reasons. It was part of the Legacy Feature in Single UNIX Specification, Version 2, but has been withdrawn and is not supported as part of Single UNIX Specification, Version 3.

If it is necessary to continue using this function in an application written for Single UNIX Specification, Version 3, define the feature test macro _UNIX03_WITHDRAWN before including any standard system headers. The macro exposes all interfaces and symbols removed in Single UNIX Specification, Version 3.

Returned value

If successful, getpass() returns a pointer to a NULL-terminated string of at most **PASS_MAX** bytes that were read from the terminal device.

If unsuccessful, getpass() returns a NULL pointer and the terminal state is restored.

Related information

- "unistd.h" on page 82

getpeername() — Get the name of the peer connected to a socket

Standards

Standards / Extensions	C or C++	Dependencies
XPG4.2 Single UNIX Specification, Version 3	both	

Format

X/Open:

```
#define _XOPEN_SOURCE_EXTENDED 1
#include <sys/socket.h>

int getpeername(int socket, struct sockaddr *__restrict__ name,
                socklen_t *__restrict__ namelen);
```

Berkeley sockets:

```
#define _OE_SOCKETS
#include <sys/types.h>
#include <sys/socket.h>

int getpeername(int socket, struct sockaddr *name,
                int *namelen);
```

General description

The getpeername() call returns the name of the peer connected to socket descriptor *socket*. *namelen* must be initialized to indicate the size of the space pointed to by *name* and is set to the number of bytes copied into the space before the call returns. The size of the peer name is returned in bytes. If the actual length of the address is greater than the length of the supplied *sockaddr*, the stored address is truncated. The *sa_len* member of the store structure contains the length of the untruncated address.

Parameter
Description

socket The socket descriptor.

name The Internet address of the connected socket that is filled by getpeername() before it returns. The exact format of *name* is determined by the domain in which communication occurs.

namelen
Must initially point to an integer that contains the size in bytes of the storage pointed to by *name*. On return, that integer contains the size required to represent the address of the connecting socket. If this value is larger than the size supplied on input, then the information contained in sockaddr is truncated to the length supplied on input. If *name* is NULL, *namelen* is ignored.

Sockets in the AF_INET6 domain: For an AF_INET6 socket, the address is returned in a sockaddr_in6 address structure. The sockaddr_in6 structure is defined in the header file **netinet/in.h**.

Special behavior for C++: To use this function with C++, you must use the _XOPEN_SOURCE_EXTENDED 1 feature test macro.

Note: The getpeername() function has a dependency on the level of the Enhanced ASCII Extensions. See "Enhanced ASCII support" on page 2113 for details.

Returned value

If successful, getpeername() returns 0.

If unsuccessful, getpeername() returns -1 and sets errno to one of the following values:

Error Code
Description

EBADF
The *socket* parameter is not a valid socket descriptor.

EFAULT
Using the *name* and *namelen* parameters as specified would result in an attempt to access storage outside of the caller's address space.

EINVAL
The *namelen* parameter is not a valid length. The socket has been shut down.

ENOBUFS
getpeername() is unable to process the request due to insufficient storage.

ENOTCONN
The socket is not in the connected state.

ENOTSOCK
The descriptor is for a file, not for a socket.

EOPTNOTSUPP
The operation is not supported for the socket protocol.

Related information

- “sys/socket.h” on page 74
- “accept() — Accept a new connection on a socket” on page 105
- “connect() — Connect a socket” on page 308
- “getsockname() — Get the name of a socket” on page 775
- “socket() — Create a socket” on page 1688

getpgid() — Get process group ID

Standards

Standards / Extensions	C or C++	Dependencies
XPG4.2 Single UNIX Specification, Version 3	both	

Format

```
#define _XOPEN_SOURCE_EXTENDED 1
#include <unistd.h>

pid_t getpgid(pid_t pid);
```

General description

The getpgid() function returns the process group ID of the process whose process ID is equal to *pid*. If *pid* is 0, getpgid() returns the PID of the calling process.

Returned value

If successful, getpgid() returns a process group ID.

If unsuccessful, getpgid() returns (pid_t)-1 and sets errno to one of the following values:

Error Code
Description

EPERM
The process whose process ID is equal to *pid* is not the same session as the calling process, and the implementation does not allow to the process group ID of that process from the calling process.

ESRCH
There is no process with a process ID equal to *pid*.

getpgid() may fail if:

Error Code
Description

EINVAL
The value of the *pid* argument is not valid.

Related information

- "unistd.h" on page 82
- "exec functions" on page 436
- "fork() — Create a new process" on page 571
- "getpgrp() — Get the process group ID"
- "getsid() — Get process group ID of session leader" on page 774
- "setregid() — Set real and effective group IDs" on page 1570
- "setsid() — Create session, set process group ID" on page 1576

getpgrp() — Get the process group ID

Standards

Standards / Extensions	C or C++	Dependencies
POSIX.1 XPG4 XPG4.2 Single UNIX Specification, Version 3	both	

Format

```
#define _POSIX_SOURCE
#include <unistd.h>

pid_t getpgrp(void);
```

General description

Finds the process group ID of the calling process.

Returned value

Returns the found value. It is always successful.

There are no documented errno values.

Example

CELEBG13

```
/* CELEBG13

   This example gets all the process group IDs.

 */
#define _POSIX_SOURCE
#include <unistd.h>
#include <sys/wait.h>

main() {
  int status;

  if (fork() == 0) {
    if (fork() == 0) {
      printf("grandchild's pid is %d, process group id is %d\n",
             (int) getpid(), (int) getpgrp());
      exit(0);
    }
    printf("child's pid is %d, process group id is %d\n",
           (int) getpid(), (int) getpgrp());
    wait(&status);
    exit(0);
  }
  printf("parent's pid is %d, process group id is %d\n",
         (int) getpid(), (int) getpgrp());
  printf("the parent's parent's pid is %d\n", (int) getppid());
  wait(&status);
}
```

Output

```
parent's pid is 5373959, process group id is 5111816
the parent's parent's pid is 5111816
child's pid is 5832710, process group id is 5111816
grandchild's pid is 196617, process group id is 5111816
```

Related information

- “sys/types.h” on page 75
- “unistd.h” on page 82
- “setpgid() — Set process group ID for job control” on page 1565
- “setsid() — Create session, set process group ID” on page 1576

getpid() — Get the process ID

Standards

Standards / Extensions	C or C++	Dependencies
POSIX.1 XPG4 XPG4.2 Single UNIX Specification, Version 3	both	

Format

```
#define _POSIX_SOURCE
#include <unistd.h>

pid_t getpid(void);
```

General description

Finds the process ID (PID) of the calling process.

Returned value

getpid() returns the found value. It is always successful.

There are no documented errno values.

Example

CELEBG14

```
/* CELEBG14 */
#define _POSIX_SOURCE
#include <stdio.h>
#include <sys/types.h>
#include <signal.h>
#include <unistd.h>

void catcher(int signum) {
  puts("catcher has control!");
}

main() {
  struct sigaction sact;

  sigemptyset(&sact.sa_mask);
  sact.sa_flags = 0;
  sact.sa_handler = catcher;
  sigaction(SIGUSR1, &sact, NULL);

  printf("sending SIGUSR1 to pid %d\n", (int) getpid());
  kill(getpid(), SIGUSR1);
}
```

Output

```
sending SIGUSR1 to pid 5570567
catcher has control!
```

Related information

- “sys/types.h” on page 75
- “unistd.h” on page 82
- “exec functions” on page 436
- “fork() — Create a new process” on page 571
- “getppid() — Get the parent process ID” on page 754
- “kill() — Send a signal to a process” on page 931

getpmsg() — Receive next message from a STREAMS file

The information for this function is included in “getmsg(), getpmsg() — Receive next message from a STREAMS file” on page 735.

getppid() — Get the parent process ID

Standards

Standards / Extensions	C or C++	Dependencies
POSIX.1 XPG4 XPG4.2 Single UNIX Specification, Version 3	both	

Format

```
#define _POSIX_SOURCE
#include <unistd.h>

pid_t getppid(void);
```

General description

Gets the parent process ID (PPID).

Returned value

getppid() returns the parent process ID. It is always successful.

There are no documented errno values.

Example

CELEBG15

```
/* CELEBG15 */
#define _POSIX_SOURCE
#include <stdio.h>
#include <sys/types.h>
#include <signal.h>
#include <unistd.h>
#include <sys/wait.h>                  /*FIX: used to be <wait.h>*/

volatile short footprint=0;

void catcher(int signum) {
  switch (signum) {
    case SIGALRM: puts("caught SIGALRM");
                  break;
    case SIGUSR2: puts("caught SIGUSR2");
                  break;
    default: printf("caught unexpected signal %d\n", signum);
  }
  footprint++;
}

main() {
  struct sigaction sact;
  int status;

  sigemptyset(&sact.sa_mask);
  sact.sa_flags = 0;
  sact.sa_handler = catcher;
  sigaction(SIGUSR2, &sact, NULL);

  sigemptyset(&sact.sa_mask);
  sact.sa_flags = 0;
  sact.sa_handler = catcher;
  sigaction(SIGALRM, &sact, NULL);

  printf("parent (pid %d) is about to fork child\n", (int) getpid());

  if (fork() == 0) {
    printf("child is sending SIGUSR2 to pid %d\n", (int) getppid());
    kill(getppid(), SIGUSR2);
    exit(0);
  }

  alarm(30);
  while (footprint == 0);
  wait(&status);
  puts("parent is exiting");
}
```

Output

```
parent (pid 6094854) is about to fork child
is sending SIGUSR2 to pid 6094854
caught SIGUSR2
parent is exiting
```

Related information

- "sys/types.h" on page 75
- "unistd.h" on page 82
- "exec functions" on page 436
- "fork() — Create a new process" on page 571
- "getpid() — Get the process ID" on page 752
- "kill() — Send a signal to a process" on page 931

getpriority() — Get process scheduling priority

Standards

Standards / Extensions	C or C++	Dependencies
XPG4.2 Single UNIX Specification, Version 3	both	

Format

```
#define _XOPEN_SOURCE_EXTENDED 1
#include <sys/resource.h>

int getpriority(int which, id_t who);
```

General description

getpriority() obtains the current priority of a process, process group or user.

Processes are specified by the values of the *which* and *who* arguments. The *which* argument may be any one of the following set of symbols defined in the *sys/resource.h* include file:

PRIO_PROCESS
indicates that the *who* argument is to be interpreted as a process ID

PRIO_PGRP
indicates that the *who* argument is to be interpreted as a process group ID

PRIO_USER
indicates that the *who* argument is to be interpreted as a user ID

The *who* argument specifies the ID (process, process group, or user). A 0 (zero) value for the *who* argument specifies the current process, process group or user ID.

Returned value

If successful, getpriority() returns the priority of the process, process group, or used ID requested in *who*. The priority is returned as an integer in the range -20 to 19 (the lower the numerical value, the higher the priority).

If more than one process is specified, getpriority() returns the highest priority pertaining to any of the specified processes.

If unsuccessful, getpriority() returns -1 and sets errno to one of the following values:

Error Code
Description

EINVAL
The symbol specified in the *which* argument was not recognized, or the value of the *who* argument is not a valid process ID, process group ID or user ID.

ESRCH
No process could be located using the *which* and *who* argument values specified.

Because getpriority() can return the value -1 on successful completion, it is necessary to set the external variable errno to 0 before a call to getpriority(). If getpriority() returns -1, then errno can be checked to see if an error occurred or if the value is a legitimate priority.

Related information

- "sys/resource.h" on page 74
- "nice() — Change priority of a process" on page 1145
- "setpriority() — Set process scheduling priority" on page 1567

getprotobyname() — Get a protocol entry by name

Standards

Standards / Extensions	C or C++	Dependencies
XPG4.2 Single UNIX Specification, Version 3	both	

Format

X/Open:

```
#define _XOPEN_SOURCE_EXTENDED 1
#include <netdb.h>

struct protoent *getprotobyname(const char *name);
```

Berkeley sockets:

```
#define _OE_SOCKETS
#include <netdb.h>

struct protoent *getprotobyname(char name);
```

General description

The getprotobyname() call searches the */etc/protocol* or *tcpip*.ETC.PROTO data set for the specified protocol name.

Parameter
Description

name The name of the protocol.

The getprotobyname() call returns a pointer to a protoent structure for the network protocol specified on the call. getprotobyname(), getprotobynumber(), and getprotoent() all use the same static area to return the protoent structure. This static area is only valid until the next one of these functions is called on the same thread.

The protoent structure is defined in the **netdb.h** include file and contains the following elements:

Element
Description

p_aliases
An array, terminated with a NULL pointer, of alternative names for the protocol.

p_name
The official name of the protocol.

p_proto
The protocol number.

Special behavior for C++: To use this function with C++, you must use the _XOPEN_SOURCE_EXTENDED 1 feature test macro.

Returned value

If successful, getprotobyname() returns a pointer to a protoent structure. The return value points to data that is overwritten by subsequent calls returning the same data structure.

If unsuccessful, getprotobyname() returns a NULL pointer, indicating an error or End Of File (EOF).

Related information

- “netdb.h” on page 49
- “endprotoent() — Work with a protocol entry” on page 423
- “getprotobynumber() — Get a protocol entry by number” on page 758
- “getprotoent() — Get the next protocol entry” on page 759
- “setprotoent() — Open the protocol information data set” on page 1569

getprotobynumber() — Get a protocol entry by number

Standards

Standards / Extensions	C or C++	Dependencies
XPG4.2 Single UNIX Specification, Version 3	both	

Format

X/Open:

```
#define _XOPEN_SOURCE_EXTENDED 1
#include <netdb.h>

struct protoent *getprotobynumber(int proto);
```

Berkeley sockets:

```
#define _OE_SOCKETS
#include <netdb.h>

struct protoent *getprotobynumber(int proto);
```

General description

The getprotobynumber() call searches the */etc/protocol* or *tcpip*.ETC.PROTO data set for the specified protocol number.

Parameter
Description

proto The protocol number.

The getprotobynumber() call returns a pointer to a `protoent` structure for the network protocol specified on the call. getprotobyname(), getprotobynumber(), and getprotoent() all use the same static area to return the `protoent` structure. This static area is only valid until the next one of these functions is called on the same thread.

The `protoent` structure is defined in the **netdb.h** include file and contains the following elements:

Element
Description

p_aliases
An array, terminated with a NULL pointer, of alternative names for the protocol.

p_name
The official name of the protocol.

p_proto
The protocol number.

Special behavior for C++: To use this function with C++, you must use the _XOPEN_SOURCE_EXTENDED 1 feature test macro.

Returned value

If successful, getprotobynumber() returns a pointer to a `protoent` structure. The return value points to data that is overwritten by subsequent calls returning the same data structure.

If unsuccessful, getprotobynumber() returns a NULL pointer, indicating an error or End Of File (EOF).

Related information

- "netdb.h" on page 49
- "endprotoent() — Work with a protocol entry" on page 423
- "getprotobyname() — Get a protocol entry by name" on page 756
- "getprotoent() — Get the next protocol entry"
- "setprotoent() — Open the protocol information data set" on page 1569

getprotoent() — Get the next protocol entry

Standards

Standards / Extensions	C or C++	Dependencies
XPG4.2 Single UNIX Specification, Version 3	both	

Format

X/Open:

```
#define _XOPEN_SOURCE_EXTENDED 1
#include <netdb.h>

struct protoent *getprotoent(void);
```

Berkeley sockets:

```
#define _OE_SOCKETS
#include <netdb.h>

struct protoent *getprotoent(void);
```

General description

The getprotoent() call reads */etc/protocol* or the *tcpip*.ETC.PROTO data set.

The getprotoent()> call returns a pointer to the next entry in the */etc/protocol* or the *tcpip*.ETC.PROTO data set.

getprotobyname(), getprotobynumber(), and getprotoent() all use the same static area to return the `protoent` structure. This static area is only valid until the next one of these functions is called on the same thread.

The `protoent` structure is defined in the **netdb.h** include file and contains the following elements:

Element
Description

p_aliases
An array, terminated with a NULL pointer, of alternative names for the protocol.

p_name
The official name of the protocol.

p_proto
The protocol number.

Special behavior for C++: To use this function with C++, you must use the _XOPEN_SOURCE_EXTENDED 1 feature test macro.

Returned value

If successful, getprotoent() returns a pointer to a `protoent` structure. The return value points to data that is overwritten by subsequent calls returning the same data structure.

If unsuccessful, getprotoent() returns a NULL pointer, indicating an error or End Of File (EOF).

Related information

- "netdb.h" on page 49
- "endprotoent() — Work with a protocol entry" on page 423
- "getprotobyname() — Get a protocol entry by name" on page 756
- "getprotobynumber() — Get a protocol entry by number" on page 758
- "setprotoent() — Open the protocol information data set" on page 1569

getpwent() — Get user database entry

The information for this function is included in "endpwent() — User database functions" on page 423.

getpwnam() — Access the user database by user name

Standards

Standards / Extensions	C or C++	Dependencies
POSIX.1 XPG4 XPG4.2 Single UNIX Specification, Version 3	both	

Format

```
#define _POSIX_SOURCE
#include <pwd.h>

struct passwd *getpwnam(const char *name);
```

General description

Accesses the `passwd` structure (defined in the pwd.h header file), which contains the following members:

pw_name
User name

pw_uid User ID (UID) number
pw_gid Group ID (GID) number
pw_dir Initial working directory
pw_shell
Initial user program

Returned value

If successful, getpwnam() returns a pointer to a passwd structure containing an entry from the user database with the specified *name*. Return values may point to the static data that is overwritten on each call.

If unsuccessful, getpwnam() returns a NULL pointer and sets errno to one of the following values:

Error Code
Description

EINVAL
A non-valid user *name* is detected.

Example

CELEBG16

```
/* CELEBG16

   This example provides information for the user data
   base, MEGA.

 */
#define _POSIX_SOURCE
#include <sys/types.h>
#include <pwd.h>

main() {
  struct passwd *p;
  char user[]="MEGA";

  if ((p = getpwnam(user)) == NULL)
    perror("getpwnam() error");
  else {
    printf("getpwnam() returned the following info for user %s:\n",
           user);
    printf("  pw_name  : %s\n",       p->pw_name);
    printf("  pw_uid   : %d\n", (int) p->pw_uid);
    printf("  pw_gid   : %d\n", (int) p->pw_gid);
    printf("  pw_dir   : %s\n",       p->pw_dir);
    printf("  pw_shell : %s\n",       p->pw_shell);
  }
}
```

Output

```
  pw_name  : MEGA
  pw_uid   : 0
  pw_gid   : 512
  pw_dir   : /u/mega
  pw_shell : /bin/sh
```

Related information

- “pwd.h” on page 61
- “sys/types.h” on page 75

- “endpwent() — User database functions” on page 423
- “getlogin() — Get the user login name” on page 731
- “getlogin_r() — Get login name” on page 732
- “getpwnam_r() — Search user database for a name”
- “getpwuid() — Access the user database by user ID” on page 763
- “getpwuid_r() — Search user database for a user ID” on page 764

getpwnam_r() — Search user database for a name

Standards

Standards / Extensions	C or C++	Dependencies
Single UNIX Specification, Version 2 Single UNIX Specification, Version 3	both	OS/390 V2R8

Format

```
#define _XOPEN_SOURCE 500
#include <sys/types.h>
#include <pwd.h>

int getpwnam_r(const char *nam, struct passwd *pwd,
char *buffer, size_t bufsize, struct passwd **result);
```

General description

The getpwnam_r() function updates the passwd structure pointed to by *pwd* and stores a pointer to that structure at the location pointed to by *result*. The structure will contain an entry from the user database with a matching name. Storage referenced by the structure is allocated from the memory provided with the *buffer* parameter, which is *bufsize* characters in size. A NULL pointer is returned at the location pointed to by *result* on error or if the requested entry is not found.

Returned value

If successful, getpwnam_r() returns 0.

If unsuccessful, getpwnam_r() sets errno to one of the following values:

Error Code
Description

EINVAL
One of the input arguments was not valid. Arguments *pwd*, *buffer*, and *result* must not be NULL. Argument *bufsize* must not be 0.

ERANGE
Insufficient storage was supplied in *buffer* and *bufsize* to contain the data to be referenced by the resulting passwd structure.

Related information

- “pwd.h” on page 61
- “sys/types.h” on page 75
- “endpwent() — User database functions” on page 423
- “getlogin() — Get the user login name” on page 731
- “getlogin_r() — Get login name” on page 732
- “getpwnam() — Access the user database by user name” on page 760
- “getpwuid() — Access the user database by user ID” on page 763

- "getpwuid_r() — Search user database for a user ID" on page 764

getpwuid() — Access the user database by user ID

Standards

Standards / Extensions	C or C++	Dependencies
POSIX.1 XPG4 XPG4.2 Single UNIX Specification, Version 3	both	

Format

```
#define _POSIX_SOURCE
#include <pwd.h>

struct passwd *getpwuid(uid_t uid);
```

General description

Gets information about a user with the specified *uid*. getpwuid() returns a pointer to a passwd structure containing an entry from the user database for the specified *uid*. This structure (defined in the pwd.h header file), contains the following members:

pw_name
User name

pw_uid User ID (UID) number

pw_gid Group ID (GID) number

pw_dir Initial working directory

pw_shell
Initial user program

Return values may point to the static data that is overwritten on each call.

Returned value

If successful, getpwuid() returns a pointer.

If unsuccessful, getpwuid() returns a NULL pointer and sets errno to one of the following values:

Error Code
Description

EMVSSAF2ERR
The system authorization facility (SAF) or RACF Get GMAP or Get UMAP service had an error.

EMVSSAFEXTRERR
The SAF or RACF RACROUTE EXTRACT service had an error.

Example

CELEBG17

```
/* CELEBG17

   This example provides information for user ID 0.
```

```
 */
#define _POSIX_SOURCE
#include <sys/types.h>
#include <pwd.h>

main() {
  struct passwd *p;
  uid_t  uid=0;

  if ((p = getpwuid(uid)) == NULL)
    perror("getpwuid() error");
  else {
    printf("getpwuid() returned the following info for uid %d:\n",
           (int) uid);
    printf("  pw_name  : %s\n",       p->pw_name);
    printf("  pw_uid   : %d\n", (int) p->pw_uid);
    printf("  pw_gid   : %d\n", (int) p->pw_gid);
    printf("  pw_dir   : %s\n",       p->pw_dir);
    printf("  pw_shell : %s\n",       p->pw_shell);
  }
}
```

Output

```
getpwuid() returned the following info for uid 0:
  pw_name  : MEGA
  pw_uid   : 0
  pw_gid   : 512
  pw_dir   : /u/mega
  pw_shell : /bin/sh
```

Related information

- "pwd.h" on page 61
- "sys/types.h" on page 75
- "endpwent() — User database functions" on page 423
- "getlogin() — Get the user login name" on page 731
- "getlogin_r() — Get login name" on page 732
- "getpwnam() — Access the user database by user name" on page 760
- "getpwnam_r() — Search user database for a name" on page 762
- "getpwuid_r() — Search user database for a user ID"

getpwuid_r() — Search user database for a user ID

Standards

Standards / Extensions	C or C++	Dependencies
Single UNIX Specification, Version 2 Single UNIX Specification, Version 3	both	OS/390 V2R8

Format

```
#define _XOPEN_SOURCE 500
#include <sys/types.h>
#include <pwd.h>

int getpwuid_r(uid_t uid, struct passwd *pwd,
char *buffer, size_t bufsize, struct passwd **result);
```

General description

The getpwuid_r() function updates the passwd structure pointed to by *pwd* and stores a pointer to that structure at the location pointed to by *result*. The structure will contain an entry from the user database with a matching uid. Storage referenced by the structure is allocated from the memory provided with the *buffer* parameter, which is *bufsize* characters in size. A NULL pointer is returned at the location pointed to by *result* on error or if the requested entry is not found.

Returned value

If successful, getpwuid_r() returns 0.

If unsuccessful, getpwuid_r() returns a NULL pointer and sets errno to one of the following values:

Error Code
  Description

EINVAL
  One of the input arguments was not valid. Arguments *pwd*, *buffer*, and *result* must not be NULL. Argument *bufsize* must not be 0.

EMVSSAF2ERR
  The system authorization facility (SAF) or RACF Get GMAP or Get UMAP service had an error.

EMVSSAFEXTRERR
  The SAF or RACF RACROUTE EXTRACT service had an error.

ERANGE
  Insufficient storage was supplied in *buffer* and *bufsize* to contain the data to be referenced by the resulting passwd structure.

Related information

- “pwd.h” on page 61
- “sys/types.h” on page 75
- “endpwent() — User database functions” on page 423
- “getlogin() — Get the user login name” on page 731
- “getlogin_r() — Get login name” on page 732
- “getpwnam() — Access the user database by user name” on page 760
- “getpwnam_r() — Search user database for a name” on page 762
- “getpwuid() — Access the user database by user ID” on page 763

getrlimit() — Get current or maximum resource consumption

Standards

Standards / Extensions	C or C++	Dependencies
XPG4.2 Single UNIX Specification, Version 3	both	

Format

```
#define _XOPEN_SOURCE_EXTENDED 1
#include <sys/resource.h>

int getrlimit(int resource, struct rlimit *rlp);
```

General description

The getrlimit() function gets resource limits for the calling process. A resource limit is a pair of values; one specifying the current (soft) limit, the other a maximum (hard) limit.

The value RLIM_INFINITY defined in sys/resource.h, is considered to be larger than any other limit value. If a call to getrlimit() returns RLIM_INFINITY for a resource, it means the implementation does not enforce limits on that resource.

The *resource* argument specifies which resource to get the hard and/or soft limits for, and may be one of the following values:

RLIMIT_CORE
The maximum size of a dump of memory (in bytes) allowed for the process. A value of 0 (zero) prevents file creation. Dump file creation will stop at this limit.

RLIMIT_CPU
The maximum amount of CPU time (in seconds) allowed for the process. If the limit is exceeded, a SIGXCPU signal is sent to the process and the process is granted a small CPU time extension to allow for signal generation and delivery. If the extension is used up, the process is terminated with a SIGKILL signal.

RLIMIT_DATA
The maximum size of the break value for the process, in bytes. In this implementation, this resource always has a hard and soft limit value of RLIM_INFINITY.

RLIMIT_FSIZE
The maximum file size (in bytes) allowed for the process. A value of 0 (zero) prevents file creation. If the size is exceeded, a SIGXFSZ signal is sent to the process. If the process is blocking, catching, or ignoring SIGXFSZ, continued attempts to increase the size of a file beyond the limit will fail with an errno of EFBIG.

RLIMIT_MEMLIMIT
The maximum amount of usable storage above the 2 gigabyte bar (in 1 megabyte segments) that can be allocated.

RLIMIT_NOFILE
The maximum number of open file descriptors allowed for the process. This number is one greater than the maximum value that may be assigned to a newly created descriptor. (That is, it is one-based.) Any function that attempts to create a new file descriptor beyond the limit will fail with an EMFILE errno.

RLIMIT_STACK
The maximum size of the stack for a process, in bytes. Note that in z/OS UNIX services, the stack is a per-thread resource. In this implementation, this resource always has a hard and soft limit value of RLIM_INFINITY. A call to setrlimit() to set this resource to any value other than RLIM_INFINITY will fail with an errno of EINVAL.

RLIMIT_AS
The maximum address space size for the process, in bytes. If the limit is exceeded, malloc() and mmap() functions will fail with an errno of ENOMEM. Automatic stack growth will also fail.

The *rlp* argument points to a `rlimit` structure. This structure contains the following members:

rlim_cur
The current (soft) limit

rlim_max
The maximum (hard) limit

See the <sys/resource.h> header for more detail.

The resource limit values are propagated across exec and fork.

Special behavior for z/OS UNIX System Services: An exception exists for exec processing in conjunction with daemon support. If a daemon process invokes exec and it had previously invoked setuid() before exec, the `RLIMIT_CPU`, `RLIMIT_AS`, `RLIMIT_CORE`, `RLIMIT_FSIZE`, and `RLIMIT_NOFILE` limit values are set based on the limit values specified in the kernel parmlib member BPXPRMxx.

For processes which are not the only process within an address space, the `RLIMIT_CPU` and `RLIMIT_AS` limits are shared with all the processes within the address space. For `RLIMIT_CPU`, when the soft limit is exceeded, action will be taken on the first process within the address space. If the action is termination, all processes within the address space will be terminated.

In addition to the `RLIMIT_CORE` limit values, the dump file defaults are set by SYSMDUMP defaults. Refer to *z/OS MVS Initialization and Tuning Reference* for information on setting up SYSMDUMP defaults using the IEADMR00 parmlib member.

Dumps of memory are taken in 4160 byte increments. Therefore, RLIMIT_CORE values affect the size of memory dumps in 4160 byte increments. For example, if the RLIMIT_CORE soft limit value is 4000, the dump will contain no data. If the RLIMIT_CORE soft limit value is 8000, the maximum size of a memory dump is 4160 bytes.

Large file support for z/OS UNIX files: Large z/OS UNIX files are supported automatically for AMODE 64 C/C++ applications. AMODE 31 C/C++ applications must be compiled with the option LANGLVL(LONGLONG) and define the _LARGE_FILES feature test macro before any headers are included to enable this function to operate on z/OS UNIX files that are larger than 2 GB in size. File size and offset fields are enlarged to 63 bits in width. Therefore, any other function operating on the file is required to define the _LARGE_FILES feature test macro as well.

Returned value

If successful, getrlimit() returns 0.

If unsuccessful, getrlimit() returns -1 and sets errno to one of the following values:

Error Code
Description

EINVAL
A non-valid *resource* was specified.

Related information

- “sys/resource.h” on page 74
- “stropts.h” on page 72
- “brk() — Change space allocation” on page 217
- “fork() — Create a new process” on page 571
- “getdtablesize() — Get the file descriptor table size” on page 702
- “malloc() — Reserve storage block” on page 1031
- “open() — Open a file” on page 1152
- “rexec() — Execute commands one at a time on a remote host” on page 1447
- “setrlimit() — Control maximum resource consumption” on page 1573
- “sigaltstack() — Set or get signal alternate stack context” on page 1630
- “sysconf() — Determine system configuration options” on page 1798
- “ulimit() — Get or set process file size limits” on page 1942

getrusage() — Get information about resource utilization

Standards

Standards / Extensions	C or C++	Dependencies
XPG4.2 Single UNIX Specification, Version 3	both	

Format

```
#define _XOPEN_SOURCE_EXTENDED 1
#include <sys/resource.h>

int getrusage(int who, struct rusage *r_usage);
```

General description

The getrusage() function provides measures of the resources used by the current process or its terminated and waited-for-child processes. If the value of the *who* argument is RUSAGE_SELF, information is returned about resources used by the current process. If the value of the *who* argument is RUSAGE_CHILDREN, information is returned about resources used by the terminated and waited-for-children of the current process. If the child is never waited for (for instance, if the parent has SA_NOCLDWAIT set or sets SIGCHLD to SIG_IGN), the resource information for the child process is discarded and not included in the resource information provided by getrusage()

The *r_usage* argument is a pointer of an object of type struct rusage in which the returned information is stored.

Returned value

If successful, getrusage() returns 0.

If unsuccessful, getrusage() returns -1 and sets errno to one of the following values:

Error Code
Description

EINVAL
The value of the *who* argument is not valid.

Related information

- "sys/resource.h" on page 74
- "exit() — End program" on page 443
- "sigaction() — Examine or change a signal action" on page 1611
- "time(),time64() — Determine current UTC time" on page 1878
- "times() — Get process and child process times" on page 1879
- "wait() — Wait for a child process to end" on page 1992

gets() — Read a string

Standards

Standards / Extensions	C or C++	Dependencies
ISO C POSIX.1 XPG4 XPG4.2 C99 Single UNIX Specification, Version 3 Language Environment	both	

Format

```
#include <stdio.h>

char *gets(char *buffer);

#define _OPEN_SYS_UNLOCKED_EXT 1
#include <stdio.h>

char *gets_unlocked(char *buffer);
```

General description

Reads bytes from the standard input stream `stdin`, and stores them in the array pointed to by *buffer*. The line consists of all characters up to and including the first newline character (`\n`) or EOF. The gets() function discards any newline character, and the NULL character (\0) is placed immediately after the last byte read. If there is an error, the value stored in *buffer* is undefined.

gets() is not supported for files opened with `type=record` or `type=blocked`.

gets() has the same restriction as any read operation, such as a read immediately following a write, or a write immediately following a read. Between a write and a subsequent read, there must be an intervening flush or reposition. Between a read and a subsequent write, there must also be an intervening flush or reposition unless an EOF has been reached.

gets_unlocked() is functionally equivalent to gets() with the exception that it is not thread-safe. This function can safely be used in a multithreaded application if and only if it is called while the invoking thread owns the (FILE*) object, as is the case after a successful call to either the flockfile() or ftrylockfile() function.

Returned value

If successful, gets() returns its argument.

If unsuccessful, gets() returns a NULL pointer to indicate an error or an EOF condition with no characters read.

Use ferror() or feof() to determine which of these conditions occurred. Note that EOF is only reached when an attempt is made to read past the last byte of data. Reading up to and including the last byte of data does *not* turn on the EOF indicator.

Example

CELEBG18

```
/* CELEBG18

   This example gets a line of input from stdin.

 */
#include <stdio.h>
#define MAX_LINE 100

int main(void)
{
   char line[MAX_LINE];
   char *result;

   printf("Enter string:\n");
   if ((result = gets(line)) != NULL)
      printf("string is %s\n",result);
   else
      if (ferror(stdin))
        printf("Error\n");
}
```

Related information

- “stdio.h” on page 68
- “feof() — Test end of file (EOF) indicator” on page 500
- “ferror() — Test for read and write errors” on page 503
- “fgets() — Read a string from a stream” on page 530
- “fputs() — Write a string” on page 605
- “puts() — Write a string” on page 1353

getservbyname() — Get a server entry by name

Standards

Standards / Extensions	C or C++	Dependencies
XPG4.2 Single UNIX Specification, Version 3	both	

Format

X/Open:

```
#define _XOPEN_SOURCE_EXTENDED 1
#include <netdb.h>

struct servent *getservbyname(const char *name, const char *proto);
```

Berkeley sockets:

```
#define _OE_SOCKETS
#include <netdb.h>

struct servent *getservbyname(char *name, char *proto);
```

General description

The getservbyname() call searches the */etc/services* or *tcpip*.ETC.SERVICES data set for the first entry that matches the specified service name and protocol name. If proto is NULL, only the service name must match.

Parameter
Description

name The service name.

proto The protocol name.

The getservbyname() call returns a pointer to a *servent* structure for the network service specified on the call. getservbyname(), getservbyport(), and getservent() all use the same static area to return the *servent* structure. This static area is only valid until the next one of these functions is called on the same thread.

The *servent* structure is defined in the **netdb.h** include file and contains the following elements:

Element
Description

s_aliases
An array, terminated with a NULL pointer, of alternative names for the service.

s_name
The official name of the service.

s_port The port number of the service.

s_proto The protocol required to contact the service.

Special behavior for C++: To use this function with C++, you must use the _XOPEN_SOURCE_EXTENDED 1 feature test macro.

Returned value

The return value points to data that is overwritten by subsequent calls returning the same data structure.

If successful, getservbyname() returns a pointer to a *servent* structure.

If unsuccessful or End Of File (EOF), getservbyname() returns a NULL pointer.

Related information

- “netdb.h” on page 49
- “endservent() — Close network services information data sets” on page 424
- “getservbyport() — Get a service entry by port” on page 772
- “getservent() — Get the next service entry” on page 773
- “setservent() — Open the network services information data set” on page 1576

getservbyport() — Get a service entry by port

Standards

Standards / Extensions	C or C++	Dependencies
XPG4.2 Single UNIX Specification, Version 3	both	

Format

X/Open:

```
#define _XOPEN_SOURCE_EXTENDED 1
#include <netdb.h>

struct servent *getservbyport(int port, const char *proto);
```

Berkeley sockets:

```
#define _OE_SOCKETS
#include <netdb.h>

struct servent *getservbyport(int port, char *proto);
```

General description

The getservbyport() call searches the */etc/services* or the *tcpip*.ETC.SERVICES data set for the first entry that matches the specified port number and protocol name. If proto is NULL, only the port number must match.

Parameter
Description

port The port number.

proto The protocol name.

The getservbyport() call returns a pointer to a *servent* structure for the port number specified on the call. getservbyname(), getservbyport(), and getservent() all use the same static area to return the *servent* structure. This static area is only valid until the next one of these functions is called on the same thread.

The *servent* structure is defined in the **netdb.h** include file and contains the following elements:

Element
Description

s_aliases
An array, terminated with a NULL pointer, of alternative names for the service.

s_name
The official name of the service.

s_port The port number of the service.

s_proto The protocol required to contact the service.

Special behavior for C++: To use this function with C++, you must use the _XOPEN_SOURCE_EXTENDED 1 feature test macro.

Returned value

The return value points to data that is overwritten by subsequent calls returning the same data structure.

If successful, getservbyport() returns a pointer to a *servent* structure.

If unsuccessful or End Of File (EOF), getservbyport() returns a NULL pointer.

Related information

- "netdb.h" on page 49
- "endservent() — Close network services information data sets" on page 424
- "getservbyname() — Get a server entry by name" on page 770
- "getservent() — Get the next service entry"
- "setservent() — Open the network services information data set" on page 1576

getservent() — Get the next service entry

Standards

Standards / Extensions	C or C++	Dependencies
XPG4.2 Single UNIX Specification, Version 3	both	

Format

X/Open:

```
#define _XOPEN_SOURCE_EXTENDED 1
#include <netdb.h>

struct servent *getservent(void);
```

Berkeley sockets:

```
#define _OE_SOCKETS
#include <netdb.h>

struct servent *getservent(void);
```

General description

The getservent() call reads the next line of the */etc/services* or the *tcpip*.ETC.SERVICES data set.

The getservent() call returns a pointer to the next entry in the */etc/services* or the *tcpip*.ETC.SERVICES data set.

getservbyname(), getservbyport(), and getservent() all use the same static area to return the *servent* structure. This static area is only valid until the next one of these functions is called on the same thread.

The *servent* structure is defined in the **netdb.h** include file and contains the following elements:

Element
Description

s_aliases
An array, terminated with a NULL pointer, of alternative names for the service.

s_name
The official name of the service.

s_port The port number of the service.

s_proto The protocol required to contact the service.

Special behavior for C++: To use this function with C++, you must use the _XOPEN_SOURCE_EXTENDED 1 feature test macro.

Returned value

The return value points to data that is overwritten by subsequent calls returning the same data structure.

If successful, getservent() returns a pointer to a *servent* structure.

If unsuccessful or End Of File (EOF), getservent() returns a NULL pointer.

Related information

- "netdb.h" on page 49
- "endservent() — Close network services information data sets" on page 424
- "getservbyname() — Get a server entry by name" on page 770
- "getservbyport() — Get a service entry by port" on page 772
- "setservent() — Open the network services information data set" on page 1576

getsid() — Get process group ID of session leader

Standards

Standards / Extensions	C or C++	Dependencies
XPG4.2 Single UNIX Specification, Version 3	both	

Format

```
#define _XOPEN_SOURCE_EXTENDED 1
#include <unistd.h>

pid_t getsid(pid_t pid);
```

General description

The getsid() function obtains the process group ID of the process that is the session leader of the process specified by *pid*. If *pid* is 0, the system uses the PID of the process calling getsid().

Returned value

If successful, getsid() returns the process group ID of the session leader of the specified process.

If unsuccessful, getsid() returns (pid_t)-1 and sets errno to one of the following values:

Error Code
Description

EPERM
The process specified by *pid* is not in the same session as the calling process, and the implementation does not allow access to the process group ID of the session leader of that process from the calling process.

ESRCH
There is no process with a process ID equal to *pid*.

Related information

- "unistd.h" on page 82
- "exec functions" on page 436
- "fork() — Create a new process" on page 571
- "getpid() — Get the process ID" on page 752
- "getppid() — Get the parent process ID" on page 754
- "setpgid() — Set process group ID for job control" on page 1565
- "setsid() — Create session, set process group ID" on page 1576

getsockname() — Get the name of a socket

Standards

Standards / Extensions	C or C++	Dependencies
XPG4.2 Single UNIX Specification, Version 3	both	

Format

X/Open:

```
#define _XOPEN_SOURCE_EXTENDED 1
#include <sys/socket.h>

int getsockname(int socket, struct sockaddr *__restrict__ name,
                socklen_t *__restrict__ namelen);
```

Berkeley sockets:

```
#define _OE_SOCKETS
#include <sys/types.h>
#include <sys/socket.h>

int getsockname(int socket, struct sockaddr *name,
                int *namelen);
```

General description

The getsockname() call stores the current name for the socket specified by the *socket* parameter into the structure pointed to by the *name* parameter. It returns the address to the socket that has been bound. If the socket is not bound to an address, the call returns with the family set, and the rest of the structure set to zero. For example, an unbound socket in the Internet domain would cause the name to point to a **sockaddr_in** structure with the *sin_family* field set to AF_INET and all other fields zeroed.

If the actual length of the address is greater than the length of the supplied *sockaddr*, the stored address is truncated. The *sa_len* member of the store structure contains the length of the untruncated address.

Parameter
Description

socket The socket descriptor.

name The address of the buffer into which getsockname() copies the name of *socket*.

namelen
Must initially point to an integer that contains the size in bytes of the storage pointed to by *name*. On return, that integer contains the size required to represent the address of the connecting socket. If this value is larger than the size supplied on input, then the information contained in sockaddr is truncated to the length supplied on input. If *name* is NULL, *namelen* is ignored.

The getsockname() call is often used to discover the port assigned to a socket after the socket has been implicitly bound to a port. For example, an application can call connect() without previously calling bind(). In this case, the connect() call completes the binding necessary by assigning a port to the socket. This assignment can be discovered with a call to getsockname().

Sockets in the AF_INET6 domain: For an AF_INET6 socket, the address is returned in a sockaddr_6 address structure. The sockaddr_in6 structure is defined in the header file **netinet/in.h**.

Special behavior for C++: To use this function with C++, you must use the _XOPEN_SOURCE_EXTENDED 1 feature test macro.

Note: The getsockname() function has a dependency on the level of the Enhanced ASCII Extensions. See “Enhanced ASCII support” on page 2113 for details.

Returned value

If successful, getsockname() returns 0.

If unsuccessful, getsockname() returns -1 and sets errno to one of the following values:

Error Code
Description

EBADF
The *socket* parameter is not a valid socket descriptor.

EFAULT
Using the *name* and *namelen* parameters as specified would result in an attempt to access storage outside of the caller's address space.

ENOBUFS
getsockname() is unable to process the request due to insufficient storage.

ENOTCONN
The socket is not in the connected state.

ENOTSOCK
The descriptor is for a file, not for a socket.

EOPNOTSUPP
The operation is not supported for the socket protocol.

Related information

- "sys/socket.h" on page 74
- "sys/types.h" on page 75
- "accept() — Accept a new connection on a socket" on page 105
- "bind() — Bind a name to a socket" on page 210
- "connect() — Connect a socket" on page 308
- "getpeername() — Get the name of the peer connected to a socket" on page 748
- "socket() — Create a socket" on page 1688

getsockopt() — Get the options associated with a socket

Standards

Standards / Extensions	C or C++	Dependencies
XPG4.2 Single UNIX Specification, Version 3	both	

Format

X/Open:

```
#define _XOPEN_SOURCE_EXTENDED 1
#include <sys/socket.h>

int getsockopt(int socket, int level, int option_name,
               void *__restrict__ option_value,
               socklen_t *__restrict__ option_len);
```

Berkeley sockets:

```
#define _OE_SOCKETS
#include <sys/types.h>
#include <sys/socket.h>

int getsockopt(int socket, int level, int option_name,
               char *option_value,
               int *option_len);
```

IPv6: To include support for IPv6 socket options, add the following code:

```
#define _OPEN_SYS_SOCK_IPV6 1
#include <netinet/in.h>
```

icmp6_filter structure: To include the icmp6_filter structure in your program, add the following code:

```
#define _OPEN_SYS_SOCK_IPV6
#include <netinet/icmp6.h>
```

General description

The getsockopt() call gets options associated with a socket. Not all options are supported by all address families. See each option for details. Options can exist at multiple protocol levels.

Parameter
Description

socket The socket descriptor.

level The level for which the option is set.

option_name
The name of a specified socket option.

option_value
The pointer to option data.

option_len
The pointer to the length of the option data.

When manipulating socket options, you must specify the level at which the option resides and the name of the option. To manipulate options at the socket level, the *level* parameter must be set to SOL_SOCKET as defined in sys/socket.h. To manipulate options at the IPv4 or IPv6 level, the *level* parameter must be set to IPPROTO_IP as defined in sys/socket.h or IPPROTO_IPV6 as defined in netinet/in.h. To manipulate options at any other level, such as the TCP level, supply the appropriate protocol number for the protocol controlling the option. The getprotobyname() call can be used to return the protocol number for a named protocol.

The *option_value* and *option_len* parameters are used to return data used by the particular get command. The *option_value* parameter points to a buffer that is to receive the data requested by the get command. The *option_len* parameter points to the size of the buffer pointed to by the *option_value* parameter. It must be initially set to the size of the buffer before calling getsockopt(). On return it is set to the actual size of the data returned.

All the socket level options except SO_LINGER, SO_RCVTIMEO and SO_SNDTIMEO, expect *option_value* to point to an integer and *option_len* to be set to the size of an integer. When the integer is nonzero, the option is enabled. When it is zero, the option is disabled. The SO_LINGER option expects *option_value* to point to a linger structure as defined in sys/socket.h. This structure is defined in the following example:

```
struct  linger
{
       int     l_onoff;               /* option on/off */
       int     l_linger;              /* linger time */
};
```

The *l_onoff* field is set to zero if the SO_LINGER option is being disabled. A nonzero value enables the option. The *l_linger* field specifies the amount of time to linger on close.

The following options are recognized at the IPv4 level:

Option Description

IP_MULTICAST_IF
(RAW and UDP) Used to get the interface IP address used for sending outbound multicast datagrams. The IP address is passed using the in_addr structure.

IP_MULTICAST_LOOP
(RAW and UDP) Used to determine whether loopback is enabled or disabled. The loopback indicator is passed as an u_char. The value 0 indicates loopback is disabled and the value 1 indicates it is enabled.

IP_MULTICAST_TTL
(RAW and UDP) Returns the IP time-to-live of outgoing multicast datagrams. The TTL value is passed as an u_char.

IP_RECVPKINFO
(RAW and UDP) Indicates whether returning the destination IP address of an incoming packet and the interface over which the packet was received is enabled or disabled. If the setsockopt() function is used to set this option, the set value is returned. The option value is passed as an int. The value 0 indicates the option is disabled and the value 1 indicates the option is enabled. When the option is enabled, the information is returned as IP_PKTINFO ancillary data on recvmsg() function calls.

This option is protected by the _OPEN_SYS_SOCK_EXT4 feature test macro.

The following options are recognized at IPv6 level:

Option Description

IPV6_ADDR_PREFERENCES
(TCP and UDP) Used to get the source address selection preference flags for a given socket. The flags are defined as bits within the 32 bit unsigned integer returned option value. The flags returned will be either the default preference flags or will be the flags previously set by setsockopt(). The constants for the flag bit values are defined in netinet/in.h.

IPV6_CHECKSUM
(RAW) Used to determine if checksum processing is enabled for a RAW (non-ICMPv6) socket. The option value returned is the offset into the user data where the checksum is located. It is passed as int. A value of -1 means the function is disabled.

IPV6_DONTFRAG
(RAW and UDP) This option turns off the automatic inserting of a fragment header in the packet for UDP and raw sockets.

IPV6_DSTOPTS
(RAW and UDP) The application can remove any sticky destination options header by calling setsockopt() for this option with a zero option length.

IPV6_HOPOPTS
(RAW and UDP) The application can remove any sticky hop-by-hop options header by calling setsockopt() for this option with a zero option length.

IPV6_MULTICAST_HOPS
(RAW and UDP) Returns the hop limit value for outbound multicast datagrams. The hop limit value is passed as an int.

IPV6_MULTICAST_IF
(RAW and UDP) Returns the interface index for the interface used for sending outbound multicast datagrams. The interface index is passed as an u_int.

IPV6_MULTICAST_LOOP
(RAW and UDP) Used to determine whether loopback of outgoing multicast packets is enabled or disabled. The loopback indicator is passed as u_int. The value 0 indicates the option is disabled and the value 1 indicates it is enabled.

IPV6_NEXTHOP
(RAW and UDP) Specifies the next hop for the datagram as a socket address structure.

IPV6_RECVDSTOPTS
(RAW and UDP) To receive destination options header this option must be enabled.

IPV6_RECVHOPLIMIT
(RAW, TCP, and UDP) Returns the received hop limit as IPV6_HOPLIMIT ancillary data on recvmsg() function calls. The option value is passed as an int. The value 0 indicates the option is disabled and the value 1 indicates the option is enabled.

IPV6_RECVHOPOPTS
(RAW and UDP) To receive a hop-by-hop options header this option must be enabled.

IPV6_RECVPATHMTU
(RAW and UDP) Returns the path MTU as IPV6_PATHMTU ancillary data on recvmsg() function calls.

IPV6_RECVPKTINFO
(RAW and UDP) Indicates whether returning the destination IP address of an incoming packet and the interface over which the packet was received is enabled or disabled. The option value is passed as an int. The value 0 indicates the option is disabled and the value 1 indicates the option is enabled. When the option is enabled, the information is returned as IPV6_PKTINFO ancillary data on recvmsg() function calls.

IPV6_RECVRTHDR
(RAW and UDP) To receive a routing header this option must be enabled.

IPV6_RECVTCLASS
(RAW, TCP, and UDP) To receive the traffic class this option must be enabled.

IPV6_RTHDR
(RAW and UDP) The application can remove any sticky routing header by calling setsockopt() for this option with a zero option length.

IPV6_RTHDRDSTOPTS
(RAW and UDP) The application can remove any sticky destination options header by calling setsockopt() for this option with a zero option length.

IPV6_TCLASS
(RAW, TCP, and UDP) To specify the traffic class value this option must be enabled.

IPV6_UNICAST_HOPS
(RAW and UDP) Returns the hop limit value for outbound unicast datagrams. The hop limit value is passed as an int.

IPV6_USE_MIN_MTU
(RAW, TCP, and UDP) Indicates whether the IP layer will use the minimum MTU size (1280) for sending packets, bypassing path MTU discovery. The option value is passed as an int. A value of -1 causes the default values for unicast (disabled) and multicast (enabled) destinations to be used. A value of 0 disables this option for unicast and multicast destinations. A value of 1 enables this option for unicast and multicast

destinations and the minimum MTU size will be used. If a setsockopt() call has not been made prior to a getsockopt() call, the default value of -1 is returned.

IPV6_V6ONLY
(RAW, TCP, and UDP) Used to determine whether a socket is restricted to IPv6 communications only. The option value is passed as an int. A non-zero value means the option is enabled (socket can only be used for IPv6 communications). 0 means the option is disabled.

The following option is recognized at ICMPv6 level:

Option Description

ICMP6_FILTER
(RAW) Used to filter ICMPv6 messages. It returns the filter value being used for this socket. It is back in an icmp6_filter structure as defined in netinet/icmp6.h.

The following options are recognized at the socket level:

Option Description

SO_ACCEPTCONN
The socket had a listen() call.

SO_BROADCAST
Toggles the ability to broadcast messages. If this option is enabled, it allows the application to send broadcast messages over *socket*, if the interface specified in the destination supports the broadcasting of packets. This option has no meaning for stream sockets. This option is valid only for the AF_INET domain.

SO_DEBUG
Reports whether debugging information is being recorded. This option stores an int value.

SO_ERROR
Returns any pending error on the socket and clears the error status. You can use SO_ERROR to check for asynchronous errors on connected datagram sockets or for other asynchronous errors (errors that are not returned explicitly by one of the socket calls).

SO_KEEPALIVE
Toggles the TCP keep-alive mechanism for a stream socket. When activated, the keep-alive mechanism periodically sends a packet on an otherwise idle connection. If the remote TCP does not respond to the packet or to retransmissions of the packet, the connection is ended with the error ETIMEDOUT. Processes writing to that socket are notified with a SIGPIPE signal. This option stores an int value. This option is valid only for the AF_INET and AF_INET6 domains.

SO_LINGER
Lingers on close if data is present. When this option is enabled and there is unsent data present when close() is called, the calling application is blocked during the close() call until the data is transmitted or the connection has timed out. If this option is disabled, the TCP/IP address space waits to try to send the data. Although the data transfer is usually successful, it cannot be guaranteed, because the TCP/IP address space

waits only a finite amount of time trying to send the data. The close() call returns without blocking the caller. This option has meaning only for stream sockets.

SO_OOBINLINE
Toggles reception of out-of-band data. When this option is enabled, out-of-band data is placed in the normal data input queue as it is received; it is then available to recv(), recvfrom(), and recvmsg() without the need to specify the MSG_OOB flag in those calls. When this option is disabled, out-of-band data is placed in the priority data input queue as it is received; it is then available to recv(), recvfrom(), and recvmsg() only if the MSG_OOB flag is specified in those calls. This option has meaning only for stream sockets.

_SO_PROPAGATEUSERID
Toggles propagating a user ID (UID) over an AF_UNIX stream socket. When enabled, user (UID) information is extracted from the system when the connect() function is invoked. Then, when the accept() function is invoked, the accepter assumes the identity of the connecter until the accepted socket is closed.

SO_RCVBUF
Reports receive buffer size information. This option stores an int value.

SO_RCVTIMEO

Reports the timeout value with the amount of time an input function waits until it completes.

If a receive operation has blocked for this much time without receiving additional data, it returns with a partial count or errno set to EWOULDBLOCK if no data is received. The default for this option is zero, which indicates that a receive operation does not time out.

SO_REUSEADDR

Toggles local address reuse. When enabled, this option allows local addresses that are already in use to be bound. SO_REUSEADDR alters the normal algorithm used in the bind() call.

The system checks at connect time to ensure that the local address and port do not have the same foreign address and port. The error EADDRINUSE is returned if the association already exists.

After the 'SO_REUSEADDR' option is active, the following situation is supported:

A server can bind() the same port multiple times as long as every invocation uses a different local IP address and the wildcard address INADDR_ANY is used only one time per port.

This option is valid only for the AF_INET and AF_INET6 domains.

SO_SECINFO
Toggles receiving security information. When enabled on an AF_UNIX UDP socket, the recvmsg() function will return security information about the sender of each datagram as ancillary data. This information contains the sender's user ID, uid, gid, and jobname and it is mapped by the secsinfo structure in sys/socket.h.

SO_SNDBUF
Reports send buffer size information. This option stores an int value.

SO_SNDTIMEO

Reports the timeout value specifying the amount of time that an output function blocks due to flow control preventing data from being sent.

If a send operation has blocked for this time, it returns with a partial count or with errno set to EWOULDBLOCK if no data is sent. The default for this option is zero, which indicates that a send operation does not time out.

SO_TYPE

This option returns the type of the socket. On return, the integer pointed to by *option_value* is set to SOCK_STREAM or SOCK_DGRAM. This option is valid for the AF_UNIX, AF_INET and AF_INET6 domains.

Special behavior for C++: To use this function with C++, you must use the _XOPEN_SOURCE_EXTENDED 1 feature test macro.

Returned value

If successful, getsockopt() returns 0.

If unsuccessful, getsockopt() returns -1 and sets errno to one of the following values:

Error Code

Description

EBADF

The *socket* parameter is not a valid socket descriptor.

EFAULT

Using *option_value* and *option_len* parameters would result in an attempt to access storage outside the caller's address space.

EINVAL

The specified option is not valid at the specified socket level.

ENOBUFS

Buffer space is not available to send the message.

ENOPROTOOPT

The *option_name* parameter is unrecognized, or the *level* parameter is not SOL_SOCKET.

ENOSYS

The function is not implemented. You attempted to use a function that is not yet available.

ENOTSOCK

The descriptor is for a file, not for a socket.

EOPNOTSUPP

The operation is not supported by the socket protocol. At least the following options are not supported:

- IPV6_JOIN_GROUP
- IPV6_LEAVE_GROUP
- IP_ADD_SOURCE_MEMBERSHIP
- IP_DROP_SOURCE_MEMBERSHIP
- IP_DROP_MEMBERSHIP
- IP_ADD_MEMBERSHIP
- IP_BLOCK_SOURCE
- IP_UNBLOCK_SOURCE

- MCAST_JOIN_GROUP
- MCAST_LEAVE_GROUP
- MCAST_BLOCK_SOURCE
- MCAST_UNBLOCK_SOURCE
- MCAST_JOIN_SOURCE_GROUP
- MCAST_LEAVE_SOURCE_GROUP

Example

The following are examples of the getsockopt() call. See “setsockopt() — Set options associated with a socket” on page 1578 for examples of how the setsockopt() call options are set.

```
int rc;
int s;
int option_value;
int option_len;
struct linger l;
int getsockopt(int s, int level, int option_name,
char *option_value,
     int *option_len);

⋮
/* Is out-of-band data in the normal input queue? */
option_len = sizeof(int);
rc = getsockopt(
        s, SOL_SOCKET, SO_OOBINLINE, (
char *) &option_value, &option_len);
if (rc == 0)
{
    if (option_len == sizeof(int))
    {
         if (option_value)
            /* yes it is in the normal queue */
         else
            /* no it is not
                  */
    }
}

⋮
/* Do I linger on close? */
option_len = sizeof(l);
rc = getsockopt(
        s, SOL_SOCKET, SO_LINGER, (char *) &l, &option_len);
if (rc == 0)
{
    if (option_len == sizeof(l))
    {
         if (l.l_onoff)
            /* yes I linger */
         else
            /* no I do not  */
    }
}
```

Related information

- “netinet/in.h” on page 53
- “sys/socket.h” on page 74
- “sys/types.h” on page 75
- “bind() — Bind a name to a socket” on page 210
- “close() — Close a file” on page 288
- “getprotobyname() — Get a protocol entry by name” on page 756

- "recvmsg() — Receive messages on a socket and store in an array of message headers" on page 1405
- "sendmsg() — Send messages on a socket" on page 1500
- "setsockopt() — Set options associated with a socket" on page 1578
- "socket() — Create a socket" on page 1688
- For more information about IPv4 socket options, see *z/OS Communications Server: IP Sockets Application Programming Interface Guide and Reference*.
- For more information about IPv6 socket options, see *z/OS Communications Server: IPv6 Network and Appl Design Guide*.

getsourcefilter() — Get source filter

Standards

Standards / Extensions	C or C++	Dependencies
RFC3678	both	z/OS V1.9

Format

```
#define _XOPEN_SYS_SOCK_EXT3
#include <netinet/in.h>

int getsourcefilter(int s, uint32_t interface, struct sockaddr *group,
  socklen_t grouplen, uint32_t *fmode, uint32_t *numsrc,
  struct sockaddr_storage *slist);
```

General description

This function allow applications to get a previously set multicast filtering state for a tuple consisting of socket, interface, and multicast group values.

A multicast filter is described by a filter mode, which is MCAST_INCLUDE or MCAST_EXCLUDE, and a list of source addresses which are filtered.

This function is protocol-independent. It can be on either AF_INET or AF_INET6 sockets of the type SOCK_DGRAM or SOCK_RAW.

If the function is unable to obtain the required storage, control will not return to the caller. Instead the application will terminate due to an out of memory condition (if the reserve stack is available and the caller is not XPLINK), or it will terminate with an abend indicating that storage could not be obtained.

Argument
Description

s Identifies the socket.

interface
Holds the index of the interface.

group Points to either a sockaddr_in structure for IPv4 or a sockaddr_in6 structure for IPv6 that holds the IP multicast address of the group.

grouplen
Gives the length of the sockaddr_in or sockaddr_in6 structure.

fmode Points to an integer that will contain the filter mode on a successful return. The value of this field will be either MCAST_INCLUDE or MCAST_EXCLUDE, which are likewise defined in <netinet/in.h>.

numsrc
It is a pointer that on input, points to the number of source adresses that will fit in the slist array. On return, points to the total number of sources associated with the filter.

slist Points to buffer into which an array of IP addresses of included or excluded (depending on the filter mode) sources will be written. If numsrc was 0 on input, a NULL pointer may be supplied.

Returned value

If successful, the function returns 0. Otherwise, it returns -1 and sets errno to one of the following values.

errno Description

EADDRNOTAVAIL
The tuple consisting of socket, interface, and multicast group values does not exist; or the specified interface address is not multicast capable.

EAFNOSUPPORT
The address family of the input sockaddr is not AF_INET or AF_INET6.

EBADF
s is not a valid socket descriptor.

EINVAL
Interface or group is not a valid address, or the socket s has already requested multicast setsockopt options (refer to z/OS Communications Server: IP Sockets Application Programming Interface Guide and Reference for details.) Or if the group address family is AF_INET and grouplen is not at least size of sockaddr_in or if the group address family is AF_INET6 and grouplen is not at least size of sockaddr_in6 or if grouplen is not at least size of sockaddr_in.

ENXIO
The specified interface index provided in the interface parameter does not exist.

EPROTOTYPE
The socket s is not of type SOCK_DGRAM or SOCK_RAW.

Related information

- “netinet/in.h” on page 53
- “setsourcefilter() — Set source filter” on page 1588

getstablesize() — Get the socket table size

Standards

Standards / Extensions	C or C++	Dependencies
z/OS UNIX	both	

Format

```
#define _OPEN_SYS_SOCK_EXT
#include <sys/socket.h>

int getstablesize(void);
```

General description

Bulk mode sockets are not supported. Do not use this function.

Returned value

If successful, getstablesize() returns the current limit for this process.

If it has not been changed by the maxdesc() function, then the default is returned. The default is the hard limit returned by getrlimit() for RLIMIT_NOFILE. This is the value set by a BPXPRMnn parmlib member on its MAXFILEPROC statement.

There are no errno values defined.

Related information

- "sys/socket.h" on page 74
- "getrlimit() — Get current or maximum resource consumption" on page 765
- "maxdesc() — Get socket numbers to extend beyond the default range" on page 1039

getsubopt() — Parse suboption arguments

Standards

Standards / Extensions	C or C++	Dependencies
XPG4.2 Single UNIX Specification, Version 3	both	

Format

```
#define _XOPEN_SOURCE_EXTENDED 1
#include <stdlib.h>

int getsubopt(char **optionp, char *const *tokens, char **valuep);
```

General description

The getsubopt() function parses suboption arguments in a flag argument that was initially parsed by getopt(). These suboption arguments must be separated by commas and may consist of either a single token, or a token-value pair separated by an equal sign. Because commas delimit suboption arguments in the option string, they are not allowed to be part of the suboption arguments or the value of a suboption argument. Similarly, because the equal sign separates a token from its value, a token must not contain an equal sign.

The getsubopt() function takes the address of a pointer to the option argument string, a vector of possible tokens, and the address of a value string pointer. If the option argument string at *optionp* contains only one suboption argument, getsubopt() updates *optionp* to point to the NULL at the end of the string. Otherwise, it isolates the suboption argument by replacing the comma separator with a NULL, and updates *optionp* to point to the start of the next suboption argument. If the suboption argument has an associated value, getsubopt() updates *valuep* to point to the value's first character. Otherwise it sets *valuep* to a NULL pointer.

The token vector is organized as a series of pointers to strings. The end of the token vector is identified by a NULL pointer.

When getsubopt() returns, if *valuep* is not a NULL pointer, then the suboption argument processed included a value. The calling program may use this information to determine if the presence or lack of a value for the suboption is an error.

Additionally, when getsubopt() fails to match the suboption argument with the tokens in the *tokens* array, the calling program should decide if this is an error, or if the unrecognized option should be passed on to another program.

Because the getsubopt() function returns thread-specific data the getsubopt() function can be used safely from a multithreaded application.

Returned value

If successful, getsubopt() returns the index of the matched token string.

If no token strings were matched, getsubopt() returns -1.

getsubopt() does not return any errno values.

Related information

- “stdlib.h” on page 70
- “getopt() — Command option parsing” on page 745

getsyntx() — Return LC_SYNTAX characters

Standards

Standards / Extensions	C or C++	Dependencies
Language Environment	both	

Format

```
#include <variant.h>

struct variant *getsyntx(void);
```

General description

Determines the encoding of the special characters defined in the LC_SYNTAX category of the current locale, and stores the encoding values in the structure of type *variant*. For details of the variant structure, see “variant.h” on page 84.

Returned value

Returns the pointer to the structure containing the values of the special characters.

If the information about the special characters is not available in the current locale, getsyntx() returns a NULL pointer.

The structure returned is not modified by the program that this function is used in. The structure may be invalidated by calls to the setlocale() function with LC_ALL, LC_CTYPE, LC_COLLATE, or LC_SYNTAX.

Example

CELEBG19

```
/* CELEBG19 */
#include <stdio.h>
#include <stdlib.h>
#include <variant.h>
#include <wchar.h>

int main(void)
{
   struct variant *var;

   var = getsyntx();
   printf("codeset        : %s\n",  var->codeset        );
   printf("backslash      : %3d\n", var->backslash      );
   printf("right_bracket  : %3d\n", var->right_bracket  );
   printf("left_bracket   : %3d\n", var->left_bracket   );
   printf("right_brace    : %3d\n", var->right_brace    );
   printf("left_brace     : %3d\n", var->left_brace     );
   printf("circumflex     : %3d\n", var->circumflex     );
   printf("tilde          : %3d\n", var->tilde          );
   printf("exclamation_mark: %3d\n", var->exclamation_mark);
   printf("number_sign    : %3d\n", var->number_sign    );
   printf("vertical_line  : %3d\n", var->vertical_line  );
   printf("dollar_sign    : %3d\n", var->dollar_sign    );
   printf("commercial_at  : %3d\n", var->commercial_at  );
   printf("grave_accent   : %3d\n", var->grave_accent   );
}
```

Related information

- "Internationalization: Locales and Character Sets" in *z/OS XL C/C++ Programming Guide*
- "locale.h" on page 40
- "variant.h" on page 84
- "setlocale() — Set locale" on page 1552

__get_system_settings() — Retrieves system parameters

Standards

Standards / Extensions	C or C++	Dependencies
z/OS UNIX	both	OS/390 V2R10

Format

```
#define _OPEN_SYS_EXT 1
#include <sys/ps.h>

struct _Optn *__get_system_settings(void);
```

General description

The __get_system_settings() function retrieves system parameter information from the BPXPRM member used during IPL, or updated by the OMVS operator command.

Returned value

If successful, __get_system_settings() returns a pointer to an _Optn structure containing the values set for the BPXPRMxx member process during IPL, or updated by the OMVS operator command.

If unsuccessful, __get_system_settings() returns NULL and may set errno to one of the following values:

Error Code
Description

ENOMEM
Insufficient memory available to allocate _Optn structure.

Related information

- "sys/ps.h" on page 74

gettimeofday(), gettimeofday64() — Get date and time

Standards

Standards / Extensions	C or C++	Dependencies
XPG4.2 Single UNIX Specification, Version 3 Language Environment	both	

Format

```
#define _XOPEN_SOURCE_EXTENDED 1

#undef _ALL_SOURCE
#include <sys/time.h>

int gettimeofday(struct timeval *__restrict__ tp,
                 void *__restrict__ tzp);

#define _ALL_SOURCE
#include <sys/time.h>

int gettimeofday(struct timeval *__restrict__ tp,
                 struct timezone *__restrict__ tzp);
#define _LARGE_TIME_API
#include <time.h>

int gettimeofday64(struct timeval64 *__restrict__ tp,
                   void *__restrict__ tzp);
```

General description

The gettimeofday() function obtains the current time, expressed as seconds and microseconds since 00:00:00 Coordinated Universal Time (UTC), January 1, 1970, and stores it in the timeval structure pointed to by *tp*.

Special behavior for _ALL_SOURCE: The gettimeofday() function has two prototypes. Which one is used depends on whether or not you define the _ALL_SOURCE feature test macro when you compile your program. If _ALL_SOURCE is NOT defined when the C/370 preprocessor processes the <sys/time.h> header, it includes a prototype for gettimeofday() which defines the second argument, *tzp*, as a void pointer and includes a C/370 pragma map statement for a C/370 version of gettimeofday() which ignores *tzp*.

If _ALL_SOURCE is defined, the C/370 preprocessor includes a prototype for gettimeofday() which defines *tzp* as a pointer to a timezone structure and includes a pragma map statement for a C/370 version of gettimeofday() which stores time zone information in the timezone structure to which the second argument points. The timezone structure contains the following members:

```
int tz_minuteswest;   /* Time west of Greenwich in minutes */
int tz_dsttime;       /* Type of DST correction to apply   */
```

When _ALL_SOURCE is defined, the gettimeofday() function:

1. invokes tzset() to set the values of the timezone and daylight external variables.
2. converts the value of the timezone external variable to minutes and stores the converted value, rounded up to the nearest minute, in *tzp*->tz_minuteswest.
3. stores the value of the daylight external variable in *tzp*->tz_dsttime.

The function gettimeofday64() will behave exactly like gettimeofday() except it will support calendar times beyond 03:14:07 UTC on January 19, 2038.

There is no support for an _ALL_SOURCE version of gettimeofday64().

Returned value

If successful, gettimeofday() returns 0.

If overflow occurs, gettimeofday() returns nonzero. Overflow occurs when the current time in seconds since 00:00:00 UTC, January 1, 1970 exceeds the capacity of the *tv_sec* member of the timeval structure pointed to by *tp*. The *tv_sec* member is type time_t.

Related information

- "limits.h" on page 39
- "sys/time.h" on page 75
- "ctime(), ctime64() — Convert time to character string" on page 359
- "ctime_r(), ctime64_r() — Convert time value to date and time character string" on page 362
- "ftime() — Set the date and time" on page 661

getuid() — Get the real user ID

Standards

Standards / Extensions	C or C++	Dependencies
POSIX.1 XPG4 XPG4.2 Single UNIX Specification, Version 3	both	

Format

```
#define _POSIX_SOURCE
#include <unistd.h>

uid_t getuid(void);
```

General description

Finds the real user ID (UID) of the calling process.

Returned value

getuid() returns the found value. It is always successful.

There are no documented errno values.

Example

CELEBG20

```
/* CELEBG20

   This example provides information for your user ID.

 */
#define _POSIX_SOURCE
#include <pwd.h>
#include <sys/types.h>
#include <unistd.h>

main() {
  struct passwd *p;
  uid_t  uid;

  if ((p = getpwuid(uid = getuid())) == NULL)
    perror("getpwuid() error");
  else {
    puts("getpwuid() returned the following info for your userid:");
    printf("  pw_name  : %s\n",       p->pw_name);
    printf("  pw_uid   : %d\n", (int) p->pw_uid);
    printf("  pw_gid   : %d\n", (int) p->pw_gid);
    printf("  pw_dir   : %s\n",       p->pw_dir);
    printf("  pw_shell : %s\n",       p->pw_shell);
  }
}
```

Output

```
  pw_name  : MVSUSR1
  pw_uid   : 25
  pw_gid   : 500
  pw_dir   : /u/mvsusr1
  pw_shell : /bin/sh
```

Related information

- “sys/types.h” on page 75
- “unistd.h” on page 82
- “geteuid() — Get the effective user ID” on page 707
- “seteuid() — Set the effective user ID” on page 1531
- “setreuid() — Set real and effective user IDs” on page 1571
- “setuid() — Set the effective user ID” on page 1592

__getuserid() — Retrieve the active MVS user ID

Standards

Standards / Extensions	C or C++	Dependencies
z/OS UNIX	both	OS/390 V2R9

Format

```
#define _OPEN_SYS_EXT
#include <sys/ps.h>

int __getuserid(char *userid, int userlen);
```

General description

Retrieves the current active user ID for the requester. When successful, the output in user ID will be the active MVS *userid*.

Returned value

If successful, __getuserid() returns 0.

If unsuccessful, __getuserid() returns -1 and sets errno to one of the following values:

Error Code
Description

EINVAL
One of the following error conditions exists:
- The length supplied by *userlen* does not allow enough storage in the string to retrieve the MVS user ID.
- The z/OS UNIX system service returned a failure.

Related information
- "sys/ps.h" on page 74

getutxent() — Read next entry in utmpx database

Standards

Standards / Extensions	C or C++	Dependencies
XPG4.2 Single UNIX Specification, Version 3	both	

Format

```
#define _XOPEN_SOURCE_EXTENDED 1
#include <utmpx.h>

struct utmpx *getutxent(void);
```

General description

The getutxent() function reads in the next entry from the utmpx database. If the database is not already open, it opens it. If it reaches the end of the database, it fails.

The pututxline() function obtains an exclusive lock in the utmpx database on the byte range of the record which is ready to write and releases the lock before returning to its caller. The functions getutxent(), getutxid(), and getutxline() might continue to read and are not affected by pututxline().

Because the getutxent() function returns thread-specific data the getutxent() function can be used safely from a multithreaded application. If multiple threads in the same process open the database, then each thread opens the database with a different file descriptor. The thread's database file descriptor is closed when the calling thread terminates or the endutxent() function is called by the calling thread.

The name of the database file defaults to /etc/utmpx. To process a different database file name use the __utmpxname() function.

For all entries that match a request, the `ut_type` member indicates the type of the entry. Other members of the entry will contain meaningful data based on the value of the `ut_type` member as follows:

EMPTY
No other members have meaningful data.

BOOT_TIME
ut_tv is meaningful.

__RUN_LVL
ut_tv and ut_line are meaningful

OLD_TIME
ut_tv is meaningful.

NEW_TIME
ut_tv is meaningful.

USER_PROCESS
ut_id, ut_user (login name of the user), ut_line, ut_pid, and ut_tv are meaningful.

INIT_PROCESS
ut_id, ut_pid, and ut_tv are meaningful.

LOGIN_PROCESS
ut_id, ut_user (implementation-specific name of the login process), ut_pid, and ut_tv are meaningful.

DEAD_PROCESS
ut_id, ut_pid, and ut_tv are meaningful.

Returned value

If successful, getutxent() returns a pointer to a utmpx structure containing a copy of the requested entry in the user accounting database.

If unsuccessful, getutxent() returns a NULL pointer.

No errors are defined for this function.

Related information

- “utmpx.h” on page 84
- “endutxent() — Close the utmpx database” on page 425
- “getutxid() — Search by ID utmpx database” on page 795
- “getutxline() — Search by line utmpx database” on page 796
- “pututxline() — Write entry to utmpx database” on page 1354
- “setutxent() — Reset to start of utmpx database” on page 1595
- “__utmpxname() — Change the utmpx database name” on page 1971

getutxid() — Search by ID utmpx database

Standards

Standards / Extensions	C or C++	Dependencies
XPG4.2 Single UNIX Specification, Version 3	both	

Format

```
#define _XOPEN_SOURCE_EXTENDED 1
#include <utmpx.h>

struct utmpx *getutxid(const struct utmpx *id);
```

General description

The getutxid() function searches forward from the current point in the utmpx database. If the database is not already open, it opens it. If the `ut_type` value of the `utmpx` structure pointed to by *id* is **BOOT_TIME**, **__RUN_LVL**, **OLD_TIME**, or **NEW_TIME**, then it stops when it finds an entry with a matching `ut_type` value. If the `ut_type` value is **INIT_PROCESS**, **LOGIN_PROCESS**, **USER_PROCESS**, or **DEAD_PROCESS**, then it stops when it finds an entry whose type is one of these four and whose `ut_id` member matches the `ut_id` member of the utmpx structure pointed to by *id*. If the UT_type value is EMPTY, getutxid() fails (returns NULL) without repositioning the utmpx database to the end. If the end of the of the database is reached without a match, getutxid() fails.

The pututxline() function obtains an exclusive lock in the utmpx database on the byte range of the record which is ready to write and releases the lock before returning to its caller. The functions getutxent(), getutxid(), and getutxline() might continue to read and are not affected by pututxline().

Because the getutxid() function returns thread-specific data the getutxid() function can be used safely from a multithreaded application. If multiple threads in the same process open the database, then each thread opens the database with a different file descriptor. The thread's database file descriptor is closed when the calling thread terminates or the endutxent() function is called by the calling thread.

The name of the database file defaults to /etc/utmpx. To process a different database file name use the __utmpxname() function.

For all entries that match a request, the `ut_type` member indicates the type of the entry. Other members of the entry will contain meaningful data based on the value of the `ut_type` member as follows:

EMPTY
No other members have meaningful data.

BOOT_TIME
ut_tv is meaningful.

__RUN_LVL
ut_tv and ut_line are meaningful

OLD_TIME
ut_tv is meaningful.

NEW_TIME
ut_tv is meaningful.

USER_PROCESS
ut_id, ut_user (login name of the user), ut_line, ut_pid, and ut_tv are meaningful.

INIT_PROCESS
ut_id, ut_pid, and ut_tv are meaningful.

LOGIN_PROCESS
ut_id, ut_user (implementation-specific name of the login process), ut_pid, and ut_tv are meaningful.

DEAD_PROCESS
ut_id, ut_pid, and ut_tv are meaningful.

Returned value

If successful, getutxid() returns a pointer to a utmpx structure containing a copy of the requested entry in the user accounting database.

If unsuccessful, getutxid() returns a NULL pointer.

No errors are defined for this function.

Related information

- "utmpx.h" on page 84
- "endutxent() — Close the utmpx database" on page 425
- "getutxent() — Read next entry in utmpx database" on page 793
- "getutxline() — Search by line utmpx database"
- "pututxline() — Write entry to utmpx database" on page 1354
- "setutxent() — Reset to start of utmpx database" on page 1595
- "__utmpxname() — Change the utmpx database name" on page 1971

getutxline() — Search by line utmpx database

Standards

Standards / Extensions	C or C++	Dependencies
XPG4.2 Single UNIX Specification, Version 3	both	

Format

```
#define _XOPEN_SOURCE_EXTENDED 1
#include <utmpx.h>

struct utmpx *getutxline(const struct utmpx *line);
```

General description

The getutxline() function searches forward from the current point in the utmpx database until it finds an entry of the type **LOGIN_PROCESS** or **USER_PROCESS** which also has a *ut_line* value matching that in the utmpx structure pointed to by argument *line*. If the database is not already open, it opens it. If it reaches the end of the database, it fails.

The pututxline() function obtains an exclusive lock in the utmpx database on the byte range of the record which is ready to write and releases the lock before returning to its caller. The functions getutxent(), getutxid(), and getutxline() might continue to read and are not affected by pututxline().

Because the getutxline() function returns thread-specific data the getutxline() function can be used safely from a multithreaded application. If multiple threads in the same process open the database, then each thread opens the database with a different file descriptor. The thread's database file descriptor is closed when the calling thread terminates or the endutxent() function is called by the calling thread.

The name of the database file defaults to /etc/utmpx. To process a different database file name use the __utmpxname() function.

The functions getutxent(), getutxid(), and getutxline() cache the last entry read from the database. For this reason, to use getutxline() function to search for multiple occurrences, it is necessary to zero out the utmpx structure pointed to by the return value from these functions.

For all entries that match a request, the `ut_type` member indicates the type of the entry. Other members of the entry will contain meaningful data based on the value of the `ut_type` member as follows:

EMPTY
No other members have meaningful data.

BOOT_TIME
ut_tv is meaningful.

__RUN_LVL
ut_tv and ut_line are meaningful

OLD_TIME
ut_tv is meaningful.

NEW_TIME
ut_tv is meaningful.

USER_PROCESS
ut_id, ut_user (login name of the user), ut_line, ut_pid, and ut_tv are meaningful.

INIT_PROCESS
ut_id, ut_pid, and ut_tv are meaningful.

LOGIN_PROCESS
ut_id, ut_user (implementation-specific name of the login process), ut_pid, and ut_tv are meaningful.

DEAD_PROCESS
ut_id, ut_pid, and ut_tv are meaningful.

Returned value

If successful, getutxline() returns a pointer to a utmpx structure containing a copy of the requested entry in the user accounting database.

If unsuccessful, getutxline() returns a NULL pointer.

No errors are defined for this function.

Related information

- "utmpx.h" on page 84
- "endutxent() — Close the utmpx database" on page 425
- "getutxent() — Read next entry in utmpx database" on page 793
- "getutxid() — Search by ID utmpx database" on page 795
- "pututxline() — Write entry to utmpx database" on page 1354
- "setutxent() — Reset to start of utmpx database" on page 1595
- "__utmpxname() — Change the utmpx database name" on page 1971

getw() — Get a machine word from a stream

Standards

Standards / Extensions	C or C++	Dependencies
XPG4 XPG4.2	both	

Format

```
#define _XOPEN_SOURCE
#include <stdio.h>

int getw(FILE *stream);
```

General description

The getw() function reads the next word from the *stream*. The size of the word is the size of an int, and varies from machine to machine. The getw() function presumes no special alignment in the file.

The getw() function may mark the *st_atime* field of the file associated with *stream* for update. The *st_atime* field will be marked for update by the first successful execution of fgetc(), fgets(), fread(), getc(), getchar(), gets(), fscanf() or scanf() using *stream* that returns data not supplied by a prior call to ungetc().

Note:

This function is kept for historical reasons. It was part of the Legacy Feature in Single UNIX Specification, Version 2, but has been withdrawn and is not supported as part of Single UNIX Specification, Version 3. New applications should use character-based input functions to replace getw() for portability.

If it is necessary to continue using this function in an application written for Single UNIX Specification, Version 3, define the feature test macro _UNIX03_WITHDRAWN before including any standard system headers. The macro exposes all interfaces and symbols removed in Single UNIX Specification, Version 3.

Returned value

If successful, getw() returns the next word from the input stream pointed to by *stream*. If the stream is at End Of File (EOF), the End Of File indicator for the stream is set and getw() returns EOF. If a read error occurs, the error indicator for the stream is set, getw() returns EOF and sets errno to indicate the error.

Refer to "fgetc() — Read a character" on page 526 for errno values.

Because the representation of EOF is a valid integer, applications wishing to check for errors should use ferror() and feof().

Related information

- "stdio.h" on page 68
- "fopen() — Open a file" on page 565
- "fwrite() — Write items" on page 676
- "putw() — Put a machine word on a stream" on page 1356

getwc() — Get a wide character

Standards

Standards / Extensions	C or C++	Dependencies
ISO C Amendment XPG4 XPG4.2 C99 Single UNIX Specification, Version 3 Language Environment	both	

Format

```
#include <wchar.h>

wint_t getwc(FILE *stream);

#define _OPEN_SYS_UNLOCKED_EXT 1
#include <wchar.h>

wint_t getwc_unlocked(FILE *stream);
```

General description

Obtains the next multibyte character from `stdin`, converts it to a wide character, and advances the associated file position indicator for `stdin`.

The getwc() function is equivalent to the fgetwc() function. Therefore, the argument should never be an expression with side effects.

The behavior of this wide-character function is affected by the LC_CTYPE category of the current locale. If you change the category, undefined results can occur.

Using non-wide-character functions with getwc() results in undefined behavior. This happens because getwc() processes a whole multibyte character and does not expect to be "within" such a character. In addition, getwc() expects state information to be set already. Because functions like fgetc() and fputc() do not obey such rules, their results fail to meet the assumptions made by getwc().

getwc() has the same restriction as any read operation for a read immediately following a write or a write immediately following a read. Between a write and a subsequent read, there must be an intervening flush or reposition. Between a read and a subsequent write, there must also be an intervening flush or reposition unless an EOF has been reached.

getwc_unlocked() is functionally equivalent to getwc() with the exception that it is not thread-safe. This function can safely be used in a multithreaded application if and only if it is called while the invoking thread owns the (FILE*) object, as is the case after a successful call to either the flockfile() or ftrylockfile() function.

Returned value

Returns the next wide character from the input stream pointed to by *stream* or else the function returns WEOF.

If there is an error, getwc() sets the error indicator. If the EOF is encountered, it sets the EOF indicator. If an encoding error is encountered, it sets EILSEQ in errno.

Use ferror() or feof() to determine whether an error or an EOF condition occurred. Note that EOF is only reached when an attempt is made to read past the last byte of data. Reading up to and including the last byte of data does *not* turn on the EOF indicator.

Example

CELEBG21

```
/* CELEBG21 */
#include <errno.h>
#include <stdio.h>
#include <stdlib.h>
#include <wchar.h>

int main(void)
{
   FILE    *stream;
   wint_t   wc;

   if ((stream = fopen("myfile.dat", "r")) == NULL) {
      printf("Unable to open file.");
      exit(1);
   }

   errno = 0;
   while ((wc = getwc(stream)) != WEOF)
      printf("wc=0x%lx\n", wc);

   if (errno == EILSEQ) {
      printf("An invalid wide character was encountered.\n");
      exit(1);
   }

   fclose(stream);
}
```

Related information

- “stdio.h” on page 68
- “wchar.h” on page 85
- “fgetwc() — Get next wide character” on page 532

getwchar() — Get a wide character

Standards

Standards / Extensions	C or C++	Dependencies
ISO C Amendment XPG4 XPG4.2 C99 Single UNIX Specification, Version 3 Language Environment	both	

Format

```
#include <wchar.h>

wint_t getwchar(void)

#define _OPEN_SYS_UNLOCKED_EXT 1;
#include <wchar.h>

wint_t getwchar_unlocked(void)
```

General description

The getwchar() function is equivalent to getwc() with the argument `stdin`.

The behavior of this wide-character function is affected by the LC_CTYPE category of the current locale. If you change the category, undefined results can occur.

getwchar_unlocked() is functionally equivalent to getwchar() with the exception that it is not thread-safe. This function can safely be used in a multithreaded application if and only if it is called while the invoking thread owns the (FILE*) object, as is the case after a successful call to either the flockfile() or ftrylockfile() function.

Returned value

Returns the next wide character from the input stream pointed to by stdin or else the function returns WEOF. If the stream is at EOF, the EOF indicator for the stream is set and fgetwc() returns WEOF. If a read error occurs, the error indicator for the stream is set and fgetwc() returns WEOF. If an encoding error occurs, the value of the macro EILSEQ is stored in errno and WEOF is returned.

Use ferror() or feof() to determine whether an error or an EOF condition occurred. Note that EOF is only reached when an attempt is made to read past the last byte of data. Reading up to and including the last byte of data does *not* turn on the EOF indicator.

Example

CELEBG22

```
/* CELEBG22 */
#include <errno.h>
#include <stdio.h>
#include <stdlib.h>
#include <wchar.h>
```

```
int main(void)
{
   wint_t   wc;

   errno = 0;
   while ((wc = getwchar()) != WEOF)
      printf("wc=0x%X\n", wc);

   if (errno == EILSEQ) {
      printf("An invalid wide character was encountered.\n");
      exit(1);
   }
}
```

Related information

- "wchar.h" on page 85
- "fgetwc() — Get next wide character" on page 532
- "getwc() — Get a wide character" on page 799

getwd() — Get the current working directory

Standards

Standards / Extensions	C or C++	Dependencies
XPG4.2 Single UNIX Specification, Version 3	both	

Format

```
#define _XOPEN_SOURCE_EXTENDED 1
#include <unistd.h>

char *getwd(char *path_name);
```

General description

The getwd() function determines an absolute path name of the current working directory of the calling process, and copies that path name into the array pointed to by *path_name* argument.

If the length of the path name of the current working directory is greater than (**PATH_MAX**+1) including the NULL byte, getwd() fails and returns a NULL pointer.

For portability to implementations conforming to earlier versions of the standards, getcwd() is preferred over this function.

Note: The getwd() function has been moved to the Legacy Option group in Single UNIX Specification, Version 3 and may be withdrawn in a future version. The getcwd() function is preferred for portability.

Returned value

If successful, getwd() returns a pointer to the string containing the absolute path name of the current working directory.

If unsuccessful, getwd() returns a NULL pointer and the contents of the array pointed to by *path_name* are undefined.

There are no errno values defined.

Related information

- “unistd.h” on page 82
- “getcwd() — Get path name of the working directory” on page 697

getwmccoll() — Get next collating element from wide string

Standards

Standards / Extensions	C or C++	Dependencies
C Library	both	

Format

```
#include <collate.h>

collel_t getwmccoll(wchar_t **src);
```

General description

If the object pointed to by *src* is not a NULL pointer, the getwmccoll() library function determines the longest sequence of wide characters in the array pointed to by *str* that constitute a valid multi-wide-character collating element. It then produces the value of type `collel_t` corresponding to that collating element. The object pointed to by *src* is assigned the address just past the last wide character of the multi-wide-character collating element processed.

Returned value

If successful, getwmccoll() returns the value of type `collel_t` that represents the collating element found.

If the object pointed to by *src* is a NULL pointer or if it points to a NULL wide character, getwmccoll() returns 0.

If the object pointed to by *src* points to a non-valid wide character, getwmccoll() returns -1 and sets errno to EILSEQ.

Related information

- “collate.h” on page 19
- “wchar.h” on page 85
- “cclass() — Return characters in a character class” on page 243
- “collequiv() — Return a list of equivalent collating elements” on page 295
- “collorder() — Return list of collating elements” on page 297
- “collrange() — Calculate the range list of collating elements” on page 298
- “colltostr() — Return a string for a collating element” on page 299
- “getmccoll() — Get next collating element from string” on page 734
- “ismccollel() — Identify a multicharacter collating element” on page 914
- “maxcoll() — Return maximum collating element” on page 1039
- “strtocoll() — Return collating element for string” on page 1763

givesocket() — Make the specified socket available

Standards

Standards / Extensions	C or C++	Dependencies
z/OS UNIX	both	

Format

```
#define _OPEN_SYS_SOCK_EXT
#include <sys/socket.h>

int givesocket(int d, struct clientid *clientid);
```

General description

The givesocket() call makes the specified socket available to a takesocket() call issued by another program. Any socket can be given. Typically, givesocket() is used by a master program that obtains sockets by means of accept() and gives them to application programs that handle one socket at a time.

Parameter
Description

d The descriptor of a socket to be given to another application.

clientid A pointer to a client ID structure specifying the program to which the socket is to be given.

To pass a socket, the giving program first calls givesocket() with the client ID structure filled in as follows:

The clientid structure:

```
struct clientid {
   int domain;
   union  {
     char name[8];
     struct {
       int NameUpper;
       pid_t pid;
     } c_pid;
   } c_name;
   char subtaskname[8];

   struct  {
     char type;
     union  {
        char specific[19];
        struct  {
          char unused[3];
          int SockToken;
        } c_close;
     } c_func;
   } c_reserved;
};
```

Element
Description

domain The domain of the input socket descriptor.

c_name.name
If the *clientid* was set by a getclientid() call, *c_name.name* can be

- set to the application program's address space name, left-justified and padded with blanks. The application program can run in the same address space as the master program, in which case this field is set to the master program's address space.
- set to blanks, so any z/OS address space can take the socket.

subtaskname

If the *clientid* was set by a getclientid() call, *subtaskname* can be

- set to the task identifier of the taker. This, combined with a *c_name.name* value, allows only a process with this *c_name.name* and *subtaskname* to take the socket.
- set to blanks. If *c_name.name* has a value and *subtaskname* is blank, any task with that *c_name.name* can take the socket.

c_pid.pid

If the *clientid* was set by a __getclientid() call, *c_pid.pid* should be set to the process id (PID) of the taker, so only a process with that PID can take the socket. The *subtaskname* field is ignored when the *c_pid* has a value.

c_reserved.type

When set to SO_CLOSE, this indicates the socket should be automatically closed by givesocket(), and a unique socket identifying token is to be returned in *c_close.SockToken*. The *c_close.SockToken* should be passed to the taking program to be used as input to takesocket() instead of the socket descriptor. The now closed socket descriptor could be re-used by the time the takesocket() is called, so the *c_close.SockToken* should be used for takesocket().

When set to _SO_SELECT, this indicates that the application intends to block on the select() for exception, waiting for the takesocket() to occur before closing the socket. If *c_reserved.type* is set to _SO_SELECT and the caller of givesocket() closes the socket before it has been taken, the connection will be severed. _SO_SELECT also allows select() to return exception status if select() is done after the socket was taken with takesocket().

When set to zero, this indicates that the application will not be calling select() to coordinate with the taker of the socket. Either the socket is not going to be closed or the giver and taker have some other method of coordination for the giver to know when the taker has called takesocket(). Note that if select() for exception is called before takesocket(), the select will return when takesocket() is called but if select() is called after takesocket(), it will hang. _SO_SELECT should be used if select() is going to be called by the giver.

Also, if the given socket is closed before the takesocket() is issued, it is possible for that socket descriptor number to be reused in the giver's process. A sequence of accept(), givesocket(), and close() calls issued several times before any takesocket() calls can result in several sockets with the same descriptor number waiting to be taken. In this case, the oldest given socket will be taken; that is, in FIFO order. Note that if select() is called when there are several given sockets with the same descriptor number waiting to be taken, select() will operate on the current active socket for that descriptor. It effectively waits for the last (newest) given socket to be taken; that is, in LIFO order.

c_close.SockToken
The unique socket identifying token returned by givesocket() to be used as input to takesocket(), instead of the socket descriptor when *c_reserved.type* has been set to SO_CLOSE.

c_reserved
Specifies binary zeros if an automatic close of a socket is not to be done by givesocket().

Using name and subtaskname for givesocket/takesocket:

1. The giving program calls getclientid() to obtain its client ID. The giving program calls givesocket() to make the socket available for a takesocket() call. The giving program passes its client ID along with the descriptor of the socket to be given to the taking program by the taking program's startup parameter list.
2. The taking program calls takesocket(), specifying the giving program's client ID and socket descriptor.
3. Waiting for the taking program to take the socket, the giving program uses select() to test the given socket for an exception condition. When select() reports that an exception condition is pending, the giving program calls close() to free the given socket.
4. If the giving program closes the socket before a pending exception condition is indicated, the connection is immediately reset, and the taking program's call to takesocket() is unsuccessful. Calls other than the close() call issued on a given socket return -1, with errno set to EBADF.

Note: For backward compatibility, a client ID can point to the struct client ID structure obtained when the target program calls getclientid(). In this case, only the target program, and no other programs in the target program's address space, can take the socket.

Using process id (PID) for givesocket/takesocket:

1. The giving program calls __getclientid() to obtain its client ID. The giving program sets the *c_pid.pid* in the clientid structure to the PID of the taking program that will take the socket (that is, issue the takesocket() call). This ensures only a process that has obtained the giver's PID can take the specified socket. If the giving program wants the socket to be automatically closed by givesocket(), *c_reserved.type* should be set to SO_CLOSE. The giving program calls givesocket() to make the socket available for a takesocket() call. The giving program passes its client ID, the descriptor of the socket to be given, and the giving program's PID to the taking program by the taking program's startup parameter list.
2. The taking program sets the *c_pid.pid* in the clientid structure to the PID of the giving program to identify the process from which the socket is to be taken. If the *c_reserved.type* field was set to SO_CLOSE on givesocket(), the *c_close.SockToken* should be used as input to the takesocket() instead of the normal socket descriptor. The taking program calls takesocket(), specifying the giving program's client ID and either the socket descriptor or *c_close.SockToken*.
3. If the *c_reserved.type* field in the clientid structure was set to SO_CLOSE on the givesocket() call, the socket is closed and the giving program does not have to wait for the taking program to issue the takesocket(). Otherwise, steps 3 and 4 of "Using name and subtaskname for givesocket/takesocket" should be followed.

Returned value

If successful, givesocket() returns 0.

If unsuccessful, givesocket() returns -1 and sets errno to one of the following values:

Error Code
  Description

EBADF
  The *d* parameter is not a valid socket descriptor. The socket has already been given.

EFAULT
  Using the *clientid* parameter as specified would result in an attempt to access storage outside the caller's address space.

EINVAL
  The *clientid* parameter does not specify a valid client identifier or the clientid domain does not match the domain of the input socket descriptor.

Related information

- "sys/socket.h" on page 74
- "accept() — Accept a new connection on a socket" on page 105
- "close() — Close a file" on page 288
- "getclientid() — Get the identifier for the calling application" on page 691
- "listen() — Prepare the server for incoming client requests" on page 972
- "select(), pselect() — Monitor activity on files or sockets and message queues" on page 1470
- "takesocket() — Acquire a socket from another program" on page 1812

glob() — Generate path names matching a pattern

Standards

Standards / Extensions	C or C++	Dependencies
XPG4 XPG4.2 Single UNIX Specification, Version 3	both	

Format

```
#define _XOPEN_SOURCE
#include <glob.h>

int glob(const char *__restrict__ pattern, int flags,
        int (*errfunc)(const char *epath, int eerrno),
        glob_t *__restrict__ pglob);
```

General description

The glob() function is a path name generator that implements the rules defined in topic about pattern matching notation in *X/Open CAE Specification, Commands and Utilities, Issue 4, Version 2*, with optional support for rule 3 in the topic about patterns used for file name expansion.

The structure *glob_t* is defined in the header <glob.h> and includes at least the following members:

gl_pathc
Count of paths matched by *pattern*.

gl_pathv
Pointer to a list of matched file names.

gl_offs
Slots to reserve at the beginning of *gl_pathv*.

The argument *pattern* is a pointer to a path name pattern to be expanded. The glob() function matches all accessible path names against this pattern and develops a list of all path names that match. In order to have access to a path name, glob() requires search permission on every component of a path except the last, and read permission on each directory of any file name component of *pattern* that contains any of the following special characters:

```
*       ?       [
```

The glob() function stores the number of matched path names into *pglob->gl_pathc* and a pointer to a list of pointers to path names into *pglob->gl_pathv*. The path names are in sort order as defined by the current setting of the LC_COLLATE category, see *X/Open CAE Specification, System Interface Definitions, Issue 4, Version 2* Section 5.3.2 , LC_COLLATE. The first pointer after the last path name is a NULL pointer. If the pattern does not match any path names, the returned number of matched paths is set to 0, and the contents of *pglob->gl_pathv* are implementation-dependent.

It is the caller's responsibility to create the structure pointed to by *pglob*. The glob() function allocates other space as needed, including the memory pointed to by *gl_pathv*.

The *flags* argument is used to control the behavior of glob() The value of *flags* is a bitwise inclusive-OR of zero or more of the following constants, which are defined in the header <glob.h>:

GLOB_APPEND
Append path names generated to the ones from a previous call to glob().

GLOB_DOOFFS
Make use of *pglob->gl_offs*. If this flag is set, *pglob->gl_offs* is used to specify how many NULL pointers to add to the beginning of *pglob->gl_pathv*. In other words, *pglob->gl_pathv* will point to *pglob->gl_offs* NULL pointers, followed by *pglob->gl_pathc* path name pointers, followed by a NULL pointer.

GLOB_ERR
Causes glob() to return when it encounters a directory that it cannot open or read. Ordinarily, glob() continues to find matches.

GLOB_MARK
Each path name that is a directory that matches *pattern* has a slash appended.

GLOB_NOCHECK
Support rule 3 in the XCU specification, Section 2.13.3 , Patterns Used for Filename Expansion. If *pattern* does not match any path name, then glob() returns a list consisting of only *pattern*, and the number of matched path names is 1.

GLOB_NOESCAPE
Disable backslash escaping.

GLOB_NOSORT
Ordinarily, glob() sorts the matching path names according to the current setting of the LC_COLLATE category, see the XBD specification, Section 5.3.2 , LC_COLLATE. When this flag is used the order of path names returned is unspecified.

The GLOB_APPEND flag can be used to append a new set of path names to those found in a previous call to glob(). The following rules apply when two or more calls to glob() are made with the same value of *pglob* and without intervening calls to globfree():

1. The first such call must not set GLOB_APPEND. All subsequent calls must set it.
2. All calls must set GLOB_DOOFFS, or all must not set it.
3. After the second call, *pglob->gl_pathv* points to a list containing the following:
   a. Zero or more NULL pointers, as specified by GLOB_DOOFFS and *pglob->gl_offs*.
   b. Pointers to the path names that were in the *pglob->gl_pathv* list before the call, in the same order as before.
   c. Pointers to the new path names generated by the second call, in the specified order.
4. The count returned in *pglob->gl_pathc* will be the total number of path names from the two calls.
5. The application can change any of the fields after a call to glob(). If it does, it must reset them to the original value before a subsequent call, using the same *pglob* value, to globfree() or glob() with the GLOB_APPEND flag.

If, during the search, a directory is encountered that cannot be opened or read and *errfunc* is not a NULL pointer, glob() calls (**errfunc*()) with two arguments:

1. The *epath* argument is a pointer to the path that failed.
2. The *eerrno* argument is the value of errno from the failure, as set by opendir(), readdir() or stat(). (Other values may be used to report other errors not explicitly documented for those functions.)

Returned value

If successful, glob() returns 0. The argument *pglob->gl_pathc* returns the number of matched path names and the argument *pglob->gl_pathv* contains a pointer to a NULL-terminated list of matched and sorted path names. However, if *pglob->gl_pathc* is 0, the content of *pglob->gl_pathv* is undefined.

If glob() terminates due to an error, it returns one of the following nonzero constants defined in <glob.h> as error return values for glob():

GLOB_ABORTED
The scan was stopped because GLOB_ERR was set or (**errfunc*()) returned nonzero.

GLOB_NOMATCH
The pattern does not match any existing path name, and GLOB_NOCHECK was notset in *flags*.

GLOB_NOSPACE
An attempt to allocate memory failed.

If (**errfunc*()) is called and returns nonzero, or if the GLOB_ERR flag is set in *flags*, glob() stops the scan and returns GLOB_ABORTED after setting *gl_pathc* and *gl_pathv* in *pglob* to reflect the paths already scanned. If GLOB_ERR is not set and either *errfunc* is a NULL pointer or (**errfunc*()) returns 0, the error is ignored.

Related information

- “glob.h” on page 32
- “exec functions” on page 436
- “fnmatch() — Match file name or path name” on page 564
- “opendir() — Open a directory” on page 1158
- “readdir() — Read an entry from a directory” on page 1382
- “stat() — Get file information” on page 1720
- “wordexp() — Perform shell word expansions” on page 2081

globfree() — Free storage allocated by glob()

Standards

Standards / Extensions	C or C++	Dependencies
XPG4 XPG4.2 Single UNIX Specification, Version 3	both	

Format

```
#define _XOPEN_SOURCE
#include <glob.h>

void globfree(glob_t *pglob);
```

General description

The globfree() function frees storage associated with *pglob* by a previous call to glob().

Returned value

globfree() returns no values.

Related information

- “glob.h” on page 32
- “glob() — Generate path names matching a pattern” on page 807

gmtime(), gmtime64() — Convert time to broken-down UTC time

Standards

Standards / Extensions	C or C++	Dependencies
ISO C POSIX.1 XPG4 XPG4.2 C99 Single UNIX Specification, Version 3 Language Environment	both	

Format

```
#include <time.h>

struct tm *gmtime(const time_t *timer);
```

```
#define _LARGE_TIME_API
#include <time.h>

struct tm *gmtime64(const time64_t *timer);
```

General description

Converts the calendar time pointed to by *timer* into a broken-down time, expressed as Coordinated Universal Time (UTC).

[2]

The value pointed to by *timer* is usually obtained by a call to the time() function.

The relationship between a time in seconds since the Epoch used as an argument to gmtime() and the tm structure (defined in the <time.h> header) is that the result is as specified in the expression given in the definition of seconds since the Epoch, where the names in the structure and in the expression correspond.

The function gmtime64() will behave exactly like gmtime() except it will break down a time64_t value pointing to a calendar time beyond 03:14:07 UTC on January 19, 2038 with a limit of 23:59:59 UTC on December 31, 9999.

Returned value

Returns a pointer to a *tm* structure containing the broken-down time, expressed in Coordinated Universal Time (UTC) corresponding to calendar time pointed to by *timer*. The fields in *tm* are shown in Table 16 on page 80. If the calendar time pointed to by *timer* cannot be converted to broken-down time (in UTC), gmtime() returns a NULL pointer.

Error code
Description

EOVERFLOW
The result cannot be represented.

Notes:

1. The range (0-60) for *tm_sec* allows for as many as one leap second.
2. The gmtime() and localtime() functions may use a common, statically allocated buffer for the conversion. Each call to one of these functions may alter the result of the previous call.
3. The calendar time returned by the time() function begins at the epoch, which was at 00:00:00 Coordinated Universal Time (UTC), January 1, 1970.

Example

CELEBG23

[2] Coordinated Universal Time (UTC) was formerly known as Greenwich Mean Time (GMT).

```
/* CELEBG23

   This example uses the &gmtime. function to convert a
   time_t representation to a Coordinated Universal Time
   character string and then converts it to a printable string
   using &asct..

 */
#include <stdio.h>
#include <time.h>

int main(void)
{
   time_t ltime;
   time(&ltime);
   printf ("Coordinated Universal Time is %s\n",
            asctime(gmtime(&ltime)));
}
```

Output

```
Coordinated Universal Time (UTC) is Fri Jun 16 21:01:44 2006
```

Related information

- "time.h" on page 79
- "asctime(), asctime64() — Convert time to character string" on page 182
- "asctime_r(), asctime64_r() — Convert date and time to a character string" on page 184
- "ctime(), ctime64() — Convert time to character string" on page 359
- "ctime_r(), ctime64_r() — Convert time value to date and time character string" on page 362
- "gmtime_r(), gmtime64_r() — Convert a time value to broken-down UTC time"
- "localdtconv() — Date and time formatting convention inquiry" on page 981
- "localtime(), localtime64() — Convert time and correct for local time" on page 984
- "localtime_r(), localtime64_r() — Convert time value to broken-down local time" on page 986
- "mktime(), mktime64() — Convert local time" on page 1078
- "strftime() — Convert to formatted time" on page 1743
- "time(),time64() — Determine current UTC time" on page 1878
- "tzset() — Set the time zone" on page 1936

gmtime_r(), gmtime64_r() — Convert a time value to broken-down UTC time

Standards

Standards / Extensions	C or C++	Dependencies
Single UNIX Specification, Version 2 Single UNIX Specification, Version 3 Language Environment	both	OS/390 V2R8

Format

```
#define _XOPEN_SOURCE 500
#include <time.h>

struct tm *gmtime_r(const time_t *__restrict__ clock,
                     struct tm *__restrict__ result);
```

```
#define _LARGE_TIME_API
#include <time.h>

struct tm *gmtime64 _r(const time64_t *__restrict__ clock,
                      struct tm *__restrict__ result);
```

General description

The gmtime_r() function converts the calendar time pointed to by *clock* into a broken-down time expressed as Coordinated Universal Time (UTC). The broken-down time is stored in the structure referred to by *result*. The gmtime_r() function also returns the address of the same structure.

The relationship between a time in seconds since the Epoch used as an argument to gmtime() and the tm structure (defined in the <time.h> header) is that the result is as specified in the expression given in the definition of seconds since the Epoch, where the names in the structure and in the expression correspond.

The function gmtime64_r() will behave exactly like gmtime_r() except it will break down a time64_t value pointing to a calendar time beyond 03:14:07 UTC on January 19, 2038 with a limit of 23:59:59 UTC on December 31, 9999.

Returned value

If successful, gmtime_r() returns the address of the structure pointed to by the argument *result*.

If an error is detected or UTC is not available, gmtime_r() returns a NULL pointer.

There are no documented errno values.

Error Code
Description

EOVERFLOW
The result cannot be represented.

Related information

- “time.h” on page 79
- “asctime(), asctime64() — Convert time to character string” on page 182
- “asctime_r(), asctime64_r() — Convert date and time to a character string” on page 184
- “ctime(), ctime64() — Convert time to character string” on page 359
- “ctime_r(), ctime64_r() — Convert time value to date and time character string” on page 362
- “gmtime(), gmtime64() — Convert time to broken-down UTC time” on page 810
- “localdtconv() — Date and time formatting convention inquiry” on page 981
- “localtime(), localtime64() — Convert time and correct for local time” on page 984
- “localtime_r(), localtime64_r() — Convert time value to broken-down local time” on page 986
- “mktime(), mktime64() — Convert local time” on page 1078
- “strftime() — Convert to formatted time” on page 1743
- “time(),time64() — Determine current UTC time” on page 1878
- “tzset() — Set the time zone” on page 1936

grantpt() — Grant access to the slave pseudoterminal device

Standards

Standards / Extensions	C or C++	Dependencies
XPG4.2 Single UNIX Specification, Version 3	both	

Format

```
#define _XOPEN_SOURCE_EXTENDED 1
#include <stdlib.h>

int grantpt(int fildes);
```

General description

The grantpt() function changes the mode and ownership of the slave pseudoterminal device. *fildes* should be the file descriptor of the corresponding master pseudoterminal. The user ID of the slave is set to the real UID of the calling process and the group ID is set to the group ID associated with the group name specified by the installation in the TTYGROUP() initialization parameter. The permission mode of the slave pseudoterminal is set to readable and writable by the owner, and writable by the group.

You can provide secure connections by either using grantpt() and unlockpt(), or by simply issuing the first open against the slave pseudoterminal from the first userid or process that opened the master terminal.

Returned value

If successful, grantpt() returns 0.

If unsuccessful, grantpt() returns -1 and sets errno to one of the following values:

Error Code
Description

EACCES
The slave pseudoterminal was opened before grantpt(), or a grantpt() was already issued. In either case, slave pseudoterminal permissions and ownership have already been updated. If you use grantpt() to change slave pseudoterminal permissions, you must issue grantpt() between the master open and the first pseudoterminal open, and grantpt() can only be issued once.

EBADF
The *fildes* argument is not a valid open file descriptor.

EINVAL
The *fildes* argument is not associated with a master pseudoterminal device.

ENOENT
The slave pseudoterminal device was not found during lookup.

Related information

- "stdlib.h" on page 70
- "open() — Open a file" on page 1152
- "ptsname() — Get name of the slave pseudoterminal device" on page 1346

- “unlockpt() — Unlock a pseudoterminal master and slave pair” on page 1964

hcreate() — Create hash search tables

Standards

Standards / Extensions	C or C++	Dependencies
XPG4 XPG4.2 Single UNIX Specification, Version 3	both	

Format

```
#define _XOPEN_SOURCE
#include <search.h>

int hcreate(size_t nel);
```

General description

The hcreate() function allocates sufficient space for a hash table containing *nel* elements, and must be called before hsearch() is used.

The *nel* argument is an estimate of the maximum number of entries that the table will contain. This number may be adjusted upward by hcreate() for the actual table allocation in order to obtain certain mathematically favorable circumstances.

Threading Behavior: see “hsearch() — Search hash tables” on page 817.

Returned value

If successful, hcreate() returns nonzero.

If hcreate() cannot allocate sufficient space for the table, it returns 0 and sets errno to one of the following values:

Error Code
 Description

ENOMEM
 Insufficient storage space is available.

Related information

- “search.h” on page 62
- “bsearch() — Search arrays” on page 220
- “hdestroy() — Destroy hash search tables” on page 816
- “hsearch() — Search hash tables” on page 817
- “lsearch() — Linear search and update” on page 1019
- “malloc() — Reserve storage block” on page 1031
- “strcmp() — Compare strings” on page 1731
- “tsearch() — Binary tree search” on page 1920

hdestroy() — Destroy hash search tables

Standards

Standards / Extensions	C or C++	Dependencies
XPG4 XPG4.2 Single UNIX Specification, Version 3	both	

Format

```
#define _XOPEN_SOURCE
#include <search.h>

void hdestroy(void);
```

General description

The hdestroy() function disposes of the search table, and may be followed by another call to hcreate(). After the call to hdestroy(), the data can no longer be considered accessible.

Threading Behavior: see "hsearch() — Search hash tables" on page 817.

Returned value

hdestroy() returns no values.

Related information

- "search.h" on page 62
- "bsearch() — Search arrays" on page 220
- "hcreate() — Create hash search tables" on page 815
- "hsearch() — Search hash tables" on page 817
- "lsearch() — Linear search and update" on page 1019
- "malloc() — Reserve storage block" on page 1031
- "strcmp() — Compare strings" on page 1731
- "tsearch() — Binary tree search" on page 1920

__heaprpt() — Obtain dynamic heap storage report

Standards

Standards / Extensions	C or C++	Dependencies
Language Environment	both	

Format

```
#include <stdlib.h>

typedef struct{long __uheap_size;
               long __uheap_bytes_alloc;
               long __uheap_bytes_free;
              } hreport_t;

int __heaprpt(hreport_t *heap_report_structure);
```

General description

__heaprpt() gets statistics about the application's storage utilization and places them in the area pointed to by the *heap_report_structure* argument. The storage report is similar in content to the user heap storage report that is generated with the RPTSTG(ON) runtime option.

To use this function, the calling program must obtain storage where the user's heap storage report will be stored. The address of this storage is passed as an argument to __heaprpt().

Returned value

If successful, __heaprpt() fills the struct hreport_t with the user's heap storage report information.

If the address is not valid, __heaprpt() returns -1 and sets errno to EFAULT.

Example

```
#include <stdlib.h>
#include <stdio.h>

int main(void)
{
  hreport_t * strptr;

  strptr = (hreport_t *) malloc(sizeof(hreport_t));

  if (__heaprpt(strptr) != 0)
     perror("__heaprpt() error");

  else
  {
     printf("Total amount of user heap storage    : %ld\n",
            strptr->__uheap_size);
     printf("Amount of user heap storage in use   : %ld\n",
            strptr->__uheap_bytes_alloc);
     printf("Amount of available user heap storage: %ld\n",
            strptr->__uheap_bytes_free);
  }

}
```

Related information

- "stdlib.h" on page 70

hsearch() — Search hash tables

Standards

Standards / Extensions	C or C++	Dependencies
XPG4 XPG4.2 Single UNIX Specification, Version 3	both	

Format

```
#define _XOPEN_SOURCE
#include <search.h>

ENTRY *hsearch(ENTRY item, ACTION action);
```

General description

The hsearch() function is a hash-table search routine. It returns a pointer into a hash table indicating the location at which an entry can be found. The *item* argument is a structure of type `ENTRY` (defined in the `<search.h>` header) containing two pointers: *item.key* points to the comparison key (a `char *`), and *item.data* (a `void *`) points to any other data to be associated with that key. The comparison function used by hsearch() is `strcmp().` The *action* argument is a member of an enumeration type `ACTION` indicating the disposition of the entry if it cannot be found in the table. *ENTER* indicates that the item should be inserted in the table at an appropriate point. *FIND* indicates that no entry should be made.

Threading Behavior: The hcreate() function allocates a piece of storage for use as the hash table. This storage is not exposed to the user, and is referred to by all threads. In other words, these functions operate on one hash table global to the process. The library serializes access to the table and attendant data across threads using an internal mutex.

Returned value

hsearch() returns a NULL pointer if either the action is *FIND* and the item could not be found or the action is *ENTER* and the table is full.

If an error occurs, hsearch() sets errno to one of the following values:

Error Code
Description

ENOMEM
Insufficient storage space is available.

Related information

- “search.h” on page 62
- “bsearch() — Search arrays” on page 220
- “hcreate() — Create hash search tables” on page 815
- “hdestroy() — Destroy hash search tables” on page 816
- “lsearch() — Linear search and update” on page 1019
- “malloc() — Reserve storage block” on page 1031
- “strcmp() — Compare strings” on page 1731
- “tsearch() — Binary tree search” on page 1920

htonl() — Translate address host to network long

Standards

Standards / Extensions	C or C++	Dependencies
XPG4.2 Single UNIX Specification, Version 3	both	

Format

XPG4.2

```
#define _XOPEN_SOURCE_EXTENDED 1
#include <arpa/inet.h>

in_addr_t htonl(in_addr_t hostlong);
```

SUSV3

```
#define _POSIX_C_SOURCE 200112L
#include <apar/inet.h>

uint32_t htonl(uint32_t hostlong);
```

Berkeley Sockets

```
#define _OE_SOCKETS
#include <sys/types.h>
#include <arpa/inet.h>
#include <netinet/in.h>

unsigned long htonl(unsigned long a);
```

General description

The htonl() function translates a long integer from host byte order to network byte order.

Parameter
Description

a The unsigned long integer to be put into network byte order.

in_addr_t *hostlong*
Is typed to the unsigned long integer to be put into network byte order.

Notes:

1. For MVS, host byte order and network byte order are the same.
2. Since this function is implemented as a macro, you need one of the feature test macros and the inet header file.

Returned value

htonl() returns the translated long integer.

Related information

- “arpa/inet.h” on page 17
- “netinet/in.h” on page 53
- “sys/types.h” on page 75
- “htons() — Translate an unsigned short integer into network byte order” on page 820
- “ntohl() — Translate a long integer into host byte order” on page 1150
- “ntohs() — Translate an unsigned short integer into host byte order” on page 1151

htons() — Translate an unsigned short integer into network byte order

Standards

Standards / Extensions	C or C++	Dependencies
XPG4.2 Single UNIX Specification, Version 3	both	

Format

XPG4.2

```
#define _XOPEN_SOURCE_EXTENDED 1
#include <arpa/inet.h>

in_port_t htons(in_port_t hostshort);
```

SUSV3

```
#define _POSIX_C_SOURCE 200112L
#include <arpa/inet.h>

uint16_t htons(uint16_t hostshort);
```

Berkeley Sockets

```
#define _OE_SOCKETS
#include <sys/types.h>
#include <netinet/in.h>
#include <arpa/inet.h>

unsigned short htons(unsigned short a);
```

General description

The htons() function translates a short integer from host byte order to network byte order.

Parameter
Description

a The unsigned short integer to be put into network byte order.

in_port_t *hostshort*
Is typed to the unsigned short integer to be put into network byte order.

Notes:

1. For MVS, host byte order and network byte order are the same.
2. Since this function is implemented as a macro, you need one of the feature test macros and the inet header file.

Returned value

htons() returns the translated short integer.

Related information

- “arpa/inet.h” on page 17
- “netinet/in.h” on page 53
- “sys/types.h” on page 75
- “htonl() — Translate address host to network long” on page 818

- “ntohl() — Translate a long integer into host byte order” on page 1150
- “ntohs() — Translate an unsigned short integer into host byte order” on page 1151

hypot(), hypotf(), hypotl() — Calculate the square root of the squares of two arguments

Standards

Standards / Extensions	C or C++	Dependencies
SAA XPG4 XPG4.2 C99 Single UNIX Specification, Version 3 C++ TR1 C99	both	z/OS V1R7

Format

SAA:

```
#include <math.h>

double hypot(double side1, double side2);
```

SAA compiler option: LANGLVL(EXTENDED), LANGLVL(SAA), or LANGLVL(SAAL2)

XPG4:

```
#define _XOPEN_SOURCE
#include <math.h>

double hypot(double side1, double side2);
```

C99:

```
#define _ISOC99_SOURCE
#include <math.h>

float hypotf(float side1, float side2);
long double hypotl(long double side1, long double side2);
```

C++ TR1 C99:

```
#define _TR1_C99
#include <math.h>

float hypot(float side1, float side2);
long double hypot(long double side1, long double side2);
```

General description

The hypot() family of functions calculates the length of the hypotenuse of a right-angled triangle based on the lengths of two sides *side1* and *side2*. A call to hypot() is equal to:

```
 sqrt(side1* side1 + side2 * side2);
```

Note: The following table shows the viable formats for these functions. See "IEEE binary floating-point" on page 94 for more information about IEEE Binary Floating-Point.

Function	SPC	Hex	IEEE
hypot	X	X	X
hypotf		X	X
hypotl		X	X

Restriction: The hypotf() function does not support the _FP_MODE_VARIABLE feature test macro.

Returned value

The hypot() family of functions returns the calculated length of the hypotenuse.

If the correct value is outside the range of representable values, ±HUGE_VAL is returned, according to the sign of the value. The value of the macro ERANGE is stored in errno, to show the calculated value is out of range. If the correct value would cause an underflow, zero is returned and the value of the macro ERANGE is stored in errno.

Special behavior for IEEE: If successful, The hypot() family of functions returns the calculated length of the hypotenuse. If the correct value overflows, hypot() sets errno to ERANGE and returns HUGE_VAL.

Example

CELEBH01

```
/* CELEBH01

   This example calculates the hypotenuse of a right-angled
   triangle with sides of 3.0 and 4.0.

 */
#include <math.h>
#include <stdio.h>

int main(void)
{
   double x, y, z;

   x = 3.0;
   y = 4.0;
   z = hypot(x,y);

   printf("The hypotenuse of the triangle with sides %lf and %lf"
          " is %lf\n", x, y, z);
}
```

Output

```
The hypotenuse of the triangle with sides 3.000000 and 4.000000 is 5.000000
```

Related information

- "math.h" on page 44
- "sqrt(), sqrtf(), sqrtl() — Calculate square root" on page 1713

hypotd32(), hypotd64(), hypotd128() — Calculate the square root of the squares of two arguments

Standards

Standards / Extensions	C or C++	Dependencies
C/C++ DFP	both	z/OS V1.11

Format

```
#define __STDC_WANT_DEC_FP__
#include <math.h>

_Decimal32  hypotd32(_Decimal32 x, _Decimal32 y);
_Decimal64  hypotd64(_Decimal64 x, _Decimal64 y);
_Decimal128 hypotd128(_Decimal128 x, _Decimal128 y);

_Decimal32  hypot(_Decimal32 x, _Decimal32 y);     /* C++ only */
_Decimal64  hypot(_Decimal64 x, _Decimal64 y);     /* C++ only */
_Decimal128 hypot(_Decimal128 x, _Decimal128 y);    /* C++ only */
```

General description

The hypot() family of functions calculates the length of the hypotenuse of a right-angled triangle based on the lengths of two sides x and y. A call to hypot() is equal to:

```
sqrt(x* x + y * y);
```

Notes:

1. These functions work in IEEE decimal floating-point format. See "IEEE decimal floating-point" on page 95IEEE Decimal Floating-Point for more information.
2. To use IEEE decimal floating-point, the hardware must have the Decimal Floating-Point Facility installed.

Returned value

If successful, The hypot() family of functions returns the calculated length of the hypotenuse.

If the correct value overflows, hypot() sets errno to ERANGE and returns HUGE_VAL_D32, HUGE_VAL_D64, or HUGE_VAL_D128 accordingly.

Example

```
/* CELEBH03

   This example illustrates the hypotd64() function.

*/

#define  __STDC_WANT_DEC_FP__
#include <math.h>
#include <stdio.h>

void main(void)
{
   _Decimal64 x, y, z;

   x = 3.0DD;
   y = 4.0DD;
   z = hypotd64(x,y);

   printf("The hypotenuse of a triangle with sides %Df and %Df"
          " is %Df\n", x, y, z);
}
```

Related information

- "math.h" on page 44
- "sqrtd32(), sqrtd64(), sqrtd128() — Calculate square root" on page 1715

ibmsflush() — Flush the application-side datagram queue

Standards

Standards / Extensions	C or C++	Dependencies
z/OS UNIX	both	

Format

```
#define _OPEN_SYS_SOCK_EXT
#include <sys/socket.h>

int ibmsflush(int s);
```

General description

Bulk mode sockets are not supported. This function always returns 0.

Returned value

ibmsflush() always returns 0.

Related information

- "sys/socket.h" on page 74

iconv() — Code conversion

Standards

Standards / Extensions	C or C++	Dependencies
XPG4 XPG4.2 Single UNIX Specification, Version 3	both	

Format

```
#include <iconv.h>

size_t iconv(iconv_t cd, char **__restrict__ inbuf,
     size_t *__restrict__ inbytesleft, char **__restrict__ outbuf,
             size_t *__restrict__ outbytesleft);
```

General description

Converts a sequence of characters, indirectly pointed to by *inbuf*, from one encoded character set into a sequence of corresponding characters in another encoded character set. The resulting character sequence is then stored into the array indirectly pointed to by *outbuf*. The encoded character sets are those specified in the iconv_open() call that returned the conversion descriptor, *cd*. If the descriptor refers to the state-dependent encoding, then before it is first used, the *cd* descriptor is in its initial shift state.

The *inbuf* argument points to a variable that points to the first character in the input buffer. *inbytesleft* indicates the number of bytes to the end of the buffer to be converted. The *outbuf* argument points to a variable that points to the first character in the output buffer. *outbytesleft* indicates the number of available bytes to the end of the buffer.

If the output character set refers to the state-dependent encoding—if it contains the multibyte characters with shift-states—the conversion descriptor *cd* is placed in its initial state by a call for which *inbuf* is a NULL pointer, or for which *inbuf* points to a NULL pointer. When iconv() is called in this way, and if *outbuf* is not a NULL pointer or a pointer to a NULL pointer, and *outbytesleft* points to a positive value, iconv() places in the output buffer the byte sequence to change the output buffer to the initial shift state. If the output buffer is not large enough to hold the entire reset sequence, iconv() fails, and sets errno to E2BIG. Subsequent calls with *inbuf* as other than a NULL pointer or a pointer to a NULL pointer cause conversion from the current state of the conversion descriptor.

If a sequence of input bytes does not form a valid character in the specified encoded character set, conversion stops after the previous successfully converted character, and iconv() sets errno to EILSEQ. If the input buffer ends with an incomplete character or shift sequence, conversion stops after the previous successfully converted bytes, and iconv() sets errno to EINVAL. If the output buffer is not large enough to hold the entire converted input, conversion stops just before the input bytes that would cause the output buffer to overflow.

The variable pointed to by *inbuf* is updated to point to the byte following the last byte of a successfully converted character. The value pointed to by *inbytesleft* is decremented to reflect the number of bytes still not converted in the input buffer. The variable pointed to by *outbuf* is updated to point to the byte following the last byte of converted output data. The value pointed to by *outbytesleft* is decremented to reflect the number of bytes still available in the output buffer. For state-dependent encoding, the conversion descriptor is updated to reflect the shift state in effect at the end of the last successfully converted byte sequence.

If iconv() encounters a character in the input buffer that is valid, but for which a conversion is not defined in the conversion descriptor, *cd*, then iconv() performs a nonidentical conversion on this character. The conversion is implementation-defined.

The <iconv.h>header file declares the `iconv_t` type that is a pointer to the object capable of storing the information about the converters used to convert characters in one coded character set to another. For state-dependent encoding, the object must be capable of storing the encoded information about the current shift state.

Special considerations for bidirectional language support: If the _BIDION environment variable is set to TRUE, iconv() performs bidirectional layout transformation to the converted characters. The required attributes for bidirectional layout transformation can be specified using the environment variable _BIDIATTR (eg. export _BIDIATTR="@ls typeoftext=visual:implicit, orientation=ltr:ltr,numerals=nominal:national"). For a detailed description of the bidirectional layout transformation, see “Bidirectional Language Support” in *z/OS XL C/C++ Programming Guide*. If the environment variable _BIDIATTR is not set, the default values will be used.

iconv() can perform bidirectional layout transformation while converting the data from the fromCodePage to the toCodePage. Bidirectional layout transformation will take place only if bidirectional language support is activated, see “iconv_open() — Allocate code conversion descriptor” on page 829 for more information about activating bidirectional layout transformation. In case iconv encounters any error in input or output buffers in the bidirectional part it will bypass the bidirectional layout transformation and continue its normal function as usual.

Special behavior for POSIX C: In the POSIX environment, a conversion descriptor returned from a successful iconv_open() may be used safely within a single thread. In addition, it may be opened on one thread, used on a second thread (iconv()), and closed (iconv_open()) on a third thread. However, you must ensure correct cross-thread sequencing and synchronization (that is: iconv_open(), followed by optional iconv() calls, followed by iconv_close()). The use of a shared conversion descriptor by iconv() across multiple threads may result in undefined behavior.

Returned value

If successful, iconv() updates the variables pointed to by the arguments to reflect the extent of the conversion and returns the number of nonidentical conversions performed.

If the entire string in the input buffer is converted, the value pointed to by *inbytesleft* will be 0. If the input conversion is stopped because of any conditions mentioned above, the value pointed to by *inbytesleft* will be nonzero and errno is set to indicate the condition.

If an error occurs, iconv() returns (size_t)-1 and sets errno to one of the following values:

Error Code
: **Description**

EBADF
: *cd* is not a valid descriptor.

ECUNNOENV
: A CUN_RS_NO_UNI_ENV error was issued by Unicode Conversion Services.

  See *z/OS Unicode Services User's Guide and Reference* documentation for user action.

ECUNNOCONV
: A CUN_RS_NO_CONVERSION error was issued by Unicode Conversion Services.

  See *z/OS Unicode Services User's Guide and Reference* documentation for user action.

ECUNNOTALIGNED
: A CUN_RS_TABLE_NOT_ALIGNED error was issued by Unicode Conversion Services.

See *z/OS Unicode Services User's Guide and Reference* documentation for user action.

ECUNERR

Function iconv() encountered an unexpected error while using Unicode Conversion Services.

See message EDC6258 for additional information.

EILSEQ
Input conversion stopped due to an input byte that does not belong to the input codeset.

EINVAL
Input conversion stopped due to an incomplete character or shift sequence at the end of the input buffer.

E2BIG Input conversion stopped due to lack of space in the output buffer.

Example

CELEBI01

```
/* CELEBI01

   This example converts an array of characters coded in encoded character
   set IBM-1047 to an array of characters coded in encoded character set
   IBM-037.
   Input is in inbuf, output will be in outbuf.

 */
#include <iconv.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

main ()
{
            char   *inptr;  /* Pointer used for input buffer  */
            char   *outptr; /* Pointer used for output buffer */
            char    inbuf[20] =
                  "ABCDEFGH!@#$1234";
                                            /* input buffer */
   unsigned char    outbuf[20];   /* output buffer  */
   iconv_t          cd;     /* conversion descriptor          */
   size_t           inleft; /* number of bytes left in inbuf  */
   size_t           outleft;/* number of bytes left in outbuf */
   int              rc;     /* return code of iconv()         */


   if ((cd = iconv_open("IBM-037", "IBM-1047")) == (iconv_t)(-1)) {
      fprintf(stderr, "Cannot open converter from %s to %s\n",
                                        "IBM-1047", "IBM-037");
      exit(8);
   }

   inleft = 16;
   outleft = 20;
   inptr = inbuf;
   outptr = (char*)outbuf;

   rc = iconv(cd, &inptr, &inleft, &outptr, &outleft);
   if (rc == -1) {
      fprintf(stderr, "Error in converting characters\n");
      exit(8);
   }
   iconv_close(cd);
}
```

Related information

- "Internationalization: Locales and Character Sets" in *z/OS XL C/C++ Programming Guide*
- "iconv.h" on page 32
- "locale.h" on page 40
- "iconv_close() — Deallocate code conversion descriptor"
- "iconv_open() — Allocate code conversion descriptor" on page 829
- "setlocale() — Set locale" on page 1552

iconv_close() — Deallocate code conversion descriptor

Standards

Standards / Extensions	C or C++	Dependencies
XPG4 XPG4.2 Single UNIX Specification, Version 3	both	

Format

```
#include <iconv.h>

int iconv_close(iconv_t cd);
```

General description

Deallocates the conversion descriptor *cd* and all other associated resources allocated by the iconv_open() function. For an illustration of using iconv_open(), see "Example" on page 827.

Returned value

If successful, iconv_close() returns 0.

If unsuccessful, iconv_close() returns -1 and sets errno to one of the following values:

Error Code
Description

EBADF
cd is not a valid descriptor.

Related information

- "Internationalization: Locales and Character Sets" in *z/OS XL C/C++ Programming Guide*
- "iconv.h" on page 32
- "locale.h" on page 40
- "iconv() — Code conversion" on page 824
- "iconv_open() — Allocate code conversion descriptor" on page 829
- "setlocale() — Set locale" on page 1552

iconv_open() — Allocate code conversion descriptor

Standards

Standards / Extensions	C or C++	Dependencies
XPG4 XPG4.2 Single UNIX Specification, Version 3	both	

Format

```
#include <iconv.h>

iconv_t iconv_open(const char *tocode, const char *fromcode);
```

General description

Performs all the initialization needed to convert characters from the encoded character set specified in the array pointed to by the *fromcode* argument to the encoded character set specified in the array pointed to by the *tocode* argument.

The conversion descriptor relates the two encoded character sets.

For state-dependent encodings, the conversion descriptor will be in an encoded-character-set-dependent initial shift state, ready for immediate use with iconv(). The conversion descriptor remains valid until it is closed with iconv_close().

Settings of *fromcode*, *tocode*, and their permitted combinations are implementation-dependent.

Note: The iconv() family of functions has been modified to utilize character conversion services provided by Unicode Services. The iconv_open(), iconv() and iconv_close()'s function interfaces will remain unchanged except for the addition of four new errno values and two new environment variables described in the following paragraphs.

There are differences in externals between the iconv() family of functions and Unicode Services. However, these differences will be managed by the iconv() family of functions except where noted in the *z/OS XL C/C++ Compiler and Runtime Migration Guide for the Application Programmer*. All conversions listed in the section of the *z/OS XL C/C++ Programming Guide* entitled "Code Set Converters Supplied" will continue to work as they did prior to the integration of Unicode Services, as long as _ICONV_TECHNIQUE is left undefined. If the application does not define the _ICONV_TECHNIQUE environment variable, iconv_open() uses a default value of LMREC.

Unicode Services supports conversions between thousands of additional character sets not listed in the *z/OS XL C/C++ Programming Guide*. A complete list of conversions supported by Unicode Services can be found in tables 25 an tables 26 in the *z/OS Unicode Services User's Guide and Reference*. To set up a conversion using iconv_open() for any of the character sets listed in these tables, the user needs to use a character string representing the CCSID's for *fromcode* and *tocode*. For example, to set up a conversion from CCSID 00256 to CCSID 00870 using conversion technique R, the user would set the _ICONV_TECHNIQUE environment variable to R and call iconv_open() as follows:

```
cd = iconv_open("00870", "00256");
```

and continue to use iconv() and iconv_close() as in previous releases.

iconv() uses the following environment variables.

_ICONV_UCS2
Tells iconv_open(Y, X) what type of conversion method to setup when there is a choice between "direct" conversion from X to Y and "indirect" X to UCS-2 to Y.

_ICONV_UCS2_PREFIX
Tells iconv_open() what z/OS dataset name prefix to use to find UCS-2 tables if they cannot be found in the HFS.

_ICONV_MODE
Selects the behavior mode for iconv_open(), iconv() and iconv_close()..

_ICONV_TECHNIQUE
This is the technique value used while using Unicode Conversion Services. For more information regarding the Unicode Conversion Services technique value, refer to Chapter 3 - Creating a Unicode Environment section of the *z/OS Unicode Services User's Guide and Reference*.

For illustration of using iconv_close(), see “Example” on page 827.

Special considerations for bidirectional language support: Performs all the initialization needed to activate the bidirectional layout transformation to be used by iconv. The following three conditions must be satisfied to enable the bidirectional layout transformation:

1. The _BIDION environment variable must be set to TRUE.
2. The current locale environment at iconv_open() time must be an Arabic or Hebrew locale (eg. Ar_AA or Iw_IL).
3. The conversion code set must be an Arabic or Hebrew code set.

Conversion code sets differ in the following three cases:

1. Case fromCodeSet is UCS-2 and toCodeSet is single byte code set. In this case toCodeSet must be an Arabic or Hebrew code set.
2. Case fromCodeSet is single byte code set and toCodeSet is UCS-2. In this case fromCodeSet must be an Arabic or Hebrew code set.
3. Case both fromCodeSet and toCodeSet are single byte code sets. In this case toCodeSet must be an Arabic or Hebrew code set.

iconv_open( ) checks for the existence of the environment variable _BIDIATTR to get the bidirectional layout transformation attributes. It will use default values in case _BIDIATTR is not defined, is unset, or in case of the existence of some erroneous values in the _BIDIATTR environment variable. The default values are code set dependent according to the Arabic or Hebrew code set used. For the Arabic 420 code set the default values will be: orientation RTL, type of text visual, shaping shaped, numerals national and swapping on. For the Hebrew 424 code set the default values will be: orientation RTL, type of text visual and swapping on. For the rest of the Arabic code sets the default values will be: orientation RTL, type of text implicit, shaping nominal, numerals national and swapping on.

iconv_open() uses the following environment variables.

_BIDION

Tells iconv_open() whether to activate bidirectional handling of the converted data or not. _BIDION can be assigned either the value TRUE, if you want to turn on bidirectional layout transformation, or the value FALSE, if you want to turn off the BiDi layout transformation. Bidirectional layout transformation can also be turned off if the variable_BIDION is not defined in the environment.

_BIDIATTR

Holds the bidirectional layout transformation attributes which will be used later by iconv, _BIDIATTR will be read only in iconv_open() time. The _BIDIATTR environment variable is in the form of input/output pairs separated by colon, at the beginning of the string there is an @ that identifies the beginning of the attributes list, then followed by the attributes in the form of `<attribute_name1>=<input1>:<output1>, <attribute_name2>=<input2>:<output2> ..... (eg. export _BIDIATTR="@ls typeoftext=visual:implicit,orientation=ltr:ltr, numerals=nominal:national").`

Returned value

If successful, iconv_open() returns a conversion descriptor.

If unsuccessful, iconv_open() returns (iconv_t)-1 and sets errno to one of the following values:

Error Code

Description

EINVAL

The conversion between encoded character sets specified is not supported.

ECUNNOENV

A CUN_RS_NO_UNI_ENV error was issued by Unicode Conversion Services.

See *z/OS Unicode Services User's Guide and Reference* for user action.

ECUNNOCONV

A CUN_RS_NO_CONVERSION error was issued by Unicode Conversion Services.

See *z/OS Unicode Services User's Guide and Reference* for user action.

ECUNNOTALIGNED

A CUN_RS_TABLE_NOT_ALIGNED error was issued by Unicode Conversion Services.

See *z/OS Unicode Services User's Guide and Reference* for user action.

ECUNERR

Function iconv() encountered an unexpected error while using Unicode Conversion Services.

Refer to message EDC6258 for additional information.

Related information

- "Internationalization: Locales and Character Sets" in *z/OS XL C/C++ Programming Guide*

- "iconv.h" on page 32
- "locale.h" on page 40
- "iconv() — Code conversion" on page 824
- "iconv_close() — Deallocate code conversion descriptor" on page 828
- "setlocale() — Set locale" on page 1552

if_freenameindex() — Free the memory allocated by if_nameindex()

Standards

Standards / Extensions	C or C++	Dependencies
RFC2553 Single UNIX Specification, Version 3	both	z/OS V1R4

Format

```
#define _OPEN_SYS_SOCK_IPV6
#include <net/if.h>

void if_freenameindex(struct if_nameindex *ptr);
```

SUSV3:

```
#define _POSIX_C_SOURCE 200112L
#include <net/if.h>

void if_freenameindex(struct if_nameindex *ptr);
```

General description

The if_freenameindex() function frees the memory allocated by if_nameindex(). The *ptr* argument must be a pointer that was returned by if_nameindex().

Returned value

No return value is defined.

Related information

- "if_indextoname() — Map a network interface index to its corresponding name"
- "if_nameindex() — Return all network interface names and indexes" on page 833
- "if_nametoindex() — Map a network interface name to its corresponding index" on page 834

if_indextoname() — Map a network interface index to its corresponding name

Standards

Standards / Extensions	C or C++	Dependencies
RFC2553 Single UNIX Specification, Version 3	both	z/OS V1R4

Format

```
#define _OPEN_SYS_SOCK_IPV6
#include <net/if.h>

char *if_indextoname(unsigned int ifindex, char *ifname);
```

SUSV3:

```
#define _POSIX_C_SOURCE 200112L
#include <net/if.h>

char *if_indextoname(unsigned int ifindex, char *ifname);
```

General description

The if_indextoname() function maps an interface index to its corresponding interface name. When this function is called, *ifname* must point to a buffer of at least IF_NAMESIZE bytes into which the interface name corresponding to interface index *ifindex* is returned. Otherwise, the function shall return a NULL pointer and set errno to indicate the error.

Returned value

Error Code
Description

EINVAL
The *ifindex* parameter was zero, or the *ifname* parameter was NULL, or both.

ENOMEM
Insufficient storage is available to obtain the information for the interface name.

ENXIO
The *ifindex* does not yield an interface name.

Related information

- “if_freenameindex() — Free the memory allocated by if_nameindex()” on page 832
- “if_nameindex() — Return all network interface names and indexes”
- “if_nametoindex() — Map a network interface name to its corresponding index” on page 834

if_nameindex() — Return all network interface names and indexes

Standards

Standards / Extensions	C or C++	Dependencies
RFC2553 Single UNIX Specification, Version 3	both	z/OS V1R4

Format

```
#define _OPEN_SYS_SOCK_IPV6
#include <net/if.h>

struct if_nameindex *if_nameindex(void);
```

SUSV3:

```
#define _POSIX_C_SOURCE 200112L
#include <net/if.h>

struct if_nameindex *if_nameindex(void);
```

General description

The if_nameindex() function returns an array of if_nameindex structures, one structure per interface. The end of the array is indicated by a structure with an if_index of zero and an if_name of NULL.

The if_nameindex structure holds the information about a single interface and is defined as a result of including the <net/if.h> header.

```
struct if_nameindex {
   unsigned int if_index;  /* 1, 2, ... */
   char       *if_name;   /* null terminated name:  "le0", ... */
};
```

The memory used for this array of structures along with the interface names pointed to by the if_name members is obtained dynamically. This memory is freed by calling the if_freenameindex() function.

Returned value

When successful, if_nameindex() returns a pointer to an array of if_nameindex structures. Upon failure, if_nameindex() returns NULL and sets errno to one of the following:

Error Code
Description

ENOMEM
Insufficient storage is available to supply the array.

Related information

- “if_freenameindex() — Free the memory allocated by if_nameindex()” on page 832
- “if_indextoname() — Map a network interface index to its corresponding name” on page 832
- “if_nametoindex() — Map a network interface name to its corresponding index”

if_nametoindex() — Map a network interface name to its corresponding index

Standards

Standards / Extensions	C or C++	Dependencies
RFC2553 Single UNIX Specification, Version 3	both	z/OS V1R4

Format

```
#define _OPEN_SYS_SOCK_IPV6
#include <net/if.h>

unsigned int if_nametoindex(const char *ifname);
```

SUSV3:

```
#define _POSIX_C_SOURCE 200112L
#include <net/if.h>

unsigned int if_nametoindex(const char *ifname);
```

General description

The if_nametoindex() function returns the interface index corresponding to the interface name *ifname*.

Returned value

When successful, if_nametoindex() returns the interface index corresponding to the interface name *ifname*. Upon failure, if_nametoindex() returns zero and sets errno to one of the following:

Error Code
Description

EINVAL
Non-valid parameter was specified. The *ifname* parameter was NULL.

ENOMEM
Insufficient storage is available to obtain the information for the interface name.

ENXIO
The specified interface name provided in the *ifname* parameter does not exist.

Related information

- "if_freenameindex() — Free the memory allocated by if_nameindex()" on page 832
- "if_indextoname() — Map a network interface index to its corresponding name" on page 832
- "if_nameindex() — Return all network interface names and indexes" on page 833

ilogb(), ilogbf(), ilogbl() — Integer unbiased exponent

Standards

Standards / Extensions	C or C++	Dependencies
XPG4.2 C99 Single UNIX Specification, Version 3 C++ TR1 C99	both	z/OS V1R7

Format

```
#define _XOPEN_SOURCE_EXTENDED 1
#include <math.h>

int ilogb(double x);
```

C99:

```
#define _ISOC99_SOURCE
#include <math.h>

int ilogbf(float x);
int ilogbl(long double x);
```

C++ TR1 C99:

```
#define _TR1_C99
#include <math.h>

int ilogb(float x);
int ilogb(long double x);
```

General description

The ilogb() functions returns the unbiased exponent of its argument *x* as an integer.

Note: The following table shows the viable formats for these functions. See “IEEE binary floating-point” on page 94 for more information about IEEE Binary Floating-Point.

Function	Hex	IEEE
ilogb	X	X
ilogbf	X	X
ilogbl	X	X

Returned value

If successful, the ilogb() functions return the unbiased exponent of *x* as an integer.

If *x* is 0, the value `FP_ILOGB0` is returned.

if *x* is a NaN, ilogb() will return `FP_ILOGBNAN`

if *x* is infinity, ilogb() will return `INT_MAX`

If the correct value is greater than {INT_MAX}, {INT_MAX} is returned and a domain error occurs.

If the correct value is less than {INT_MIN}, {INT_MIN} is returned and a domain error occurs.

Special behavior for hex: This function will return the unbiased exponent minus 1 (Because hex representation has no hidden bit, this treatment is needed to satisfy the logb() inequality).

Error Code
 Description

EDOM
 The x argument is zero, NaN, or ±inf, or the correct value is not representable as an integer.

Related information

- “math.h” on page 44
- “logb(), logbf(), logbl() — Unbiased exponent” on page 991

ilogbd32(), ilogbd64(), ilogbd128() — Integer unbiased exponent

Standards

Standards / Extensions	C or C++	Dependencies
C/C++ DFP	both	z/OS V1.8

Format

```
#define __STDC_WANT_DEC_FP__
#include <math.h>

int  ilogbd32(_Decimal32 x);
int  ilogbd64(_Decimal64 x);
int  ilogbd128(_Decimal128 x);
int  ilogb(_Decimal32 x);   /* C++ only */
int  ilogb(_Decimal64 x);   /* C++ only */
int  ilogb(_Decimal128 x);  /* C++ only */
```

General description

Returns the unbiased exponent of its argument *x* as an integer. For typical numbers, the value returned is the logarithm of |*x*| rounded down (toward -INF) to the nearest integer value.

Notes:

1. To use IEEE decimal floating-point, the hardware must have the Decimal Floating-Point Facility installed.
2. These functions work in IEEE decimal floating-point format. See "IEEE Decimal Floating-Point" for more information.

Returned value

If successful, these functions return the unbiased exponent of *x* as an integer.

If *x* is equal to 0.0, ilogb() will return _FP_DEC_ILOGB0 (= -INT_MAX).

If *x* is a NaN or infinity, ilogb() will return INT_MAX.

Example

```
/* CELEBI11

   This example illustrates the ilogbd128() function.

*/

#define  __STDC_WANT_DEC_FP__
#include <math.h>
#include <stdio.h>

int main(void)
{
   _Decimal128 x = -12345.678901DL;
   int y;

   y = ilogbd128(x);

   printf("The result of ilogbd128(%DDf) is %d\n", x, y);
}
```

Related information

- “math.h” on page 44
- “frexpd32(), frexpd64(), frexpd128() — Extract mantissa and exponent of the decimal floating-point value” on page 624
- “ilogb(), ilogbf(), ilogbl() — Integer unbiased exponent” on page 835
- “logbd32(), logbd64(), logbd128() — Unbiased exponent” on page 993

imaxabs() — Absolute value for intmax_t

Standards

Standards / Extensions	C or C++	Dependencies
C99 Single UNIX Specification, Version 3 C++ TR1 C99	both	z/OS V1R7

Format

```
#define _ISOC99_SOURCE
#include <inttypes.h>

intmax_t imaxabs(intmax_t j);
```

Compile requirement: Function imaxabs() requires long long to be available.

General description

The imaxabs() function computes the absolute value of *j*. When the input value is INTMAX_MIN, the value is undefined. The imaxabs() function is similar to llabs() and labs(). The only difference being that the return value and the argument passed in are of type intmax_t.

Returned value

The imaxabs function returns the absolute value of *j*.

Example

```
#define _ISOC99_SOURCE
#include <inttypes.h>
#include <stdio.h>
int main(void)
{
  intmax_t a = -1234;

  intmax_t b = imaxabs(a);


  printf("%jd \n", b );
}

Output:

1234
```

Related information

- inttypes.h
- labs()
- llabs()

imaxdiv() — Quotient and remainder for intmax_t

Standards

Standards / Extensions	C or C++	Dependencies
C99 Single UNIX Specification, Version 3 C++ TR1 C99	both	z/OS V1R7

Format

```
#define _ISOC99_SOURCE
#include <inttypes.h>

imaxdiv_t imaxdiv(intmax_t numer, intmax_t denom);
```

Compile requirement: Function imaxdiv() requires long long to be available.

General description

The imaxdiv() function computes numer / denom and numer % denom in a single operation. The imaxdiv function is similar to lldiv() and ldiv(). The only difference being that the return value is of type imaxdiv_t and those being passed in are of type intmax_t.

Returned value

imaxdiv() returns a structure of type imaxdiv_t comprising both the quotient and the remainder. If either part of the result cannot be represented, the behavior is undefined. if the denominator is zero, a divide by zero exception is raised.

Example

```
#define _ISOC99_SOURCE
#include <inttypes.h>
#include <stdio.h>

int main(void)
{
 intmax_t num = 45;
 intmax_t den = 7;
 imaxdiv_t res;
       printf("Original numerator: %jd and denominator: %jd "
             ,num,den);
       res = imaxdiv(num,den);
       printf("Quotient: %jd  Remainder: %jd\n"
             , res.quot,res.rem);
}

Output

Original numerator: 45 and denominator: -7 Quotient: -6  Remainder: 3
```

Related information

- inttypes.h
- ldiv()
- lldiv()

ImportWorkUnit() — WLM import service

Standards

Standards / Extensions	C or C++	Dependencies
z/OS UNIX	both	OS/390 V2R9

Format

```
#include <sys/__wlm.h>

int ImportWorkUnit(wlmxtok_t *exporttoken,
                   wlmetok_t *enclavetoken,
                   unsigned int *conntoken);
```

General description

Imports an enclave that has been previously exported using the ExportWorkUnit() function. The caller must invoke UnDoImportWorkUnit() when it no longer needs access to the enclave.

The ImportWorkUnit() function uses the following parameters:

**enclavetoken*
: Points to a work unit export token that was returned from a call to ExportWorkUnit().

**exporttoken*
: Points to a data field of type wlmetok_t where the ImportWorkUnit() function is to return the WLM work unit enclave token.

**conntoken*
: Specifies the connect token that represents the WLM connection.

Returned value

If successful, ImportWorkUnit() returns 0.

If unsuccessful, ImportWorkUnit() returns -1 and sets errno to one of the following values:

Error Code
Description

EFAULT
An argument of this function contained an address that was not accessible to the caller.

EINVAL
An argument of this function contained a value that is not correct.

EMVSSAF2ERR
An error occurred in the security product.

EMVSWLMERROR
A WLM service failed. Use __errno2() to obtain the WLM service reason code for the failure.

EPERM
The calling thread's address space is not permitted to the BPX.WLMSERVER Facility class. The caller's address space must be

permitted to the BPX.WLMSERVER Facility class, if the BPX.WLMSERVER class is defined. If BPX.WLMSERVER is not defined, the calling process is not defined as a superuser (UID=0).

Related information

- "sys/__wlm.h" on page 77
- "ExportWorkUnit() — WLM export service" on page 452
- "UnDoExportWorkUnit() — WLM undo export service" on page 1955
- "UnDoImportWorkUnit() — WLM undo import service" on page 1956
- For more information, see *z/OS MVS Programming: Workload Management Services, SC34-2663*

index() — Search for character

Standards

Standards / Extensions	C or C++	Dependencies
XPG4.2 Single UNIX Specification, Version 3	both	

Format

```
#define _XOPEN_SOURCE_EXTENDED 1
#include <strings.h>

char *index(const char *string, int c);
```

General description

The index() function locates the first occurrence of *c* (converted to an unsigned char) in the string pointed to by *string*. The character *c* can be the NULL character (\0); the ending NULL is included in the search.

The string argument to the function must contain a NULL character (\0) marking the end of the string.

The index() function is identical to "strchr() — Search for character" on page 1729.

Note: The index() function has been moved to the Legacy Option group in Single UNIX Specification, Version 3 and may be withdrawn in a future version. The strchr() function is preferred for portability.

Returned value

If successful, index() returns a pointer to the first occurrence of *c* (converted to an unsigned character) in the string pointed to by *string*.

If unsuccessful because *c* was not found, index() returns a NULL pointer.

There are no errno values defined.

Related information

- "strings.h" on page 72
- "memchr() — Search buffer" on page 1061
- "rindex() — Search for character" on page 1449
- "strchr() — Search for character" on page 1729

- "strrchr() — Find last occurrence of character in string" on page 1759
- "strspn() — Search string" on page 1761
- "strstr() — Locate substring" on page 1762

inet6_is_srcaddr() - Socket address verification

Standards

Standards / Extensions	C or C++	Dependencies
RFC 5014	both	

Format

```
#define _OPEN_SYS_SOCK_IPV6
#include <netinet/in.h>

short inet6_is_srcaddr(struct sockaddr_in6 *srcaddr, uint32_t flags);
```

General description

The inet6_is_srcaddr() function validates source address selection preference flags against the given socket address.

Argument
Description

srcaddr
A non-NULL pointer to a sockaddr_in6 structure initialized as follows:

- Clear the entire structure for sizeof(struct sockaddr_in6).
- *sin6_family* must be set to AF_INET6.
- Set *sin6_len* to the correct length for AF_INET6.
- Set *sin6_addr* to a 128-bit IPv6 source address to validate against the flags.
- The *sin6_scope_id* must be set if the address is link-local.

flags The source preference flags which can be set to any of the following:

- IPV6_PREFER_SRC_HOME - prefer home address as source
- IPV6_PREFER_SRC_COA - prefer care-of address as source
- IPV6_PREFER_SRC_TMP - prefer temporary address as source
- IPV6_PREFER_SRC_PUBLIC - prefer public address as source
- IPV6_PREFER_SRC_CGA - prefer CGA address as source
- IPV6_PREFER_SRC_NONCGA - prefer a non-CGA address as source

These flags can be combined into a flag set to express complex address preferences, but some can result in a contradictory flag set.

For example, the following flags are mutually exclusive:

- IPV6_PREFER_SRC_HOME and IPV6_PREFER_SRC_COA
- IPV6_PREFER_SRC_TMP and IPV6_PREFER_SRC_PUBLIC
- IPV6_PREFER_SRC_CGA and IPV6_PREFER_SRC_NONCGA

Returned value

When the IPv6 address corresponds to a valid address in the node and satisfies the given preference flags, inet6_is_srcaddr() returns 1.

If the input address matches an address in the node, but does not satisfy the preference flags indicated, the function returns 0.

If unsuccessful, inet6_is_srcaddr() returns -1 and sets errno to the following:

Error Code
Description

EADDRNOTAVAIL
The address provided does not match a home address in the node.

EAFNOSUPPORT
The address provided does not have a family of AF_INET6.

EAGAIN
A TCP/IP stack is not active to process the request.

EINVAL
Undefined flags were used, or a link-local IPv6 address was used with a zero scopeid, or a global address was used with a non-zero scopeid.

Related information

- "netinet/in.h" on page 53
- "bind() — Bind a name to a socket" on page 210
- "bind2addrsel() - Bind with source address selection" on page 216
- "getaddrinfo() — Get address information" on page 684
- "getsockopt() — Get the options associated with a socket" on page 777
- "setsockopt() — Set options associated with a socket" on page 1578

inet6_opt_append() — Add an option with length "len" and alignment "align"

Standards

Standards / Extensions	C or C++	Dependencies
RFC3542	both	z/OS V1R7

Format

```
#define _OPEN_SYS_SOCK_IPV6
#include <netinet/in.h>

int inet6_opt_append(void *extbuf, socklen_t extlen, int offset,
                     uint8_t type, socklen_t len, uint8_t align,
                     void **databufp);
```

General description

inet6_opt_append() returns the updated total length after adding an option with length *len* and alignment *align*. If *extbuf* is not NULL, it inserts any necessary padding and sets the type and length fields. A pointer to the location for the option content in *databufp* is then returned.

offset should be the length returned by inet6_opt_init() or the previous inet6_opt_append(). *type* is the 8-bit option type and *len* is the length of the option data (excluding the option type and option length fields).

Returned value

If successful, inet6_opt_append() returns the updated total length of the extension header.

Upon failure, returns -1 and errno is set to one of the following:

EINVAL If one of the following is true:
- *extbuf* is NULL and *extlen* is non-zero;
- *extbuf* is non-NULL and *extlen* is not a positive multiple of 8;
- *offset* is less than the size of the empty extension header;
- *type* is not valid (specifies one of the PAD options);
- *len* is less than 0 or greater than 255;
- *align* is not 1, 2, 4, or 8;
- *align* is greater than *len*;
- new updated total length would exceed *extlen* (extbuf is non-NULL);
- *databufp* is NULL (extbuf is non-NULL).

Usage notes

1. The option, *type*, must have a value from 2 to 255 (0 and 1 are reserved for the Pad1 and PadN options).
2. The option data length must have a value between 0 and 255, including the values 0 and 255. It is the length of the option data that follows.
3. The *align* parameter must have a value of 1, 2, 4, or 8 and can not exceed the value of *len*.
4. Once inet6_opt_append() has been called, the application can use *databufp* directly or use inet6_opt_set_val() to specify the content of the option.

Related information

- “netinet/in.h” on page 53
- “inet6_opt_find() — Search for an option specified by the caller”
- “inet6_opt_finish() — Return the updated total length of extension header” on page 846
- “inet6_opt_get_val() — Extract data items in the data portion of the option” on page 847
- “inet6_opt_init() — Return the number of bytes for empty extension header” on page 848
- “inet6_opt_next() — Parse received option headers returning the next option” on page 849
- “inet6_opt_set_val() — Insert data items into the data portion of the option” on page 850

inet6_opt_find() — Search for an option specified by the caller

Standards

Standards / Extensions	C or C++	Dependencies
RFC3542	both	z/OS V1R7

Format

```
#define _OPEN_SYS_SOCK_IPV6
#include <netinet/in.h>

int inet6_opt_find(void *extbuf, socklen_t extlen, int offset,
                   uint8_t type, socklen_t *lenp, void **databufp);
```

General description

inet6_opt_find() is similar to inet6_opt_next(), except it lets the caller specify the option type to be searched for.

Returned value

If successful, inet6_opt_find() returns the updated "previous" total length computed by advancing past the option that was returned and past any options that did not match the type.

Upon failure, returns -1 and errno is set to one of the following:

EINVAL If one of the following is true:
- *extbuf* is NULL;
- *extlen* is not a positive multiple of 8;
- *offset* is less than 0 or greater than or equal to *extlen*;
- *lenp* or *databufp* is NULL;
- the option was not located;
- the extension header is malformed.

Usage notes

The returned "previous" length can be passed to subsequent calls of inet6_opt_find() for finding the next occurrence of the same option type.

Related information

- “netinet/in.h” on page 53
- “inet6_opt_append() — Add an option with length "len" and alignment "align"” on page 843
- “inet6_opt_finish() — Return the updated total length of extension header” on page 846
- “inet6_opt_get_val() — Extract data items in the data portion of the option” on page 847
- “inet6_opt_init() — Return the number of bytes for empty extension header” on page 848
- “inet6_opt_next() — Parse received option headers returning the next option” on page 849
- “inet6_opt_set_val() — Insert data items into the data portion of the option” on page 850

inet6_opt_finish() — Return the updated total length of extension header

Standards

Standards / Extensions	C or C++	Dependencies
RFC3542	both	z/OS V1R7

Format

```
#define _OPEN_SYS_SOCK_IPV6
#include <netinet/in.h>

int inet6_opt_finish(void *extbuf, socklen_t extlen, int offset);
```

General description

inet6_opt_finish() returns the updated total length taking into account the final padding of the extension header to make it a multiple of 8 bytes. If *extbuf* is not NULL the function also initializes the option by inserting a Pad1 or PadN option of the proper length.

Returned value

If successful, inet6_opt_finish() returns the total length of the extension header including the final padding.

Upon failure, returns -1 and errno is set to one of the following:

EINVAL If one of the following is true:
- *extbuf* is NULL and *extlen* is non-zero;
- *extbuf* is non-NULL and *extlen* is not a positive multiple of 8;
- *extbuf* is non-NULL and offset is greater than *extlen;*
- *offset* is less than the size of the empty extension header.

Usage notes

offset should be the length returned by inet6_opt_init() or inet6_opt_append().

Related information

- “netinet/in.h” on page 53
- “inet6_opt_append() — Add an option with length "len" and alignment "align"” on page 843
- “inet6_opt_find() — Search for an option specified by the caller” on page 844
- “inet6_opt_get_val() — Extract data items in the data portion of the option” on page 847
- “inet6_opt_init() — Return the number of bytes for empty extension header” on page 848
- “inet6_opt_next() — Parse received option headers returning the next option” on page 849
- “inet6_opt_set_val() — Insert data items into the data portion of the option” on page 850

inet6_opt_get_val() — Extract data items in the data portion of the option

Standards

Standards / Extensions	C or C++	Dependencies
RFC3542	both	z/OS V1R7

Format

```
#define _OPEN_SYS_SOCK_IPV6
#include <netinet/in.h>

int inet6_opt_get_val(void *databuf, int offset,
                      void *val, socklen_t vallen);
```

General description

inet6_opt_get_val() extracts data items of various sizes in the data portion of the option.

Returned value

If successful, inet6_opt_get_val() returns the offset for the next field (*offset* + *vallen*) that can be used when extracting option content with multiple fields.

Upon failure, returns -1 and sets errno to one of the following:

EINVAL If one of the following is true:
- *databuf* is NULL;
- *val* is null;
- *offset* is less than 0;
- *offset* + *vallen* is greater than the option length.

Usage notes

1. *databuf* should be a pointer returned by inet6_opt_next() or inet6_opt_find().
2. *val* should point to the destination for the extracted data.
3. *offset* specifies from where in the data portion of the option the value should be extracted; the first byte after the option type and length is accessed by specifying an *offset* of zero.

Related information

- “netinet/in.h” on page 53
- “inet6_opt_append() — Add an option with length "len" and alignment "align"” on page 843
- “inet6_opt_find() — Search for an option specified by the caller” on page 844
- “inet6_opt_finish() — Return the updated total length of extension header” on page 846
- “inet6_opt_init() — Return the number of bytes for empty extension header” on page 848
- “inet6_opt_next() — Parse received option headers returning the next option” on page 849

- “inet6_opt_set_val() — Insert data items into the data portion of the option” on page 850

inet6_opt_init() — Return the number of bytes for empty extension header

Standards

Standards / Extensions	C or C++	Dependencies
RFC3542	both	z/OS V1R7

Format

```
#define _OPEN_SYS_SOCK_IPV6
#include <netinet/in.h>

int inet6_opt_init(void *extbuf, socklen_t extlen);
```

General description

inet6_opt_init() returns the number of bytes needed for the empty extension header. If *extbuf* is not NULL, the extension header is initialized to have the correct length field and the *extlen* value must be a positive, non-zero, multiple of 8, or the function will fail.

Returned value

If successful, inet6_opt_init() returns the number of bytes needed for the empty extension header.

Upon failure, returns -1 and errno is set to one of the following:

EINVAL If one of the following is true:
- *extbuf* is NULL and *extlen* is non-zero;
- *extbuf* is non-NULL and *extlen* is not a positive multiple of 8.

Related information

- “netinet/in.h” on page 53
- “inet6_opt_append() — Add an option with length "len" and alignment "align"” on page 843
- “inet6_opt_find() — Search for an option specified by the caller” on page 844
- “inet6_opt_finish() — Return the updated total length of extension header” on page 846
- “inet6_opt_get_val() — Extract data items in the data portion of the option” on page 847
- “inet6_opt_next() — Parse received option headers returning the next option” on page 849
- “inet6_opt_set_val() — Insert data items into the data portion of the option” on page 850

inet6_opt_next() — Parse received option headers returning the next option

Standards

Standards / Extensions	C or C++	Dependencies
RFC3542	both	z/OS V1R7

Format

```
#define _OPEN_SYS_SOCK_IPV6
#include <netinet/in.h>

int inet6_opt_next(void *extbuf, socklen_t extlen, int offset,
                   uint8_t *typep, socklen_t *lenp, void **databufp);
```

General description

inet6_opt_next() parses received option extension headers and returns the next option.

Returned value

If successful, inet6_opt_next() returns the updated "previous" length computed by advancing past the option that was returned. This returned "previous" length can then be passed to subsequent calls to inet6_opt_next(). This function does not return any PAD1 or PADN options.

Upon failure, returns -1 and errno is set to one of the following:

EINVAL If one of the following is true:

- *extbuf* is NULL;
- *extlen* is not a positive multiple of 8;
- *offset* is less than 0 or greater than or equal to *extlen*;
- *typep*, *lenp*, or *databufp* is NULL;
- there are no more options;
- the extension header is malformed.

Usage notes

1. *extbuf* and *extlen* specifies the extension header.
2. *offset* should either be zero (for the first option) or the length returned by a previous call to inet6_opt_next() or inet6_opt_find(). It specifies the position to continue scanning the extension buffer. The next option is returned by updating *typep*, *lenp*, and *databufp*.
3. *typep* points to the option type field.
4. *lenp* stores the length of the option data (excluding the option type and option length fields).
5. *databufp* points to the data field of the of the option.

Related information

- “netinet/in.h” on page 53
- “inet6_opt_append() — Add an option with length "len" and alignment "align"” on page 843

- "inet6_opt_find() — Search for an option specified by the caller" on page 844
- "inet6_opt_finish() — Return the updated total length of extension header" on page 846
- "inet6_opt_get_val() — Extract data items in the data portion of the option" on page 847
- "inet6_opt_init() — Return the number of bytes for empty extension header" on page 848
- "inet6_opt_set_val() — Insert data items into the data portion of the option"

inet6_opt_set_val() — Insert data items into the data portion of the option

Standards

Standards / Extensions	C or C++	Dependencies
RFC3542	both	z/OS V1R7

Format

```
#define _OPEN_SYS_SOCK_IPV6
#include <netinet/in.h>

int inet6_opt_set_val(void *databuf, int offset,
                      void *val, socklen_t vallen);
```

General description

inet6_opt_set_val() inserts items of various sizes in the data portion of the option.

Returned value

If successful, inet6_opt_set_val() returns the *offset* for the next field (*offset* + *vallen*) that can be used when composing option content with multiple fields.

Upon failure, returns -1 and errno is set to one of the following:

EINVAL If one of the following is true:
- *databuf* is NULL;
- *val* is NULL;
- *offset* is less than 0;
- *offset* + *vallen* is greater than the option length.

Usage notes

1. *databuf* should be a pointer returned by inet6_opt_append().
2. *val* should point to the data to be inserted.
3. *offset* specifies where in the data portion of the option the value should be inserted; the first byte after the option type and length is accessed by specifying an *offset* of 0.

Related information

- "netinet/in.h" on page 53

- “inet6_opt_append() — Add an option with length "len" and alignment "align"” on page 843
- “inet6_opt_find() — Search for an option specified by the caller” on page 844
- “inet6_opt_finish() — Return the updated total length of extension header” on page 846
- “inet6_opt_get_val() — Extract data items in the data portion of the option” on page 847
- “inet6_opt_init() — Return the number of bytes for empty extension header” on page 848
- “inet6_opt_next() — Parse received option headers returning the next option” on page 849

inet6_rth_add() — Add an IPv6 address to end of the routing header

Standards

Standards / Extensions	C or C++	Dependencies
RFC3542	both	z/OS V1R7

Format

```
#define _OPEN_SYS_SOCK_IPV6
#include <netinet/in.h>

int inet6_rth_add(void *bp, const struct in6_addr *addr);
```

General description

inet6_rth_add() adds the IPv6 address pointed to by *addr* to the end of the routing header that is being constructed.

Returned value

If successful, inet6_rth_add() returns 0 and the segleft member of the routing header is updated to account for the new address.

Upon failure, returns -1 and errno is set to one of the following:

EINVAL If one of the following is true:
- *bp* is NULL;
- the routing header indicates an unsupported header type;
- the routing header contains a non-valid number of segments for the type;
- there is not enough room to add the address.

Related information

- “netinet/in.h” on page 53
- “inet6_rth_getaddr() — Return pointer to the IPv6 address specified” on page 852
- “inet6_rth_init() — Initialize an IPv6 routing header buffer” on page 852
- “inet6_rth_reverse() — Reverse the order of the addresses” on page 853
- “inet6_rth_segments() — Return number of segments contained in header” on page 854
- “inet6_rth_space() — Return number of bytes for a routing header” on page 855

inet6_rth_getaddr() — Return pointer to the IPv6 address specified

Standards

Standards / Extensions	C or C++	Dependencies
RFC3542	both	z/OS V1R7

Format

```
#define _OPEN_SYS_SOCK_IPV6
#include <netinet/in.h>

struct in6_addr *inet6_rth_getaddr(const void *bp, int index);
```

General description

inet6_rth_getaddr() returns a pointer to the IPv6 address specified by *index* in the routing header described by *bp*.

Returned value

If successful, inet6_rth_getaddr() returns a pointer to the IPv6 address.

Upon failure, returns NULL and errno is set to one of the following:

EINVAL If one of the following is true:
- *bp* is NULL;
- the routing header indicates an unsupported header type;
- the routing header contains a non-valid number of segments;
- *index* is less than 0 or greater than or equal to the number of segments.

Usage notes

1. To obtain the number of segments in the routing header, a call to inet6_rth_segments() should be made first.
2. *index* must have a value between 0 and one less than the value returned by inet6_rth_segments().

Related information

- “netinet/in.h” on page 53
- “inet6_rth_add() — Add an IPv6 address to end of the routing header” on page 851
- “inet6_rth_init() — Initialize an IPv6 routing header buffer”
- “inet6_rth_reverse() — Reverse the order of the addresses” on page 853
- “inet6_rth_segments() — Return number of segments contained in header” on page 854
- “inet6_rth_space() — Return number of bytes for a routing header” on page 855

inet6_rth_init() — Initialize an IPv6 routing header buffer

Standards

Standards / Extensions	C or C++	Dependencies
RFC3542	both	z/OS V1R7

Format

```
#define _OPEN_SYS_SOCK_IPV6
#include <netinet/in.h>

void *inet6_rth_init(void *bp, socklen_t bp_len,
                     int type, int segments);
```

General description

inet6_rth_init() initializes the buffer pointed to by *bp* to contain a routing header of the specified *type* and sets ip6r_len based on the *segments* parameter.

Returned value

When successful, inet6_rth_init() returns the pointer to the buffer, bp. This is then used as the first argument to the inet6_rth_add() function.

Upon failure, returns NULL and errno is set to one of the following:

EINVAL If one of the following is true::
- *bp* is NULL;
- *type* indicates an unsupported header type;
- *segments* is not valid for the *type*;
- the buffer is not large enough, *bp_len* is too small.

Usage notes

1. The caller must allocate the buffer; its size can be determined by calling inet6_rth_space().
2. Any cmsghdr fields must be initialized when the application uses ancillary data.

Related information

- “netinet/in.h” on page 53
- “inet6_rth_add() — Add an IPv6 address to end of the routing header” on page 851
- “inet6_rth_getaddr() — Return pointer to the IPv6 address specified” on page 852
- “inet6_rth_reverse() — Reverse the order of the addresses”
- “inet6_rth_segments() — Return number of segments contained in header” on page 854
- “inet6_rth_space() — Return number of bytes for a routing header” on page 855

inet6_rth_reverse() — Reverse the order of the addresses

Standards

Standards / Extensions	C or C++	Dependencies
RFC3542	both	z/OS V1R7

Format

```
#define _OPEN_SYS_SOCK_IPV6
#include <netinet/in.h>

int inet6_rth_reverse(const void *in, void *out);
```

General description

inet6_rth_reverse() takes a routing header, pointed to by *in*, and writes a new routing header that sends datagrams along the reverse of that route. It reverses the order of the addresses and sets the segleft member in the new routing header to the number of segments required to send the datagram back to where it originated. Both arguments are allowed to point to the same buffer (the reversal can occur in place).

Returned value

If successful, inet6_rth_reverse() returns 0.

Upon failure, returns -1 and errno is set to one of the following:

EINVAL If one of the following is true:
- *in* is NULL or *out* is NULL;
- the input routing header indicates an unsupported header type;
- the input routing header contains a non-valid number of segments;
- *in* and *out* overlap, but *in* and *out* are not the same buffer.

Related information

- “netinet/in.h” on page 53
- “inet6_rth_add() — Add an IPv6 address to end of the routing header” on page 851
- “inet6_rth_getaddr() — Return pointer to the IPv6 address specified” on page 852
- “inet6_rth_init() — Initialize an IPv6 routing header buffer” on page 852
- “inet6_rth_segments() — Return number of segments contained in header”
- “inet6_rth_space() — Return number of bytes for a routing header” on page 855

inet6_rth_segments() — Return number of segments contained in header

Standards

Standards / Extensions	C or C++	Dependencies
RFC3542	both	z/OS V1R7

Format

```
#define _OPEN_SYS_SOCK_IPV6
#include <netinet/in.h>

int inet6_rth_segments(const void *bp);
```

General description

inet6_rth_segments() returns the number of segments (addresses) contained in the routing header described by *bp*.

Returned value

If successful, inet6_rth_segments() returns the number of segments or 0, if there are none in the header.

Upon failure, returns -1 and errno is set to one of the following:

EINVAL If one of the following is true:
- *bp* is NULL;
- the routing header indicates an unsupported header type;
- the routing header contains a non-valid number of segments.

Related information

- "netinet/in.h" on page 53
- "inet6_rth_add() — Add an IPv6 address to end of the routing header" on page 851
- "inet6_rth_getaddr() — Return pointer to the IPv6 address specified" on page 852
- "inet6_rth_init() — Initialize an IPv6 routing header buffer" on page 852
- "inet6_rth_reverse() — Reverse the order of the addresses" on page 853
- "inet6_rth_space() — Return number of bytes for a routing header"

inet6_rth_space() — Return number of bytes for a routing header

Standards

Standards / Extensions	C or C++	Dependencies
RFC3542	both	z/OS V1R7

Format

```
#define _OPEN_SYS_SOCK_IPV6
#include <netinet/in.h>

socklen_t inet6_rth_space(int type, int segments);
```

General description

inet6_rth_space() calculates the number of bytes required to hold a routing header for the specified *type* containing the specified number of *segments* (addresses).

Returned value

If successful, inet6_rth_space() returns the number of bytes, space required, for the routing header.

Upon failure, returns 0 and errno is set to one of the following:

EINVAL If one of the following is true:

- *type* indicates an unsupported header type;
- *segments* is not valid for the *type*.

Usage notes

1. This function returns the size but does not allocate the space required for the ancillary data. This allows an application to allocate a larger buffer, if other ancillary data objects are desired, because all the ancillary data objects must be specified to sendmsg() as a single msg_control buffer.
2. For an IPv6 Type 0 routing header, the number of *segments* must be between 0 and 127, inclusive. When the application uses ancillary data it must pass the returned length to CMSG_SPACE() to determine how much memory is needed for the ancillary data object (including the cmsghdr structure).

Related information

- “netinet/in.h” on page 53
- “inet6_rth_add() — Add an IPv6 address to end of the routing header” on page 851
- “inet6_rth_getaddr() — Return pointer to the IPv6 address specified” on page 852
- “inet6_rth_init() — Initialize an IPv6 routing header buffer” on page 852
- “inet6_rth_reverse() — Reverse the order of the addresses” on page 853
- “inet6_rth_segments() — Return number of segments contained in header” on page 854

inet_addr() — Translate an Internet address into network byte order

Standards

Standards / Extensions	C or C++	Dependencies
XPG4.2 Single UNIX Specification, Version 3	both	

Format

X/Open:

```
#define _XOPEN_SOURCE_EXTENDED 1
#include <arpa/inet.h>

in_addr_t inet_addr(const char *cp);
```

Berkeley sockets:

```
#define _OE_SOCKETS
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>

unsigned long inet_addr(char *cp);
```

General description

The inet_addr() function interprets character strings representing host addresses expressed in standard dotted-decimal notation and returns host addresses suitable for use as an Internet address.

Parameter
Description

cp A character string in standard dotted-decimal (**.**) notation.

Values specified in standard dotted-decimal notation take one of the following forms:

```
a.b.c.d
a.b.c
a.b
a
```

When a 4-part address is specified, each part is interpreted as a byte of data and assigned, from left to right, to one of the 4 bytes of an Internet address.

When a three-part address is specified, the last part is interpreted as a 16-bit quantity and placed in the two rightmost bytes of the network address. This makes the three-part address format convenient for specifying class-B network addresses as **128.net.host**.

When a two-part address is specified, the last part is interpreted as a 24-bit quantity and placed in the three rightmost bytes of the network address. This makes the two-part address format convenient for specifying class-A network addresses as **net.host**.

When a one-part address is specified, the value is stored directly in the network address space without any rearrangement of its bytes.

Numbers supplied as address parts in standard dotted-decimal notation can be decimal, hexadecimal, or octal. Numbers are interpreted in C language syntax. A leading 0x implies hexadecimal; a leading 0 implies octal. A number without a leading 0 implies decimal.

The address must be terminated with a null or other white space character to be valid. Any or all of the parts of the address may be empty strings. Each empty part resolves to the value 0. Input character strings that begin with a null or other white space character is treated as an empty address and will result in the value INADDR_ANY being returned.

Special behavior for C++: To use this function with C++, you must use the _XOPEN_SOURCE_EXTENDED 1 feature test macro.

Notes:
1. To provide an ASCII input/output format for applications using this function, define the feature test macro __LIBASCII as described in topic "__LIBASCII" on page 7.
2. The inet_addr() function has a dependency on the level of the Enhanced ASCII Extensions. See "Enhanced ASCII support" on page 2113 for details.

Returned value

If successful, inet_addr() returns the Internet address in network byte order.

If the input character string is not in the correct format, the value INADDR_NONE is returned and errno is set to EINVAL.

Related information

- "arpa/inet.h" on page 17
- "netinet/in.h" on page 53
- "sys/socket.h" on page 74
- "sys/types.h" on page 75
- "inet_makeaddr() — Create an Internet host address" on page 859
- "inet_netof() — Get the network number from the Internet host sddress" on page 860
- "inet_network() — Get the network number from the decimal host address" on page 861
- "inet_ntoa() — Get the decimal Internet host address" on page 862
- "inet_ntop() — Convert Internet address format from binary to text" on page 863
- "inet_pton() — Convert Internet address format from text to binary" on page 865

inet_lnaof() — Translate a local network address into host byte order

Standards

Standards / Extensions	C or C++	Dependencies
XPG4.2	both	

Format

X/Open:

```
#define _XOPEN_SOURCE_EXTENDED 1
#include <arpa/inet.h>

in_addr_t inet_lnaof(struct in_addr in);
```

Berkeley sockets:

```
#define _OE_SOCKETS
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>

unsigned long inet_lnaof(struct in_addr in);
```

General description

The inet_lnaof() function breaks apart the Internet host address and returns the local network address portion.

Parameter
Description

in The host Internet address.

Special behavior for C++: To use this function with C++, you must use the _XOPEN_SOURCE_EXTENDED 1 feature test macro.

Returned value

The local network address is returned in host byte order.

Related information

- “arpa/inet.h” on page 17
- “netinet/in.h” on page 53
- “sys/socket.h” on page 74
- “sys/types.h” on page 75
- “inet_makeaddr() — Create an Internet host address”
- “inet_netof() — Get the network number from the Internet host sddress” on page 860
- “inet_network() — Get the network number from the decimal host address” on page 861
- “inet_ntoa() — Get the decimal Internet host address” on page 862

inet_makeaddr() — Create an Internet host address

Standards

Standards / Extensions	C or C++	Dependencies
XPG4.2	both	

Format

X/Open:

```
#define _XOPEN_SOURCE_EXTENDED 1
#include <arpa/inet.h>

struct in_addr inet_makeaddr(in_addr_t net, in_addr_t lna);
```

Berkeley sockets:

```
#define _OE_SOCKETS
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>

struct in_addr inet_makeaddr(unsigned long net, unsigned long lna);
```

General description

The inet_makeaddr() function takes a network number and a local network address and constructs an Internet address.

Parameter
Description

net The network number.

lna The local network address.

Special behavior for C++: To use this function with C++, you must use the _XOPEN_SOURCE_EXTENDED 1 feature test macro.

Returned value

The Internet address is returned in network byte order.

Related information

- “arpa/inet.h” on page 17
- “netinet/in.h” on page 53
- “sys/socket.h” on page 74

- “sys/types.h” on page 75
- “inet_lnaof() — Translate a local network address into host byte order” on page 858
- “inet_netof() — Get the network number from the Internet host sddress”
- “inet_network() — Get the network number from the decimal host address” on page 861
- “inet_ntoa() — Get the decimal Internet host address” on page 862
- “inet_ntop() — Convert Internet address format from binary to text” on page 863
- “inet_pton() — Convert Internet address format from text to binary” on page 865

inet_netof() — Get the network number from the Internet host sddress

Standards

Standards / Extensions	C or C++	Dependencies
XPG4.2	both	

Format

X/Open:

```
#define _XOPEN_SOURCE_EXTENDED 1
#include <arpa/inet.h>

in_addr_t inet_netof(struct in_addr in);
```

Berkeley sockets:

```
#define _OE_SOCKETS
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>

unsigned long inet_netof(struct addr_in in);
```

General description

The inet_netof() function breaks apart the Internet host address and returns the network number portion.

Parameter
Description

in The Internet address in network byte order.

Special behavior for C++: To use this function with C++, you must use the _XOPEN_SOURCE_EXTENDED 1 feature test macro.

Returned value

The network number is returned in host byte order.

Related information

- “arpa/inet.h” on page 17
- “netinet/in.h” on page 53
- “sys/socket.h” on page 74

- “sys/types.h” on page 75
- “inet_lnaof() — Translate a local network address into host byte order” on page 858
- “inet_makeaddr() — Create an Internet host address” on page 859
- “inet_ntoa() — Get the decimal Internet host address” on page 862
- “inet_ntop() — Convert Internet address format from binary to text” on page 863
- “inet_pton() — Convert Internet address format from text to binary” on page 865

inet_network() — Get the network number from the decimal host address

Standards

Standards / Extensions	C or C++	Dependencies
XPG4.2	both	

Format

X/Open:

```
#define _XOPEN_SOURCE_EXTENDED 1
#include <arpa/inet.h>

in_addr_t inet_network(const char *cp);
```

Berkeley sockets:

```
#define _OE_SOCKETS
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>

unsigned long inet_network(char cp);
```

General description

The inet_network() function interprets character strings representing addresses expressed in standard dotted-decimal notation and returns numbers suitable for use as a network number.

Parameter
Description

cp A character string in standard, dotted-decimal (.) notation.

Note: The input value is handled as an octal number when there are 3 integers within the dotted-decimal notation. For example, the input value of inet_network(“40.001.016.000”) validly returns X'28010e00' (40.1.14.0) since the 016 is treated as an octal number.

Special behavior for C++: To use this function with C++, you must use the _XOPEN_SOURCE_EXTENDED 1 feature test macro.

Note: The inet_network() function has a dependency on the level of the Enhanced ASCII Extensions. See “Enhanced ASCII support” on page 2113 for details.

Returned value

The network number is returned in host byte order.

Related information

- “arpa/inet.h” on page 17
- “netinet/in.h” on page 53
- “sys/socket.h” on page 74
- “sys/types.h” on page 75
- “inet_lnaof() — Translate a local network address into host byte order” on page 858
- “inet_makeaddr() — Create an Internet host address” on page 859
- “inet_ntoa() — Get the decimal Internet host address”
- “inet_ntop() — Convert Internet address format from binary to text” on page 863
- “inet_pton() — Convert Internet address format from text to binary” on page 865

inet_ntoa() — Get the decimal Internet host address

Standards

Standards / Extensions	C or C++	Dependencies
XPG4.2 Single UNIX Specification, Version 3	both	

Format

X/Open:

```
#define _XOPEN_SOURCE_EXTENDED 1
#include <arpa/inet.h>

char *inet_ntoa(struct in_addr in);
```

Berkeley sockets:

```
#define _OE_SOCKETS
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>

char *inet_ntoa(struct in_addr in);
```

General description

The inet_ntoa() function returns a pointer to a string expressed in the dotted-decimal notation. inet_ntoa() accepts an Internet address expressed as a 32-bit quantity in network byte order and returns a string expressed in dotted-decimal notation.

Parameter
    Description

in The host Internet address.

To provide an ASCII input/output format for applications using this function, define feature test macro __LIBASCII as described “__LIBASCII” on page 7.

Note: The inet_ntoa() function has a dependency on the level of the Enhanced ASCII Extensions. See “Enhanced ASCII support” on page 2113 for details.

Special behavior for C++: To use this function with C++, you must use the _XOPEN_SOURCE_EXTENDED 1 feature test macro.

Returned value

Returns a pointer to the Internet address expressed in dotted-decimal notation. The storage pointed to exists on a per-thread basis and is overwritten by subsequent calls.

Related information

- “arpa/inet.h” on page 17
- “netinet/in.h” on page 53
- “sys/socket.h” on page 74
- “sys/types.h” on page 75
- “inet_addr() — Translate an Internet address into network byte order” on page 856
- “inet_lnaof() — Translate a local network address into host byte order” on page 858
- “inet_makeaddr() — Create an Internet host address” on page 859
- “inet_netof() — Get the network number from the Internet host sddress” on page 860
- “inet_network() — Get the network number from the decimal host address” on page 861
- “inet_ntop() — Convert Internet address format from binary to text”
- “inet_pton() — Convert Internet address format from text to binary” on page 865

inet_ntop() — Convert Internet address format from binary to text

Standards

Standards / Extensions	C or C++	Dependencies
RFC2553 Single UNIX Specification, Version 3	both	z/OS V1R2

Format

```
#define _OPEN_SYS_SOCK_IPV6
#include <arpa/inet.h>

const char *inet_ntop(int af, const void *src, char *dst, socklen_t size);
```

SUSV3:

```
#define _POSIX_C_SOURCE 200112L
#include <arpa/inet.h>

const char *inet_ntop(int af, const void *__restrict__ src,
                      char * __restrict__ dst, socklen_t size);
```

General description

The inet_ntop() function converts from an Internet address in binary format, specified by *src*, to standard text format, and places the result in *dst*, when *size*, the space available in *dst*, is sufficient. The argument *af* specifies the family of the Internet address. This can be AF_INET or AF_INET6.

The argument *src* points to a buffer holding an IPv4 Internet address if the *af* argument is AF_INET, or an IPv6 Internet address if the *af* argument is AF_INET6. The address must be in network byte order.

The argument *dst* points to a buffer where the function will store the resulting text string. The *size* argument specifies the size of this buffer. The application must specify a non-NULL *dst* argument. For IPv6 addresses, the buffer must be at least 46 bytes. For IPv4 addresses, the buffer must be at least 16 bytes.

In order to allow applications to easily declare buffers of the proper size to store IPv4 and IPv6 addresses in string form, the following two constants are defined in <netinet/in.h>:

```
#define INET_ADDRSTRLEN 16
#define INET6_ADDRSTRLEN 46
```

Note: The inet_ntop() function has a dependency on the level of the Enhanced ASCII Extensions. See “Enhanced ASCII support” on page 2113 for details.

Returned value

If successful, inet_ntop() returns a pointer to the buffer containing the converted address.

If unsuccessful, inet_ntop() returns NULL and sets errno to one of the following values:

Error Code
Description

EAFNOSUPPORT
The address family specified in *af* is unsupported.

ENOSPC
The destination buffer *size* is too small.

Note: For Enhanced ASCII usage, the inet_ntop() function has a dependency on the level of the Enhanced ASCII Extensions. See “Enhanced ASCII support” on page 2113 for details.

Related information

- “arpa/inet.h” on page 17
- “netinet/in.h” on page 53
- “sys/socket.h” on page 74
- “inet_addr() — Translate an Internet address into network byte order” on page 856
- “inet_makeaddr() — Create an Internet host address” on page 859
- “inet_netof() — Get the network number from the Internet host sddress” on page 860
- “inet_network() — Get the network number from the decimal host address” on page 861
- “inet_ntoa() — Get the decimal Internet host address” on page 862
- “inet_pton() — Convert Internet address format from text to binary” on page 865

inet_pton() — Convert Internet address format from text to binary

Standards

Standards / Extensions	C or C++	Dependencies
RFC2553 Single UNIX Specification, Version 3	both	z/OS V1R2

Format

```
#define _OPEN_SYS_SOCK_IPV6
#include <arpa/inet.h>

int inet_pton(int af, const char *src, void *dst);
```

SUSV3:

```
#define _POSIX_C_SOURCE 200112L
#include <arpa/inet.h>

int inet_pton(int af, const char *__restrict__ src, void *__restrict__ dst);
```

General description

The inet_pton() function converts an Internet address in its standard text format into its numeric binary form. The argument *af* specifies the family of the address.

Note: AF_INET and AF_INET6 address families are currently supported.

The input argument *src* is a null terminated string. It points to the string being passed in. The argument *dst* points to a buffer into which inet_pton() stores the numeric address. The address is returned in network byte order. The caller must ensure that the buffer pointed to by *dst* is large enough to hold the numeric address.

If the *af* argument is AF_INET, inet_pton() accepts a string in the standard IPv4 dotted-decimal form:

```
ddd.ddd.ddd.ddd
```

where *ddd* is a 1 to 3 digit decimal number between 0 and 255.

If the *af* argument is AF_INET6, the *src* string must be in one of the following standard IPv6 text forms:

1. The preferred form is *x:x:x:x:x: x:x: x:x:*, where the *x*'s are the hexadecimal values of the eight 16-bit pieces of the address. Leading zeros in individual fields can be omitted, but there should be at least one numeral in every field.
2. A string of contiguous zero fields in the preferred form can be shown as *::* The *::* can only appear once in an address. Unspecified addresses *(0:0:0:0:0:0:0:0:)* may be represented simply as *::*.
3. A third form that is sometimes more convenient when dealing with a mixed environment of IPv4 and IPv6 is *x:x:x:x:x:x:d.d.d.d.*, where *x*'s are the hexadecimal values of the six high-order 16-bit pieces of the address, and the *d*'s are the decimal values of the four low-order 8-bit pieces of the address (standard IPv4 representation).

Notes:

1. A more extensive description of the IPv6 standard representations can be found in RFC2373.
2. The inet_pton() function has a dependency on the level of the Enhanced ASCII Extensions. See "Enhanced ASCII support" on page 2113 for details.

Returned value

If successful, inet_pton() returns 1 and stores the binary form of the Internet address in the buffer pointed to by *dst*.

If unsuccessful because the input buffer pointed to by *src* is not a valid string, inet_pton() returns 0.

If unsuccessful because the *af* argument is unknown, inet_pton() returns -1 and sets errno to one of the following values:

Error Code
Description

EAFNOSUPPORT
The address family specified in *af* is unsupported.

Related information

- "arpa/inet.h" on page 17
- "sys/socket.h" on page 74
- "inet_addr() — Translate an Internet address into network byte order" on page 856
- "inet_makeaddr() — Create an Internet host address" on page 859
- "inet_netof() — Get the network number from the Internet host sddress" on page 860
- "inet_network() — Get the network number from the decimal host address" on page 861
- "inet_ntoa() — Get the decimal Internet host address" on page 862
- "inet_ntop() — Convert Internet address format from binary to text" on page 863

initgroups() — Initialize the supplementary group ID list for the process

Standards

Standards / Extensions	C or C++	Dependencies
z/OS UNIX	both	

Format

```
#define _OPEN_SYS
#include <sys/types.h>
#include <grp.h>

int initgroups(const char *user, const gid_t basegid);
```

General description

The initgroups() function obtains the supplementary group membership of *user*, and sets the current process supplementary group IDs to that list. The *basegid* is also included in the supplementary group IDs list.

The caller of this function must be a superuser or must specify the password of the target user name specified on the initgroups() call - issue the passwd() function before initgroups().

Returned value

If successful, initgroups() returns 0.

If unsuccessful, initgroups() returns -1 and sets errno to one of the following values:

Error Code
Description

EINVAL
The number of supplementary groups for the specified user plus the basegid group exceeds the maximum number of groups allowed, or a non-valid *user* is specified.

EMVSERR
An MVS environmental or internal error occurred.

EMVSSAF2ERR
The Security Authorization Facility (SAF) had an error.

EPERM
The caller is not authorized, only authorized users are allowed to alter the supplementary group IDs list.

Related information

- "grp.h" on page 32
- "sys/types.h" on page 75
- "getgroupsbyname() — Get supplementary group IDs by user name" on page 716
- "setgroups() — Set the supplementary group ID list for the process" on page 1538

initstate() — Initialize generator for random()

Standards

Standards / Extensions	C or C++	Dependencies
XPG4.2 Single UNIX Specification, Version 3	both	

Format

```
#define _XOPEN_SOURCE_EXTENDED 1
#include <stdlib.h>

char *initstate(unsigned seed, char *state, size_t size);
```

General description

The initstate() function allows a state array, pointed to by the *state* argument, to be initialized for future use in calls to the random() functions by the calling thread. The *size* argument, which specifies the size in bytes of the state array, is used by the initstate() function to decide how sophisticated a random-number generator to use; the larger the state array, the more random the numbers. Values for the amount of state information are 8, 32, 64, 128, and 256 bytes. While other amounts are rounded down to the nearest known value. The *seed* argument specifies a starting point for the random-number sequence and provides for restarting at the same point. The initstate() function returns a pointer to the previous state information array.

Returned value

If successful, initstate() returns a pointer to the previous state array.

If unsuccessful, initstate() returns a NULL pointer.If initstate() is called with *size* less than 8, it will return NULL.

Related information

- "stdlib.h" on page 70
- "drand48() — Pseudo-random number generator" on page 402
- "random() — A better random-number generator" on page 1375
- "setstate() — Change generator for random()" on page 1590
- "srandom() — Use seed to initialize generator for random()" on page 1718

insque() — Insert an element into a doubly-linked list

Standards

Standards / Extensions	C or C++	Dependencies
XPG4.2 Single UNIX Specification, Version 3	both	

Format

```
#define _XOPEN_SOURCE_EXTENDED 1
#include <search.h>

void insque(void *element, void *pred);
```

General description

The insque() function inserts the element pointed to by *element* into a doubly-linked list immediately after the element pointed to by *pred*. The function operates on pointers to structures which have a pointer to their successor in the list as their first element, and a pointer to their predecessor as the second. The application is free to define the remaining contents of the structure, and manages all storage itself. To insert the first element into a linear (non-circular) list, an application would call *insque(element, NULL);*. To insert the first element into a circular list, the application would set the element's forward and back pointers to point to the element.

Returned value

insque() returns no values.

Related information

- "search.h" on page 62
- "remque() — Remove an element from a double linked list" on page 1430

ioctl() — Control device

Standards

Standards / Extensions	C or C++	Dependencies
XPG4.2 Single UNIX Specification, Version 3	both	

Format

Terminals:

```
#include <sys/ioctl.h>

int ioctl(int fildes int cmd, ... /* arg */);
```

Sockets:

```
#define _XOPEN_SOURCE_EXTENDED 1

/**     OR      **/

#define _OE_SOCKETS

#include <sys/ioctl.h>
#include <net/rtrouteh.h>
#include <net/if.h>
int ioctl(int fildes, int cmd, ... /* arg */);
```

STREAMS:

```
#define _XOPEN_SOURCE_EXTENDED 1
#include <stropts.h>
int ioctl(int fildes int cmd, ... /* arg */);
```

General description

ioctl() performs a variety of control functions on devices. The *cmd* argument and an optional third argument (with varying type) are passed to and interpreted by the device associated with *fildes*.

The *cmd* argument selects the control function to be performed and will depend on the device being addressed.

The *arg* argument represents additional information that is needed by this specific device to perform the requested function. The type of *arg* depends upon the particular control request, but it is either an integer or a pointer to a device-specific data structure.

ioctl() information is divided into the following section s:

- Terminals
- Sockets
- STREAMS
- ACLs

Terminals

The following ioctl() commands are used with terminals:

Command
: **Description**

TIOCSWINSZ
: Set window size. Used as the second operand in an ioctl() against a terminal. The window size information pointed to by the third operand is copied into an area in the kernel associated with the terminal, and a SIGWINCH signal is generated against the foreground process group.

TIOCGWINSZ
: Get window size. Used as the second operand in an ioctl() against a terminal. The current window size is returned in the area pointed to by the third operand - a winsize structure.

The winsize structure is the third operand in an ioctl() call when you use TIOCSWINSZ or TIOCGWINSZ. The structure contains four unsigned short integers:

Field **Description**

ws_row
: Number of rows in the window, in characters.

ws_col
: Number of columns in the window, in characters. This assumes single-byte characters. Multibyte characters may take more room.

ws_xpixel
: Horizontal size of the window, in pixels.

ws_ypixel
: Vertical size of the window, in pixels.

Sockets

The following ioctl() commands are used with sockets:

Command
: **Description**

FIONBIO
: Sets or clears nonblocking I/O for a socket. *arg* is a pointer to an integer. If the integer is 0, nonblocking I/O on the socket is cleared. Otherwise, the socket is set for nonblocking I/O.

FIONREAD
: Gets the number of immediately readable bytes for the socket. *arg* is a pointer to an integer. Sets the value of the integer to the number of immediately readable characters for the socket.

FIONWRITE
: Returns the number of bytes that can be written to the connected peer AF_UNIX stream socket before the socket blocks or returns EWOULDBLOCK. The number of bytes returned is not guaranteed unless there is serialization by the using applications.

FIOGETOWN
: Returns the PID that has been set that designates the recipient of signals.

FIOSETOWN
: Sets the PID to be used when sending signals

FIOGETOWN and FIOSETOWN are equivalent to the F_GETOWN and F_SETOWN commands of fctl(). For information on the values for pid, refer to that function. This function is only valid for AF_INET stream sockets.

SECIGET
Gets the peer socket's security identity values for an AF_UNIX connected stream socket. The MVS user ID, effective UID, and effective GID of the peer process are returned in the **seci** structure, which is mapped by BPXYSECI. This option is valid only for the AF_UNIX domain.

SECIGET_T
Returns both the process and, if available, the task level security information of the peer for an AF_UNIX stream connected to the socket. The task level security information is from the task that issued the connect() or accept(). The security information is returned in a struct __sect_s as defined in <sys/ioctl.h>. The security information is not available until accept() completes. The availability of the peer's task level security data is determined by the task level userid length field. If zero, the peer does not have task level security data.

SIOCADDRT
Adds a routing table entry. *arg* is a pointer to a **rtentry** structure, as defined in <net/rtrouteh.h>. The routing table entry, passed as an argument, is added to the routing tables. This option is valid only for the AF_INET domain.

SIOCATMARK
Queries whether the current location in the data input is pointing to out-of-band data. *arg* is a pointer to an integer. SIOCATMARK sets the argument to 1 if the socket points to a mark in the data stream for out-of-band data; otherwise, it sets the argument to 0. Refer to recv(), recvfrom() and recvmsg() for more information on receiving out-of-band data.

SIOCDELRT
Deletes a routing table entry. *arg* is a pointer to a **rtentry** structure, as defined in <net/rtrouteh.h>. If it exists, the routing table entry passed as an argument is deleted from the routing tables. This option is valid only for the AF_INET domain.

SIOCGIFADDR

Gets the network interface address. *arg* is a pointer to an **ifreq** structure, as defined in <net/if.h>. The interface address is returned in the argument. This option is valid only for the AF_INET domain.

This macro is protected by the _OPEN_SYS_IF_EXT feature.

SIOCGIFBRDADDR

Gets the network interface broadcast address. *arg* is a pointer to an **ifreq** structure, as defined in <net/if.h>. The interface broadcast address is returned in the argument. This option is valid only for the AF_INET domain.

This macro is protected by the _OPEN_SYS_IF_EXT feature.

SIOCGIFCONF

Gets the network interface configuration. *arg* is a pointer to an **ifconf** structure, as defined in <net/if.h>. The interface configuration is returned in the buffer pointed to by the **ifconf** structure. The returned data's length is returned in the field that had originally contained the length of the buffer. This option is valid only for the AF_INET domain.

This macro is protected by the _OPEN_SYS_IF_EXT feature.

SIOCGIFCONF6

Gets the name, address, and other information about the IPv6 network interfaces that are configured. This is similar to the SIOCGIFCONF command for IPv4.

To request OSM interfaces, the application must have READ authorization to the EZB.OSM.sysname.tcpname resource.

A struct __net_ifconf6header_s is passed as the argument of the ioctl. This structure specifies the buffer where the configuration information is to be written and is returned with the number of entries and entry length of each struct, and __net_ifconf6entry_s that was written to the output buffer. These structures are defined in <sys/ioctl.h>.

If __nif6h_buflen and __nif6h_buffer are both zero, a query function is performed and the header is returned with:

__nif6h_version

The maximum supported version.

Note: If the version number is supplied (not zero), the entry length returned will be for the specified version. (If it is supported)

__nif6h_entries

The total number of entries that will be output.

__nif6h_entrylen

The length of each individual entry.

If a call to get information fails with either

```
errno = ERANGE, or
errno = EINVAL and __nif6h_version has changed
```

The call was converted into a query function and the header has been filled in as described above. In these cases, the content of the output buffer is undefined.

If Common INET is configured and multiple TCP/IP stacks are attached to the socket, the output from each stack that is enabled for IPv6 will be concatenated in the output buffer and the header will contain the total number of entries returned from all the stacks. The version returned with the query function will be the highest version supported by all the stacks.

This ioctl can be issued on an AF_INET or AF_INET6 socket.

Error Code

Description

EAFNOSUPPORT

No IPv6 enabled TCP/IP stacks are active.

EINVAL

The input version number is not supported.

ERANGE

The buffer is too small to contain all of the IPv6 network interface entries.

SIOCGIFDSTADDR

Gets the network interface destination address. *arg* is a pointer to an **ifreq** structure, as defined in <net/if.h>. The interface destination (point-to-point) address is returned in the argument. This option is valid only for the AF_INET domain.

This macro is protected by the _OPEN_SYS_IF_EXT feature.

SIOCGIFFLAGS

Gets the network interface flags. *arg* is a pointer to an **ifreq** structure, as defined in <net/if.h>. The interface flags are returned in the argument. This option is valid only for the AF_INET domain.

This macro is protected by the _OPEN_SYS_IF_EXT feature.

SIOCGIFMETRIC

Gets the network interface routing metric. *arg* is a pointer to an **ifreq** structure, as defined in <net/if.h>. The interface routing metric is returned in the argument. This option is valid only for the AF_INET domain.

This macro is protected by the _OPEN_SYS_IF_EXT feature.

SIOCGIFMTU

Gets the network interface MTU (maximum transmission unit). arg is a pointer to an ifreq structure, as defined in <net/if.h>. The interface MTU is returned in the argument, arg->ifr_mtu. This option is only valid for the AF_INET domain.

This macro is protected by the _OPEN_SYS_IF_EXT feature.

SIOCGIFNETMASK

Gets the network interface network mask. *arg* is a pointer to an **ifreq** structure, as defined in <net/if.h>. The interface network mask is returned in the argument. This option is valid only for the AF_INET domain.

This macro is protected by the _OPEN_SYS_IF_EXT feature.

SIOCGSPLXFQDN

Gets the fully qualified domain name for a given server and domain name in a sysplex. This is an special purpose command to support applications that have registered with WorkLoad Manager (WLM) for connection optimization services using the Domain Name System (DNS). 'arg' is a pointer to sysplexFqDn structure, as defined in <ezbzsdnc.h>. sysplexFqDn contains pointer to sysplexFqDnData structure, as defined in <ezbzsdnc.h>.

sysplexFqDnData structure contains server name(input), group name(input) and fully qualified domain name(output).

ioctl() with the SIOCGSPLXFQDN command will fail if:

Error Code

Description

EFAULT

Write user storage failed

EINVAL

One of the following:
- Group name required
- Buffer length not valid
- Socket call parameter error

ENXIO
One of the following:
- Sysplex address not found
- Res not found In DNS
- Time out
- Time Unexpected Error

Example: The following is an example of the ioctl() call used with SIOCGSPLXFQDN.

```
#include <ezbzsdnc.h>
    sysplexFqDn       splxFqDn;
    sysplexFqDnData   splxData;
    int               rc;

    splxFqDn.splxVersion = splxDataVersion;
    splxFqDn.splxBufLen  = sizeof(sysplexFqDnData);
    splxFqDn.splxBufAddr = &splxData;

    /*  Assign values to splxData.groupName, */
    /*  splxData.serverName if required      */
        .
        .

    /* Get the fully qualified domain name   */
    rc = ioctl(s,SIOCGSPLXFQDN, (char *) &splxFqDn);

    /* splxData.domainName contains the fully*/
    /* qualified domain name.                */
```

SIOCSECENVR

Used to SET or GET the security environment for a server socket. arg points to a struct __seco_s where element __seco_argument is set to 1 for a SET and 2 for a GET request.

When used with the SET argument, the AF_UNIX stream socket server will designate the server socket as one that requires the full security environment of the connecting client to be available before the connect will complete successfully. During connect processing, connect obtains the security environment of the connector and anchors it off the connector's socket for use by the server. If the security environment cannot be obtained during connect processing, the connect will fail. This command has no effect on sockets that do not become server sockets.

When used with the GET argument, the AF_UNIX stream socket server will copy the previously SET security environment from the connector's address space to the server's address space so it can be used as input on calls to the security product. This command has meaning only for server sockets that previously issued SIOCSECENVR with the SET argument.

SIOCSIFMETRIC

Sets the network interface routing metric. *arg* is a pointer to an **ifreq** structure, as defined in <net/if.h>. SIOCSIFMETRIC sets the interface routing metric to the value passed in the argument. This option is valid only for the AF_INET domain.

This macro is protected by the _OPEN_SYS_IF_EXT feature.

SIOCSPARTNERINFO
Sets an indicator to retrieve the partner security credentials during connection setup and saves the information, enabling an application to issue a SIOCGPARTNERINFO ioctl without suspending the application, or

at least minimizing the time to retrieve the information. The SIOCSPARTNERINFO ioctl must be issued prior to the SIOCGPARTNERINFO ioctl. For more information, see *z/OS Communications Server: IP Programmer's Guide and Reference*.

SIOCSVIPA
Defines or deletes an IPv4 dynamic VIPA. *arg* is a pointer to a **dvreq** structure as defined in <ezbzdvpc.h>. This option is valid only for the AF_INET domain.

SIOCSVIPA6
Defines or deletes an IPv6 dynamic VIPA. *arg* is a pointer to a **dvreq6** structure as defined in <ezbzdvpc.h>. This option is valid only for the AF_INET6 domain.

SIOCTIEDESTHRD
Associates (ties) or disassociates (unties) a descriptor with a thread. `arg` is a pointer to an int. When `*arg` is 1, the descriptor is tied to the calling thread. When `*arg` is 0, the descriptor is untied from the calling thread. If the task should terminate before the descriptor is closed or untied from the task, z/OS UNIX file system thread termination processing will close the descriptor. This command can be used on both heavy weight and medium weight threads.

Special behavior for C++: To use this function with C++, you must use the _XOPEN_SOURCE_EXTENDED 1 feature test macro.

Terminal and sockets returned value

If successful, ioctl() returns 0.

If unsuccessful, ioctl() returns -1 and sets errno to one of the following values:

Error Code
Description

EBADF
The *fildes* parameter is not a valid socket descriptor.

EINVAL
The request is not valid or not supported.

EIO The process group of the process issuing the function is an orphaned, background process group, and the process issuing the function is not ignoring or blocking SIGTTOU.

EMVSPARM
Incorrect parameters were passed to the service.

ENODEV
The device is incorrect. The function is not supported by the device driver.

ENOTTY
An incorrect file descriptor was specified. The file type was not character special.

Example

The following is an example of the ioctl() call.

```
int s;
int dontblock;
int rc;
⋮
/* Place the socket into nonblocking mode */
dontblock = 1;
rc = ioctl(s, FIONBIO, (
char *) &dontblock);
⋮
```

STREAMS

The following ioctl() commands are used with STREAMS:

L_PUSH

Pushes the module whose name is pointed to by *arg* onto the top of the current STREAM, just below the STREAM head. It then calls the open() function of the newly-pushed module.

ioctl() with the I_PUSH command will fail if:

Error Code
Description

EINVAL
Non-valid module name.

ENXIO
Open function of new module failed.

ENXIO
Hang-up received on *fildes*

L_POP

Removes the module just below the STREAM pointed to by *fildes*. The *arg* argument should be 0 in an I_POP request.

ioctl() with the I_POP command will fail if:

Error Code
Description

EINVAL
No module present in the STREAM.

ENXIO
Hang-up received on *fildes*.

L_LOOK

Retrieves the name of the module just below the STREAM head of the STREAM pointed to by *fildes* and places it in a character string pointed to by *arg*. The buffer pointed to by *arg* should be at least FMNAMESZ+1 bytes long, where FMNAMESZ is defined in <stropts.h>.

ioctl() with the I_LOOK command will fail if:

Error Code
Description

EINVAL
No module present in the STREAM.

L_FLUSH

This request flushes read and/or write queues, depending on the value of *arg*. Valid *arg* values are:

FLUSHR
Flush all read queues.

FLUSHW
Flush all write queues.

FLUSHRW
Flush all read and all write queues.

ioctl() with the I_FLUSH command will fail if:

Error Code
Description

EAGAIN or ENOSR
Unable to allocate buffers for flush message.

EINVAL
Non-valid *arg* value.

ENXIO
Hang-up received on *fildes*.

I_FLUSHBAND
Flushes a particular band of messages. The *arg* argument points to a bandinfo structure. The bi_flag member may be one of FLUSHER, FLUSHW, or FLUSHRW as described above. The bi_pri member determines the priority band to be flushed.

I_SETSIG
Requests that the STREAMS implementation send the SIGPOLL signal to the calling process when a particular event has occurred on the STREAM associated with *fildes*. I_SETIG supports an asynchronous processing capability in STREAMS. The value of *arg* is a bitmask that specifies the events for which the process should be signaled. It is the bitwise OR of any combination of the following constants:

S_RDNORM
A normal (priority band set to 0) message has arrived at the head of a STREAM head read queue. A signal will be generated even if the message is of zero length.

S_RDBAND
A message with a nonzero priority band has arrived at the head of a STREAM head read queue. A signal will be generated even if the message is of zero length.

S_INPUT
A message, other than a high-priority message, has arrived at the head of a STREAM head read queue. A signal will be generated even if the message is of zero length.

S_HIPRI
A high-priority message is present on a STREAM head read queue. A signal will be generated even if the message is of zero length.

S_OUTPUT
The write queue for normal data (priority band 0) just below the STREAM head is no longer full. This notifies the process that there is room on the queue for sending (or writing) normal data downstream.

S_WRNORM
Same as S_OUTPUT.

S_WRBAND
The write queue for a nonzero priority band just below the STREAM head is no longer full. This notifies the process that there is no room on the queue for sending (or writing) priority data downstream.

S_MSG
A STREAMS signal message that contains the SIGPOLL signal has reached the front of the STREAM head read queue.

S_ERROR
Notification of an error condition has reached the STREAM head.

S_HANGUP
When used in conjunction with S_RDBAND, SIGURG is generated instead of SIGPOLL when a priority message reaches the front of the STREAM head read queue.

If *arg* is 0, the calling process will be unregistered and will not receive further SIGPOLL signals for the STREAM associated with *fildes*.

Processes that wish to receive SIGPOLL signals must explicitly register to receive them using I_SETSIG. If several processes register to receive this signal for the same event on the same STREAM, each process will be signaled when the event occurs.

ioctl() with the I_SETSIG command will fail if:

Error Code
Description

EAGAIN
There were insufficient resources to store the signal request.

EINVAL
The value of *arg* is not valid.

EINVAL
The value of *arg* is 0 and the calling process is not registered to receive the SIGPOLL signal.

I_GETSIG
Returns the events for which the calling process is currently registered to be sent a SIGPOLL signal. The events are returned as a bitmask in an int pointed to by *arg*, where the events are those specified in the description of I_SETSIG above.

ioctl() with the I_GETSIG command will fail if:

Error Code
Description

EINVAL
Process is not registered to receive the SIGPOLL signal.

I_FIND
This request compares the names of all modules currently present in the STREAM to the name pointed to by *arg*, and returns 1 if the name module is present in the STREAM, or returns 0 if the named module is not present.

ioctl() with the I_FIND command will fail if:

Error Code
Description

EINVAL
arg does not contain a valid module name.

I_PEEK
This request allows a process to retrieve the information in the first message on the STREAM head read queue without taking the message off the queue. It is analogous to getmsg() except that this command does not remove the message from the queue. The *arg* argument points to a strpeek structure.

The maxlen member in the ctlbuf and databuf strbuf structure must be set to the number of bytes of control information and/or data information, respectively, to retrieve. The flags member may be marked RS_HIPRI or 0, as described by getmsg() - getpmsg(). If the process sets flags to RS_HIPRI, for example, I_PEEK will only look for a high-priority message on the STREAM head read queue.

I_PEEK returns 1 if a message was retrieved, and returns 0 if no message was found on the STREAM head read queue, or if the RS_HIPFI flag was set in flags and a high-priority message was not present on the STREAM head read queue. It does not wait for a message to arrive. On return, ctlbuf specifies information in the control buffer, databuf specifies information in the data buffer, and flags contains the value RS_HIPRI or 0.

I_SRDOPT
Sets the read mode using the value of the argument *arg*. Read modes are described in read(). Valid *arg* flags are:

RNORM
Byte-stream mode, the default.

RMSGD
Message-discard mode.

RMSGN
Message-nondiscarded mode.

The bitwise inclusive-OR of RMSGD and RMSGN will return EINVAL. The bitwise inclusive-OR of RNORM and either RMSGD or RMSGN will result in the other flag overriding RNORM which is the default.

In addition, treatment of control messages by the STREAM head may be changed by setting any of the following flag in *arg*:

RPROTNORM
Fail read() with EBADMSG if a message containing a control part is at the front of the STREAM head read queue.

RPROTDAT
Deliver the control part of a message as data when a process issues a read().

RPROTDIS
Discard the control part of a message, delivery any data portion, when a process issues a read().

ioctl() with the I_SRDOPT command will fail if:

Error Code
Description

EINVAL
The *arg* argument is not valid.

I_GRDOPT
Returns the current read mode setting, as described above, in an int pointed to by the argument *arg*. Read modes are described in read().

I_NREAD
Counts the number of data bytes in the data part of the first message on the STREAM head read queue and places this value in the init pointed to by *arg*. The return value for the command is the number of messages on the STREAM head read queue. For example, if 0 is returned in *arg*, but the ioctl() return value is greater than 0, this indicates that a zero-length message is next on the queue.

I_FDINSERT
Creates a message from specified buffer(s), adds information about another STREAM, and sends the message downstream. The message contains a control part and an optional data part. The data and control parts to be sent are distinguished by placement in separate buffers, as described below. The *arg* argument points to a **strfdinsert** structure.

The **len** member in the ctlbuf strbuf structure must be set to the size of a pointer plus the number of bytes of control information to be sent with the message. The *fildes* member specifies the file descriptor of the other STREAM, and the offset member, which must be suitably aligned for use as a pointer, specifies the offset from the start of the control buffer where I_FDINSERT will store a pointer whose interpretation is specific to the STREAM end. The **len** member in the databuf strbuf structure must be set to the number of bytes of data information to be sent with the message, or 0 if no data part is to be sent.

The **flags** member specifies the type of message to be created. A normal message is created if flags is set to 0, and a high-priority message is created if **flags** is set to RS_HIPRI. For non-priority messages, I_FDINSERT will block if the STREAM write queue is full due to internal flow control conditions. For priority messages, I_FDINSERT does not block on this condition. For non-priority messages, I_FDINSERT does not block when the write queue is full and O_NONBLOCK is set. Instead, it fails and sets errno to EAGAIN.

I_FDINSERT also blocks, unless prevented by lack of internal resources, waiting for the availability of message blocks in the STREAM, regardless of priority or whether O_NONBLOCK has been specified. No partial message is sent.

ioctl() with the I_FDINSERT command will fail if:

Error Code
Description

EAGAIN
A non-priority message is specified, the O_NONBLOCK flag is set, and the STREAM write queue is full due to internal flow control conditions.

EAGAIN or ENOSR
Buffers can not be allocated for the message that is to be created.

EINVAL
One of the following:

- The *fd* member of the **strfdinsert** structure is not a valid, open STREAM file descriptor.
- The size of a pointer plus *offset* is greater than the *len* member for the buffer specified through *ctlptr*
- the *offset* member does not specify a properly-aligned location in the data buffer.
- An undefined value is stored in **flags**

ENXIO
Hang-up received for *fd* or *fildes*.

ERANGE
The *len* member for the buffer specified through *databuf* does not fall within the range specified by the maximum and minimum packet sizes of the topmost STREAM module or the *len* member for the buffer specified through *databuf* is larger than the maximum configured size of the data part of a message; or the *len* member for the buffer specified through *ctlbuf* is larger than the maximum configured size of the control part of a message.

I_STR Constructs an internal STREAMS ioctl() message from the data pointed to by *arg*, and sends that message downstream.

This mechanism is provided to send ioctl() requests to downstream modules and drivers. It allows information to be sent with ioctl(), and returns to the process any information sent upstream by the downstream recipient. I_STR blocks until the system responds with either a positive or negative acknowledgement message, or until the request "times out" after some period of time. If the request times out, it fails with errno set to ETIME.

At most, one I_STR can be active on a STREAM. Further I_STR calls will block until the active I_STR completes at the STREAM head. The default timeout interval for these requests is 15 seconds. The O_NONBLOCK flag has no effect on this call.

To send requests downstream, *arg* must point to a **strioctl** structure.

The **ic_cmd** member is the internal ioctl() command intended for a downstream module or driver and **ic_timeout** is the number of seconds (-1 = infinite, 0 = use implementation-dependent timeout interval, >0 = as specified) an I_STR request will wait for acknowledgement before timing out. **ic_len** member has two uses: on input, it contains the length of the data argument passed in, and on return from the command, it contains the number of bytes being returned to the process (the *buffer* pointed to by **ic_dp** should be large enough to contain the maximum amount of data that any module or the driver in the STREAM can return.)

The STREAM head will convert the information pointed to by the **strioctl** structure to an internal ioctl() command message and send it downstream.

ioctl() with the I_STR command will fail if:

Error Code
Description

EAGAIN or ENOSR
Unable to allocate buffers for the ioctl() message.

EINVAL
This *ic_len* member is less than 0 or larger than the maximum configured size of the data part of a message, or *ic_timeout* is less than -1.

ENXIO
Hang-up received on *fildes*.

ETIME
A downstream ioctl() timed out before acknowledgement was received.

An I_STR can also fail while waiting for an acknowledgement if a message indicating an error or a hang-up is received at the STREAM head. In addition, an error code can be returned in the positive or negative acknowledgement message, in the event the ioctl() command sent downstream fails. For these cases, I_STR fails with errno set to the value in the message.

I_SWROPT
Sets the write mode using the value of the argument *arg*. Valid bit settings for *arg* are:

SNDZERO
Send a zero-length message downstream when a write() if 0 bytes occurs. To not send a zero-length message when a write() of 0 bytes occurs, this bit must not be set in *arg* (for example, *arg* would be set to 0).

ioctl() with the I_SWROPT command will fail if:

Error Code
Description

EINVAL
arg is not the above value.

I_GWROPT
Returns the current write mode setting as described above, in the int that is pointed to by the argument *arg*.

I_SENDFD
I_SENDFD creates a new reference to the open file description associated with the file descriptor *arg* and writes a message on the STREAMS-based pipes *fildes* containing the reference, together with the user ID and group ID of the calling process.

ioctl() with the I_SENDFD command will fail if:

Error Code
Description

EAGAIN
The sending STREAM is unable to allocate a message block to contain the file pointer; or the read queues of the receiving STREAM head is full and cannot accept the message sent by I_SENDFD.

EBADF
The *arg* argument is not a valid, open file descriptor.

EINVAL
The *fildes* argument is not connected to a STREAM pipe.

ENXIO
Hang-up received on *fildes*.

I_RECVFD

Retrieves the reference to an open file description from a message within a STREAMS-based pipe using the I_SENDFD command, and allocates a new file descriptor in the calling process that refers to this open file description. The *arg* argument is a pointer to an **strrecvfd** data structure as defined in <stropts.h>.

The **fd** member is a file descriptor. The **uid** and **gid** members are the effective user ID and effective group ID, respectively, of the sending process.

If O_NONBLOCK is not set I_RECVFD blocks until a message is present at the STREAM head. If O_NONBLOCK is set, I_RECVFD fails with errno set to EAGAIN if no message is present at the STREAM head.

If the message at the STREAM head is a message sent by an I_SENDFD, a new file descriptor is allocated for the open file descriptor referenced in the message. The new file descriptor is placed in the **fd** member of the **strrecvfd** structure pointed to by *arg*.

ioctl() with the I_RECVFD command will fail if:

Error Code
Description

EAGAIN
A message is not present at the STREAM head read queue and the O_NONBLOCK flag is set.

EBADMSG
The message at the STREAM head read queue is not a message containing a passed file descriptor.

EMFILE
The process has the maximum number of file descriptors currently open that is allowed.

ENXIO
Hang-up received on *fildes*.

I_LIST

This request allows the process to list all the module names on the STREAM, up to and including the topmost driver names. If *arg* is a NULL pointer, the return value is the number of modules, including the driver, that are on the STREAM pointed to by *fildes*. This lets the process allocate enough space for the module names. Otherwise, it should point to an **str_list** structure.

The **sl_nmods** member indicates the number of entries the process has allocated in the array. Upon return, the **sl_modlist** member of the **str_list** structure contains the list of module names. The number of entries that have been filled into the **sl_modlist** array is found in the **sl_nmode** member (the number includes the number of module including the driver). The return value from ioctl() is 0. The entries are filled in starting at the top of the STREAM and continuing downstream until either the end of the STREAM is reached, or the number of requested modules (**sl_nmods**) is satisfied.

ioctl() with the I_LIST command will fail if:

Error Code
Description

EAGAIN or ENOSR
Unable to allocate buffers.

EINVAL
The **sl_nmods** member is less than 1.

I_ATMARK
This request allows the process to see if the message at the head of the STREAM head read queue is marked by some module downstream. The *arg* argument determines how the checking is done when there may be multiple marked messages on the STREAM head read queue. It may take on the following values:

ANYMARK Check if the message is marked.

LASTMARK Check if the message is the last one marked on the queue.

The bitwise inclusive-OR of the flags ANYMARK and LASTMARK is permitted.

The return value is 1 if the mark condition is satisfied and 0 otherwise.

ioctl() with the I_ATMARK command will fail if:

EINVAL Non-valid *arg* value.

I_CKBAND
Check if he message of a given priority band exists on the STREAM head read queue. This returns 1 if a message of the given priority exists, 0 if no message exists, or -1 on error. *arg* should be of type int.

ioctl() with the I_CKBAND command will fail if :

EINVAL Non-valid *arg* value.

I_GETBAND
Return the priority band of the first message on the STREAM head read queue in the integer referenced by *arg*.

ioctl() with the I_GETBAND command will fail if:

ENODATA No message on the STREAM head read queue.

I_CANPUT
Check if a certain band is writable. *arg* is set to the priority band in question. The return value is 0 if the band is flow-controlled, 1 if the band is writable, or -1 on error.

ioctl() with the I_CANPUT command will fail if:

EINVAL Non-valid *arg* value.

I_SETCLTIME
This request allows the process to set the time the STREAM head will delay when a STREAM is closing and there is data on the write queues. Before closing each module or driver, if there is a data on its write queue, the STREAM head will delay for the specified amount of time to allow the data to drain. If, after the delay, data is still present, it will be flushed. The *arg* argument is a pointer to an integer specifying the number of milliseconds to delay, rounded up to the nearest valid value. If I_SETCLTIME is not performed on a STREAM, an implementation-dependent default timeout interval is used.

ioctl() with the I_SETCLTIME command will fail if:

EINVAL Non-valid *arg* value.

I_GETCLTIME
This request returns the close time delay in the integer pointed to by *arg*

Multiplexed STREAMS configurations: The following four commands are used for connecting and disconnecting multiplexed STREAMS configurations. These commands use an implementation-dependent default timeout interval.

I_LINK
Connects two STREAMS, where *fildes* is the file descriptor of the STREAM connected to the multiplexing driver, and *arg* is the file descriptor of the STREAM connected to another driver. The STREAM designated by *arg* gets connected below the multiplexing driver. I_LINK requires the multiplexing driver to the send an acknowledgement message to the STREAM head regarding the connection. This call returns a multiplexer ID number (an identifier used to disconnect the multiplexer; see (I_UNLINK) on success, and -1 on failure.

ioctl() with the I_LINK command will fail if:

Error Code
Description

EAGAIN or ENOSR
Unable to allocate STREAMS storage to perform the I_LINK.

EBADF
The *arg* argument is not a valid, open file descriptor.

EINVAL
The *fildes* does not support multiplexing; or *arg* is not a STREAM or is already connected downstream from a multiplexer, or the specified I_LINK operation would connect the STREAM head in more than one place in the multiplexed STREAM.

ENXIO
Hang-up received on *fildes*.

ETIME
Time out before acknowledgement message was received at STREAM head.

An I_LINK can also fail while waiting for the multiplexing driver to acknowledge the request, if a message indicating an error or a hang-up is received at the STREAM head of *fildes*. In addition, an error code can be returned in the positive or negative acknowledgement message. For these cases, I_LINK fails with errno set to the value in the message.

I_UNLINK
Disconnects the two STREAMs specified by *fildes* and *arg*. *fildes* is the file descriptor of the STREAM connected to the multiplexing driver. The *arg* argument is the multiplexer ID number that was returned by the I_LINK ioctl() command when a STREAM was connected downstream from the multiplexing driver. If *arg* is MUXID_ALL, then all STREAMs that were connected to *fildes* are disconnected. As in I_LINK, this command requires acknowledgement.

ioctl() with the I_UNLINK command will fail if:

Error Code
Description

EAGAIN or ENOSR
Unable to allocate buffers for the acknowledgement message.

EINVAL
Non-valid multiplexer ID number.

ENXIO
Hang-up received on *fildes*.

ETIME
Time out before acknowledgement message was received at STREAM head.

An I_UNLINK can also fail while waiting for the multiplexing driver to acknowledge the request if a message indicating an error or a hang-up is received at the STREAM head of *fildes* In addition, an error code can be returned in the positive or negative acknowledgement message. For these cases, I_UNLINK fails with errno set to the value in the message.

I_PLINK
Creates a *persistent connection* between two STREAMs, where *fildes* is the file descriptor of the STREAM connected to another driver. This call creates a persistent connection which can exist even if the file descriptor *fildes* associated with the upper STREAM to the multiplexing driver is closed. The STREAM designated by *arg* gets connected using a persistent connection below the multiplexing driver. I_PLINK requires the multiplexing driver to send an acknowledgement message to the STREAM head. This call returns a multiplexer ID number (an identifier that may be used to disconnect the multiplexer, see I_PUNLINK) on success, and -1 on failure.

ioctl() with the I_PLINK command will fail if:

Error Code
Description

EAGAIN or ENOSR
Unable to allocate STREAMS storage to perform the I_PLINK.

EBADF
The *arg* argument is not valid, open file descriptor.

EINVAL
The *fildes* argument does not support multiplexing; or *arg* is not a STREAM or is already connected downstream from a multiplexer; or the specified I_PLINK operation would connect the STREAM head in more than one place in the multiplexed STREAM.

ENXIO
Hang-up received on *fildes*.

ETIME
Time out before acknowledgement message was received at STREAM head.

An I_PLINK can also fail while waiting for the multiplexing driver to acknowledge the request, if a message indicating an error or a hang-up is received at the STREAM head of *fildes* In addition, an error code can be returned in the positive or negative acknowledgement message. For these cases, I_PLINK fails with errno set to the value in the message.

I_PUNLINK

Disconnects the two STREAMs specified by *fildes* and *arg* from a persistent connection. The *fildes* argument is the file descriptor of the STREAM connected to the multiplexing driver. The *arg* argument is the multiplexer ID number that was returned by the I_PLINK ioctl() command when a STREAM was connected downstream from the multiplexing driver. If *arg* is MUXID_ALL then all STREAMs which are persistent connections to *fildes* are disconnected. As in I_PLINK, this command requires the multiplexing driver to the acknowledge the request.

ioctl() with the I_PUNLINK command will fail if:

Error Code
Description

EAGAIN or ENOSR
Unable to allocate buffers for the acknowledgement message.

EINVAL
Non-valid multiplexer ID number.

ENXIO
Hang-up received on *fildes*.

ETIME
Time out before acknowledgement message was received at STREAM head.

An I_PUNLINK can also fail while waiting for the multiplexing driver to acknowledge the request if a message indicating an error or a hang-up is received at the STREAM head of *fildes*. In addition, an error code can be returned in the positive or negative acknowledgement message. For these cases, I_PUNLINK fails with errno set to the value in the message.

STREAMS returned value

If successful, ioctl() returns a value other than -1 that depends upon the STREAMS device control function.

If unsuccessful, ioctl() returns -1 and sets errno to one of the following values.

Note: It is impossible for ioctl() to perform any STREAMS type commands successfully, since z/OS UNIX services do not provide any STREAMS-based files. The function will always return -1 with errno set to indicate the failure. See "open() — Open a file" on page 1152 for more information.

Under the following general conditions, ioctl() will fail if:

Error Code
Description

EBADF
The *fildes* argument is not a valid open file descriptor.

EINTR
A signal was caught during the ioctl() operation.

EINVAL
The STREAM or multiplexer referenced by *fildes* is linked (directly or indirectly) downstream from a multiplexer.

If an underlying device driver detects an error, ioctl() will fail if:

Error Code
Description

EINVAL
The *cmd* or *arg* argument is not valid for this device.

EIO Some physical I/O error has occurred.

ENODEV
The *fildes* argument refers to a valid STREAMS device, but the corresponding device driver does not support ioctl().

ENOTTY
The *fildes* argument is not associated with a STREAMS device that accepts control functions.

ENXIO
The *cmd* or *arg* argument is not valid for this device driver, but the service requested can not be performed on this particular sub-device.

If a STREAM is connected downstream from a multiplexer, any ioctl() command except I_UNLINK and I_PUNLINK will set errno to EINVAL.

ACLs

The following ioctl() commands are used with ACLs:

Command
Description

SETFACL
Set ACL. Used to set information into an Access Control List. *arg* specifies the user buffer containing the input ACL which is mapped by struct ACL_buf followed immediately by an array of struct ACL_entrys. *arglen* specifies the combined length of the struct ACL_buf and the array of struct ACL_entrys. See *z/OS UNIX System Services Programming: Assembler Callable Services Reference* for more information about ACL_buf and the ACL_entrys.

GETFACL
Get ACL. Used to retrieve information from an Access Control List. *arg* specifies the user buffer into which the requested ACL will be returned. The data is mapped by struct ACL_BUF followed immediately by an array of struct ACL_entrys. See *z/OS UNIX System Services Programming: Assembler Callable Services Reference* for more information about ACL_buf and the ACL_entrys. *Arglen* specifies the combined length of the struct ACL and the array of struct ACL_entrys in the user buffer.

ACLs returned value

If successful, ioctl() returns 0.

If unsuccessful, ioctl() returns -1 and sets errno to one of the following values:

Error Code
Description

EBADF
The *fildes* parameter is not a valid file descriptor.

EINVAL
The request is not valid or not supported.

EMVSPARM
Incorrect parameters were passed to the service.

ENODEV
The device is incorrect. The function is not supported by the device driver.

Example

The following is an example of the ioctl() call.

```
int s;
int rc;
int acllen;
ext_acl_t aclbufp;
s = open("datafile", O_RDWR);
acllen = sizeof struct ACL_buf + (1024 * sizeof ACL_entry);
aclbufp = (ext_acl_t) malloc(acllen);
rc  =  ioctl(s, GETFACL, acllen, aclbufp)
```

Related information

- “net/if.h” on page 49
- “net/rtrouteh.h” on page 50
- “stropts.h” on page 72
- “sys/ioctl.h” on page 73
- “close() — Close a file” on page 288
- “fcntl() — Control open file descriptors” on page 474
- “getmsg(), getpmsg() — Receive next message from a STREAMS file” on page 735
- “open() — Open a file” on page 1152
- “pipe() — Create an unnamed pipe” on page 1179
- “poll() — Monitor activity on file descriptors and message queues” on page 1185
- “putmsg(), putpmsg() — Send a message on a STREAM” on page 1350
- “read() — Read from a file or socket” on page 1376
- “sigaction() — Examine or change a signal action” on page 1611
- “write() — Write data on a file or socket” on page 2085

__ipdbcs() — Retrieve the list of requested DBCS tables to load

Standards

Standards / Extensions	C or C++	Dependencies
z/OS UNIX	both	

Format

```
#include <__ftp.h>

struct __ipdbcss *__ipdbcs(void);
```

General description

The __ipdbcs() function determines the values that IP address resolution initialization found in the resolver configuration data set for the keywords LoadDBCSTables. If the LoadDBCSTables keywords are not found in the resolver configuration data set, the structure returned has a count of zero and each element in the structure list points to a NULL string.

Returned value

If successful, __ipdbcs() returns a NULL-terminated character string containing the complete structure *__ipdbcss* with each entry in *__ip_dbcs_list*[] initialized either to a valid name or to a NULL string. The number of valid names, up to the maximum of 8, is placed in *__ipdbcsnum*. If no table names are specified then *__ipdbcsnum* is set to zero.

If unsuccessful, __ipdbcs() returns NULL and stores one of the following error values in h_errno. __ipdbcs() is only unsuccessful if IP Address Resolution initialization fails to complete.

Error Code
Description

NO_RECOVERY
An error occurred that will continue to fail if tried again. Storage could not be obtained for this thread to contain the _res structure.

TRY_AGAIN
An error occurred while initializing the __res_state structure name selected, which can be retried.

Related information

- "__ftp.h" on page 32
- "__ipdspx() — Retrieve the data set prefix specified" on page 891
- "__ipmsgc() — Determine the case to use for FTP messages" on page 893

__ipDomainName() — Retrieve the resolver supplied domain name

Standards

Standards / Extensions	C or C++	Dependencies
Language Environment	both	OS/390 V2R9

Format

```
#include <__ftp.h>

char *__ipDomainName(void);
```

General description

Lets an application get the values which IP address resolution initialization established for the domain name (supplied by keywords Domain or DomainOrigin).

Returned value

If successful, __ipDomainName() returns the NULL-terminated character string which is the name found for the domain name or a NULL string if no domain name was found in the IP address resolution initialization.

If unsuccessful, __ipDomainName() returns NULL and stores one of the following error values in h_errno. The __ipDomainName() function is only unsuccessful if IP address resolution initialization fails to complete.

Error Code
Description

NO_RECOVERY
An error occurred that will continue to fail if tried again. Storage could not be obtained for this thread to contain the _res structure.

TRY_AGAIN
An error occurred while initializing the __res_state structure name selected, which can be retried.

Related information

- "__ftp.h" on page 32
- "__ipdbcs() — Retrieve the list of requested DBCS tables to load" on page 889
- "__ipdspx() — Retrieve the data set prefix specified"
- "__iphost() — Retrieve the resolver supplied hostname" on page 892
- "__ipmsgc() — Determine the case to use for FTP messages" on page 893
- "__ipnode() — Retrieve the resolver supplied node name" on page 893
- "__iptcpn() — Retrieve the resolver supplied jobname or user ID" on page 894

__ipdspx() — Retrieve the data set prefix specified

Standards

Standards / Extensions	C or C++	Dependencies
z/OS UNIX	both	

Format

```
#include <__ftp.h>

char *__ipdspx(void);
```

General description

The __ipdspx() function determines the value that IP address resolution initialization found in the resolver configuration data set for the keyword DataSetPrefix. If no DataSetPrefix keyword is found in the resolver configuration data set, then the default value is returned.

Returned value

If successful, __ipdspx() returns the NULL-terminated character string that was supplied in the configuration data set. If the configuration data set did not supply a value for the keyword DataSetPrefix, then __ipdspx() returns the string TCPIP.

If unsuccessful, __ipdspx() returns NULL and stores one of the following error values in h_errno. __ipdspx() is only unsuccessful if IP Address Resolution initialization fails to complete.

Error Code
Description

NO_RECOVERY
An error occurred that will continue to fail if tried again. Storage could not be obtained for this thread to contain the _res structure.

TRY_AGAIN
An error occurred while initializing the __res_state structure name selected, which can be retried.

Related information

- "__ftp.h" on page 32
- "__ipdbcs() — Retrieve the list of requested DBCS tables to load" on page 889
- "__ipmsgc() — Determine the case to use for FTP messages" on page 893

__iphost() — Retrieve the resolver supplied hostname

Standards

Standards / Extensions	C or C++	Dependencies
z/OS UNIX	both	

Format

```
#include <__ftp.h>

char *__iphost(void);
```

General description

The __iphost() function lets an application determine the values that IP address resolution initialization found in the resolver configuration data set for the keyword HOSTname. If the keyword is not found in the resolver configuration data set, the char string returned will be a NULL string.

Returned value

If successful, __iphost() returns the NULL-terminated character string, which is the name supplied on the HOSTname keyword found in the resolver configuration file.

If unsuccessful, __iphost() returns NULL and stores one of the following error values in h_errno. __iphost() is only unsuccessful if IP Address Resolution initialization fails to complete.

Error Code
Description

NO_RECOVERY
An error occurred that will continue to fail if tried again. Storage could not be obtained for this thread to contain the _res structure.

TRY_AGAIN
An error occurred while initializing the __res_state structure name selected, which can be retried.

Related information

- "__ftp.h" on page 32
- "__ipdbcs() — Retrieve the list of requested DBCS tables to load" on page 889
- "__ipdspx() — Retrieve the data set prefix specified" on page 891
- "__ipmsgc() — Determine the case to use for FTP messages" on page 893
- "__ipnode() — Retrieve the resolver supplied node name" on page 893
- "__iptcpn() — Retrieve the resolver supplied jobname or user ID" on page 894

__ipmsgc() — Determine the case to use for FTP messages

Standards

Standards / Extensions	C or C++	Dependencies
z/OS UNIX	both	

Format

```
#include <__ftp.h>

int __ipmsgc(void);
```

General description

The __ipmsgc() function determines the value that IP address resolution initialization found in the resolver configuration data set for the keyword MessageCase. If no MessageCase keyword is found in the resolver configuration data set, then the default value is returned.

The *init* argument returned is one of the following set of symbols defined in the __ftp.h header file, each one stands for a message case selection.

__MIXED
Represents mixed case value selected for the messages FTP will send.

__UPPER
Represents uppercase value selected for the messages FTP will send.

Returned value

__ipmsgc() is always successful and returns either the value of the __MIXED or the value of __UPPER for all requests. __MIXED is the default value.

Related information

- “__ftp.h” on page 32
- “__ipdbcs() — Retrieve the list of requested DBCS tables to load” on page 889 “__ipdspx() — Retrieve the data set prefix specified” on page 891

__ipnode() — Retrieve the resolver supplied node name

Standards

Standards / Extensions	C or C++	Dependencies
z/OS UNIX	both	

Format

```
#include <__ftp.h>

char *__ipnode(void);
```

General description

The __ipnode() function lets an application determine the values that IP address resolution initialization found as the NodeID name used by the VMCF platform. If the VMCF nodeID name is not found, the char string returned will be a NULL string.

Returned value

If successful, __ipnode() returns the NULL-terminated character string, which is the name found for the VMCF platform.

If unsuccessful, __ipnode() returns NULL and stores one of the following error values in h_errno. __ipnode() is only unsuccessful if IP Address Resolution initialization fails to complete.

Error Code
Description

NO_RECOVERY
An error occurred that will continue to fail if tried again. Storage could not be obtained for this thread to contain the _res structure.

TRY_AGAIN
An error occurred while initializing the __res_state structure name selected, which can be retried.

Related information

- “__ftp.h” on page 32
- “__ipdbcs() — Retrieve the list of requested DBCS tables to load” on page 889
- “__ipdspx() — Retrieve the data set prefix specified” on page 891
- “__iphost() — Retrieve the resolver supplied hostname” on page 892
- “__ipmsgc() — Determine the case to use for FTP messages” on page 893
- “__iptcpn() — Retrieve the resolver supplied jobname or user ID”

__iptcpn() — Retrieve the resolver supplied jobname or user ID

Standards

Standards / Extensions	C or C++	Dependencies
z/OS UNIX	both	

Format

```
#include <__ftp.h>

char *__iptcpn(void);
```

General description

The __iptcpn() function lets an application determine the values that IP address resolution initialization found in the resolver configuration data set for either of the keywords TCPIPuserid or TCPIPjobname, whichever is the last one read. If neither keyword is found in the resolver configuration data set, the char string returned will be a NULL string.

Returned value

If successful, __iptcpn() returns the NULL-terminated character string which is the name supplied on the TCPIPuserid or TCPIPjobname keyword found in the resolver configuration file.

If unsuccessful, __iptcpn() returns NULL and stores one of the following error values in h_errno. __iptcpn() is only unsuccessful if IP Address Resolution initialization fails to complete.

Error Code
Description

NO_RECOVERY
An error occurred that will continue to fail if tried again. Storage could not be obtained for this thread to contain the _res structure.

TRY_AGAIN
An error occurred while initializing the __res_state structure name selected, which can be retried.

Related information

- "__ftp.h" on page 32
- "setibmopt() — Set IBM TCP/IP image" on page 1540

isalnum() to isxdigit() — Test integer value

Standards

Standards / Extensions	C or C++	Dependencies
ISO C POSIX.1 XPG4 XPG4.2 C99 Single UNIX Specification, Version 3	both	

Format

```
#include <ctype.h>

int isalnum(int c);
int isalpha(int c);
int iscntrl(int c);
int isdigit(int c);
int isgraph(int c);
int islower(int c);
int isprint(int c);
int ispunct(int c);
int isspace(int c);
int isupper(int c);
int isxdigit(int c);
```

General description

The functions listed above, which are all declared in ctype.h, test a given integer value. The valid integer values for *c* are those representable as an *unsigned char* or EOF.

By default, the functions are defined as macros when ctype.h is included. For better performance, the macro forms are recommended over the functional forms.

However, to get the functional forms, do one or more of the following:

- For C only: do *not* include ctype.h.
- Specify `#undef`, for example, `#undef islower`
- Surround the call statement by parentheses, for example, `(islower)('a')`

Here are descriptions of each function in this group.

isalnum()
Test for an upper- or lowercase letter, or a decimal digit, as defined in the `alnum` locale source file and in the `alnum` class of the LC_CTYPE category of the current locale.

isalpha()
Test for an alphabetic character, as defined in the `alpha` locale source file and in the `alpha` class of the LC_CTYPE category of the current locale.

iscntrl()
Test for any control character, as defined in the `cntrl` locale source file and in the `cntrl` class of the LC_CTYPE category of the current locale.

isdigit()
Test for a decimal digit, as defined in the `digit` locale source file and in the `digit` class of the LC_CTYPE category of the current locale.

isgraph()
Test for a printable character excluding space, as defined in the `graph` locale source file and in the `graph` class of the LC_CTYPE category of the current locale.

islower()
Test for a lowercase character, as defined in the `lower` locale source file and in the `lower` class of the LC_CTYPE category of the current locale.

isprint()
Test for a printable character including space, as defined in the `print` locale source file and in the `print` class of the LC_CTYPE category of the current locale.

ispunct()
Test for any nonalphanumeric printable character, excluding space, as defined in the `punct` locale source file and in the `punct` class of the LC_CTYPE category of the current locale.

isspace()
Test for a white space character, as defined in the `space` locale source file and in the `space` class of the LC_CTYPE category of the current locale.

isupper()
Test for an uppercase character, as defined in the `upper` locale source file and in the `upper` class of the LC_CTYPE category of the current locale.

isxdigit()
Test for a hexadecimal digit, as defined in the `xdigit` locale source file and in the `xdigit` class of the LC_CTYPE category of the current locale.

The space, uppercase, and lowercase characters can be redefined by their respective class of the LC_CTYPE in the current locale. The LC_CTYPE category is discussed in the “Internationalization: Locales and Character Sets” in *z/OS XL C/C++ Programming Guide*.

Returned value

If the integer satisfies the test condition, these functions return nonzero.

If the integer does not satisfy the test condition, these functions return 0.

Example

CELEBI02

```
/* CELEBI02

   This example analyzes all characters between code 0x0 and
   code UPPER_LIMIT.
   The output of this example is a 256-line table showing the
   characters from 0 to 255, and a notification of whether they
   have the attributes tested.

 */
#include <stdio.h>
#include <ctype.h>

#define UPPER_LIMIT   0xFF

int main(void)
{
   int ch;

   for ( ch = 0; ch <= UPPER_LIMIT; ++ch )
   {
      printf("%3d ", ch);
      printf("%#04x ", ch);
      printf(" %c",  isprint(ch)  ? ch   : ' ');
      printf("%3s ", isalnum(ch)  ? "Alphanumeric" : " ");
      printf("%2s ", isalpha(ch)  ? "Alphabetic"  : " ");
      printf("%2s",  iscntrl(ch)  ? "Control"  : " ");
      printf("%2s",  isdigit(ch)  ? "Digit"  : " ");
      printf("%2s",  isgraph(ch)  ? "Graphic"  : " ");
      printf("%2s ",  islower(ch)  ? "Lower"  : " ");
      printf("%3s",  ispunct(ch)  ? "Punctuation" : " ");
      printf("%2s",  isspace(ch)  ? "Space"  : " ");
      printf("%3s",  isprint(ch)  ? "Printable" : " ");
      printf("%2s ",  isupper(ch)  ? "Upper"  : " ");
      printf("%2s ",  isxdigit(ch) ? "Hex"  : " ");

      putchar('\n');
   }
}
```

Related information

- “ctype.h” on page 21
- “isblank() — Test for blank character classification” on page 905
- “iswalnum() to iswxdigit() — Test wide integer value” on page 920
- “setlocale() — Set locale” on page 1552
- “tolower(), toupper() — Convert character case” on page 1897

isalpha() — Test for an alphabetic character

The information for this function is included in “isalnum() to isxdigit() — Test integer value” on page 895.

isascii() — Test for 7-bit US-ASCII character

Standards

Standards / Extensions	C or C++	Dependencies
XPG4 XPG4.2 Single UNIX Specification, Version 3	both	POSIX(ON)

Format

_XOPEN_SOURCE:

```
#define _XOPEN_SOURCE
#include <ctype.h>

int isascii(int c);
```

_ALL_SOURCE:

```
#define _ALL_SOURCE
#include <ctype.h>

int isascii(int c);
```

General description

Special behavior for _XOPEN_SOURCE: The isascii() function tests whether c is a 7-bit US-ASCII character code. The isascii() function is defined on all integer values.

Special behavior for _ALL_SOURCE: The isascii() function tests whether the character with EBCDIC encoding *c* in the current locale is a member of the set of POSIX Portable Characters and POSIX Control Characters shown below.

Returned value

Special behavior for _XOPEN_SOURCE: isascii() returns nonzero if *c* is a 7-bit US-ASCII character code between 0 and hexadecimal 007F inclusive; otherwise it returns 0.

Special behavior for _ALL_SOURCE: isascii() returns nonzero if *c* is the EBCDIC encoding in the current locale for a character in the set of POSIX Portable Characters and Control Characters; otherwise it returns 0.

Following is a list of the symbolic names, IBM-1047 EBCDIC code page encoding, and ISO8859-1 ASCII encoding for the set of POSIX Portable Characters and POSIX Control Characters. Cases where EBCDIC character encoding varies across EBCDIC Country Extended Code Pages (CECPs) are noted.

Table 33. Characters for which isascii() returns nonzero

Character (Symbolic Name)	IBM-1047 Encoding (Hex)	ISO8859-1 Encoding (Hex)
<NUL>	00	00
<SOH>	01	01
<STX>	02	02
<ETX>	03	03

Table 33. Characters for which isascii() returns nonzero (continued)

Character (Symbolic Name)	IBM-1047 Encoding (Hex)	ISO8859-1 Encoding (Hex)
<EOT>	37	04
<ENQ>	2D	05
<ACK>	2E	06
<BEL> <alert>	2F	07
<BS> <backspace>	16	08
<HT> <tab>	05	09
<NL> <newline>	15	0A
<VT> <vertical-tab>	0B	0B
<FF> <form-feed>	0C	0C
<CR> <carriage-return>	0D	0D
<SO>	0E	0E
<SI>	0F	0F
<DLE>	10	10
<DC1>	11	11
<DC2>	12	12
<DC3>	13	13
<DC4>	3C	14
<NAK>	3D	15
<SYN>	32	16
<ETB>	26	17
<CAN>	18	18
<EM>	19	19
<SUB>	3F	1A
<ESC>	27	1B
<IFS/IS4>	1C	1C
<IGS/IS3>	1D	1D
<IRS/IS2>	1E	1E
<IUS/ITB/IS1>	1F	1F
<space>	40	20
<exclamation-mark>	5A (cecp variant)	21
<quotation-mark>	7F	22
<number-sign>	7B (cecp variant)	23
<dollar-sign>	5B (cecp variant)	24
<percent-sign>	6C	25
<ampersand>	50	26
<apostrophe>	7D	27
<left-parenthesis>	4D	28
<right-parenthesis>	5D	29
<asterisk>	5C	2A
<plus-sign>	4E	2B

Table 33. Characters for which isascii() returns nonzero (continued)

Character (Symbolic Name)	IBM-1047 Encoding (Hex)	ISO8859-1 Encoding (Hex)
<comma>	6B	2C
<hyphen>	60	2D
<period>	4B	2E
<slash>	61	2F
<zero>	F0	30
<one>	F1	31
<two>	F2	32
<three>	F3	33
<four>	F4	34
<five>	F5	35
<six>	F6	36
<seven>	F7	37
<eight>	F8	38
<nine>	F9	39
<colon>	7A	3A
<semicolon>	5E	3B
<less-than-sign>	4C	3C
<equals-sign>	7E	3D
<greater-than-sign>	6E	3E
<question-mark>	6F	3F
<commercial-at>	7C (cecp variant)	40
<A>	C1	41
<B>	C2	42
<C>	C3	43
<D>	C4	44
<E>	C5	45
<F>	C6	46
<G>	C7	47
<H>	C8	48
<I>	C9	49
<J>	D1	4A
<K>	D2	4B
<L>	D3	4C
<M>	D4	4D
<N>	D5	4E
<O>	D6	4F
<P>	D7	50
<Q>	D8	51
<R>	D9	52
<S>	E2	53

Table 33. Characters for which isascii() returns nonzero (continued)

Character (Symbolic Name)	IBM-1047 Encoding (Hex)	ISO8859-1 Encoding (Hex)
<T>	E3	54
<U>	E4	55
<V>	E5	56
<W>	E6	57
<X>	E7	58
<Y>	E8	59
<Z>	E9	5A
<left-square-bracket>	AD (cecp variant)	5B
<backslash>	E0 (cecp variant)	5C
<right-square-bracket>	BD (cecp variant)	5D
<circumflex>	5F (cecp variant)	5E
<underscore>	6D	5F
<grave-accent>	79 (cecp variant)	60
<a>	81	61
<b>	82	62
<c>	83	63
<d>	84	64
<e>	85	65
<f>	86	66
<g>	87	67
<h>	88	68
<i>	89	69
<j>	91	6A
<k>	92	6B
<l>	93	6C
<m>	94	6D
<n>	95	6E
<o>	96	6F
<p>	97	70
<q>	98	71
<r>	99	72
<s>	A2	73
<t>	A3	74
<u>	A4	75
<v>	A5	76
<w>	A6	77
<x>	A7	78
<y>	A8	79
<z>	A9	7A
<left-brace>	C0 (cecp variant)	7B

Table 33. Characters for which isascii() returns nonzero (continued)

Character (Symbolic Name)	IBM-1047 Encoding (Hex)	ISO8859-1 Encoding (Hex)
<vertical-line>	4F (cecp variant)	7C
<right-brace>	D0 (cecp variant)	7D
<tilde>	A1 (cecp variant)	7E
<DEL>	07	7F

Related information

- “ctype.h” on page 21
- “toascii() — Translate integer to a 7-bit ASCII character” on page 1891

isastream() — Test a file descriptor

Standards

Standards / Extensions	C or C++	Dependencies
XPG4.2 Single UNIX Specification, Version 3	both	

Format

```
#define _XOPEN_SOURCE_EXTENDED 1
#include <stropts.h>

int isastream(int fildes);
```

General description

The isastream() function tests whether *filedes*, an open file descriptor, is associated with a STREAMS-based file.

Returned value

If successful, isastream() returns 1 if *fildes* refers to a STREAMS-based file and 0 if not.

If unsuccessful, isastream() returns -1 and sets errno to one of the following values.

Note: z/OS UNIX services do not supply any STREAMS devices or pseudodevices. It is impossible for isastream() to return 1 since there are no STREAMS-based file descriptors. It will return 0 unless *fildes* is not a valid open file descriptor, in which case it will return -1 with errno set to indicate the failure. See “open() — Open a file” on page 1152

Error Code
Description

EBADF
The *fildes* argument is not a valid open file descriptor.

Related information

- “stropts.h” on page 72

isatty() — Test if descriptor represents a terminal

Standards

Standards / Extensions	C or C++	Dependencies
POSIX.1 XPG4 XPG4.2 Single UNIX Specification, Version 3	both	

Format

```
#define _POSIX_SOURCE
#include <unistd.h>

int isatty(int fildes);
```

General description

Determines if a file descriptor, *fildes*, is associated with a terminal.

isatty() only works in an environment where either a controlling terminal exists, or stdin and stderr refer to tty devices. Specifically, it does not work in a TSO environment.

Returned value

isatty() returns 1 if the given file descriptor is a terminal, or 0 otherwise.

Special behavior for XPG4

isatty() returns 1 if the given file descriptor is a terminal, or 0 otherwise and sets errno to one of the following values:

Error Code
Description

EBADF
The *fildes* argument is not a valid open file descriptor.

ENOTTY
The *fildes* argument is not associated with a terminal.

Example

CELEBI03

```
/* CELEBI03

   This example determines if a file descriptor is
   associated with a terminal.

 */
#define _POSIX_SOURCE
#include <unistd.h>
#include <stdio.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>

void check_fd(int fd) {
  printf("fd %d is ", fd);
  if (!isatty(fd))
    printf("NOT ");
  puts("a tty");
}

main() {
  int p[2], fd;
  char fn[]="temp.file";

  if (pipe(p) != 0)
    perror("pipe() error");
  else {
    if ((fd = creat(fn, S_IWUSR)) < 0)
      perror("creat() error");
    else {
      check_fd(0);
      check_fd(fileno(stderr));
      check_fd(p[1]);
      check_fd(fd);
      close(fd);
      unlink(fn);
    }
    close(p[0]);
    close(p[1]);
  }
}
```

Output

```
fd 0 is a tty
fd 2 is a tty
fd 4 is NOT a tty
fd 5 is NOT a tty
```

Related information

- "unistd.h" on page 82
- "ttyname() — Get the name of a terminal" on page 1931

__isBFP() — Determine application floating-point format

Standards

Standards / Extensions	C or C++	Dependencies
	both	OS/390 V2R6

Format

```
#include <_Ieee754.h>

int __isBFP(void);
```

General description

The __isBFP() function determines the application floating-point mode.

Returned value

__isBFP() returns 1 if the floating-point mode of the caller is IEEE, and returns 0 if the floating-point mode of the caller is hexadecimal.

Related information

- "_Ieee754.h" on page 32
- "fp_read_rnd() — Determine rounding mode" on page 579
- "fp_swap_rnd() — Swap rounding mode" on page 580

isblank() — Test for blank character classification

Standards

Standards / Extensions	C or C++	Dependencies
Language Environment Single UNIX Specification, Version 3 C++ TR1 C99	both	

Format

```
#include <ctype.h>

int isblank(int c);
```

General description

Tests whether the current LC_CTYPE locale category assigns *c* the blank character attribute. The *tab* and *space* characters have the blank attribute in the POSIX locale (with name "POSIX" or "C").

To avoid infringing on the user's name space, this nonstandard function has two names. One name is prefixed with two underscore characters, and one name is not. The name without the prefix underscore characters is exposed only when you use LANGLVL(EXTENDED).

To use this function, you must either invoke the function using its external entry point name (for example, the name that begins with two underscore characters), or compile with LANGLVL(EXTENDED). When you use LANGLVL(EXTENDED) any relevant information in the header is also exposed.

Note: The isblank() function has a dependency on the level of the Enhanced ASCII Extensions. See "Enhanced ASCII support" on page 2113 for details.

Returned value

isblank() returns nonzero if the current LC_CTYPE locale category assigns *c* the blank character attribute.

Otherwise, isblank() returns 0.

Example

```
/* This example tests if c is a blank type.  */
#include <stdio.h>
#include <ctype.h>
#include <locale.h>

void check(char c) {
  if ((c != ' ') && (isprint(c)))
    printf(" %c  is ", c);
  else
    printf("x%02x is ", c);
  if (!isblank(c))
```

```
    printf("not ");
  puts("a blank type character");
}

main() {
  printf("\nIn LC_CTYPE category of locale \ with name \"%s\":\n",
     setlocale(LC_CTYPE, NULL));
  check('a');
  check(' ');
  check(0x00);
  check('\n');
  check('\t');
}
```

Output:

```
In LC_CTYPE category of locale with name "......";
 a  is not a blank type character
x40 is a blank type character
x00 is not a blank type character
x15 is not a blank type character
x05 is a blank type character
```

Related information

- "ctype.h" on page 21
- "iswalnum() to iswxdigit() — Test wide integer value" on page 920
- "iswblank() — Test for blank character classification" on page 922
- "setlocale() — Set locale" on page 1552

iscics() — Verify whether CICS is running

Standards

Standards / Extensions	C or C++	Dependencies
Language Environment	both	

Format

```
#include <cics.h>

int iscics(void);
```

General description

Determines whether the program is running under CICS.

Returned value

If your program is currently running under CICS, iscics() returns nonzero.

If not running under CICS, iscics() returns 0.

Example

CELEBI04

```
/* CELEBI04

   This example tests to see if the program is running under CICS.
   If not, it calls a subroutine ABCPGM; otherwise, it uses a CICS EXEC
   statement to invoke ABCPGM.
```

```
 */
#define _POSIX_SOURCE
#ifdef __cplusplus
  extern "OS" void ABCPGM(char *);
#else
  #pragma linkage(ABCPGM, OS)
  void ABCPGM(char *);
#endif

#include <stdio.h>
#include <cics.h>
#include <string.h>
#include <sys/wait.h>
#include <sys/types.h>
#include <unistd.h>

int main(void)
{
 char mydata[123];

 if (iscics() == 0)
    {
     /* not a CICS environment */
     ABCPGM(mydata);
    }

 else {
       /* this is a CICS environment */
       EXEC CICS, LINK PROGRAM,("ABCPGM  "), COMMAREA(mydata);
      }
}
```

Related information

- "cics.h" on page 18

iscntrl() — Test for control classification

The information for this function is included in "isalnum() to isxdigit() — Test integer value" on page 895.

isdigit() — Test for decimal-digit classification

The information for this function is included in "isalnum() to isxdigit() — Test integer value" on page 895.

isfinite() — Determines if its argument has a finite value

Standards

Standards / Extensions	C or C++	Dependencies
C99 Single UNIX Specification, Version 3 C/C++ DFP C++ TR1 C99	both	z/OS V1R8

Format

```
#define _ISOC99_SOURCE
#include <math.h>

int isfinite(real-floating x);
```

```
#define __STDC_WANT_DEC_FP__
#include <math.h>

int isfinite(real-floating x);  /* C only */
int isfinite(decimal-floating x);  /*C only */
bool isfinite(real-floating x);  /* C++ only */

#define _TR1_C99
#include <math.h>

bool isfinite(real-floating x);  /* C++ only */
```

General description

The isfinite() macro or function template determines if its argument has a finite value.

Function	Hex	IEEE
isfinite	X	X

Notes:

1. To use IEEE decimal floating-point, the hardware must have the Decimal Floating-Point Facility installed.
2. This function works in IEEE decimal floating-point format. See "IEEE Decimal Floating-Point" for more information.

Returned value

The isfinite() macro returns 1 if and only if its argument value is finite, else returns 0. The C++ function template returns true if and only if its argument value is finite, else returns false.

Special behavior in hex: The isfinite() macro always returns 1. The C++ function template always returns true.

Related information

- "math.h" on page 44

isgraph() — Test for graphic classification

The information for this function is included in "isalnum() to isxdigit() — Test integer value" on page 895.

isgreater() — Determines if X is greater than Y

Standards

Standards / Extensions	C or C++	Dependencies
C99 Single UNIX Specification, Version 3 C/C++ DFP C++ TR1 C99	both	z/OS V1R8

Format

```
#define _ISOC99_SOURCE
#include <math.h>

int isgreater(real-floating x, real-floating y);

#define __STDC_WANT_DEC_FP__
#include <math.h>

int isgreater(real-floating x, real-floating y);  /* C only */
int isgreater(decimal-floating x, decimal-floating y); /* C only */
bool isgreater(real-floating x, real-floating y);  /* C++ only */
bool isgreater(decimal-floating x, decimal-floating y); /* C++ only */

#define _TR1_C99
#include <math.h>

bool isgreater(real-floating x, real-floating y);  /* C++ onl */
```

General description

The isgreater() macro or function template determines whether the argument *x* is greater than *y*. It is equivalent to (*x*) > (*y*), but no exception is raised if *x* or *y* are NaN.

Function	Hex	IEEE
isgreater	X	X

Note:

- To use IEEE decimal floating-point, the hardware must have the Decimal Floating-Point Facility installed.
- This function works in IEEE decimal floating-point format. See "IEEE Decimal Floating-Point" for more information.

Returned value

The isgreater() macro returns 1 if the value of *x* is greater than *y*, else returns 0. The C++ function template returns true if the value of x is greater than y, else returns false.

Related information

- "math.h" on page 44

isgreaterequal() — Determines if X is greater than or equal to Y

Standards

Standards / Extensions	C or C++	Dependencies
C99 Single UNIX Specification, Version 3 C/C++ DFP C++ TR1 C99	both	z/OS V1R8

Format

```
#define _ISOC99_SOURCE
#include <math.h>

int isgreaterequal(real-floating x, real-floating y);

#define __STDC_WANT_DEC_FP__
#include <math.h>

int isgreaterequal(real-floating x, real-floating y);  /* C only */
int isgreaterequal(decimal-floating x, decimal-floating y);  /* C only */
bool isgreaterequal(real-floating x, real-floating y);  /* C++ only */
bool isgreaterequal(decimal-floating x, decimal-floating y);  /* C++ only */

#define _TR1_C99
#include <math.h>

bool isgreaterequal(real-floating x, real-floating y);  /* C++ only */
```

General description

The isgreaterequal() macro or function template determines whether the argument *x* is greater than or equal to *y*. It is equivalent to (*x*) >= (*y*), but no exception is raised if *x* or *y* are NaN.

Function	Hex	IEEE
isgreaterequal	X	X

Notes:

1. To use IEEE decimal floating-point, the hardware must have the Decimal Floating-Point Facility installed.
2. This function works in IEEE decimal floating-point format. See "IEEE Decimal Floating-Point" for more information.

Returned value

The isgreaterequal() macro returns 1 if the value of *x* is greater than or equal to *y*, else returns 0. The C++ function template returns true if the value of x is greater than or equal to y, else returns false.

Related information

- “math.h” on page 44

isinf() — Determines if X is ± infinity

Standards

Standards / Extensions	C or C++	Dependencies
C99 Single UNIX Specification, Version 3 C/C++ DFP C++ TR1 C99	both	z/OS V1R8

Format

```
#define _ISOC99_SOURCE
#include <math.h>

int isinf(real-floating x);

#define __STDC_WANT_DEC_FP__
#include <math.h>

int isinf(real-floating x);  /* C only */
int isinf(decimal-floating x);  /* C only */
bool isinf(real-floating x);  /* C++ only */

#define _TR1_C99
#include <math.h>

bool isinf(real-floating x);  /* C++ only */
```

General description

The isinf() macro or function template determines if its argument is plus or minus infinity.

Function	Hex	IEEE
isinf	X	X

Notes:

1. To use IEEE decimal floating-point, the hardware must have the Decimal Floating-Point Facility installed.
2. This function works in IEEE decimal floating-point format. See "IEEE Decimal Floating-Point" for more information.

Returned value

The isinf() macro returns 1 if the argument is plus or minus infinity, else returns 0. The C++ function template returns true if the argument is plus or minus infinity, else returns false.

Special behavior in hex: The isinf() macro returns zero. The C++ function template returns false.

Related information

- "math.h" on page 44

isless() — Determines if X is less than Y

Standards

Standards / Extensions	C or C++	Dependencies
C99 Single UNIX Specification, Version 3 C/C++ DFP C++ TR1 C99	both	z/OS V1R8

Format

```
#define _ISOC99_SOURCE
#include <math.h>

int isless(real-floating x, real-floating y);

#define __STDC_WANT_DEC_FP__
#include <math.h>

int isless(real-floating x, real-floating y);  /* C only */
int isless(decimal-floating x, decimal-floating y);  /* C only */
bool isless(real-floating x, real-floating y);  /* C++ only */
bool isless(decimal-floating x, decimal-floating y);  /* C++ only */

#define _TR1_C99
#include <math.h>

bool isless(real-floating x, real-floating y);  /* C++ only */
```

General description

The isless() macro or function template determines whether the argument *x* is less than *y*. It is equivalent to (*x*) < (*y*), but no exception is raised if *x* or *y* are NaN.

Function	Hex	IEEE
isless	X	X

Notes:

1. To use IEEE decimal floating-point, the hardware must have the Decimal Floating-Point Facility installed.
2. This function works in IEEE decimal floating-point format. See "IEEE Decimal Floating-Point" for more information.

Returned value

The isless() macro returns 1 if the value of *x* is less than *y*, else returns 0. The C++ function template returns true if the value of x is less than y, else returns false.

Related information

- "math.h" on page 44

islessequal() — Determines if X is less than or equal to Y

Standards

Standards / Extensions	C or C++	Dependencies
C99 Single UNIX Specification, Version 3 C/C++ DFP C++ TR1 C99	both	z/OS V1R8

Format

```
#define _ISOC99_SOURCE
#include <math.h>

int islessequal(real-floating x, real-floating y);
```

```
#define __STDC_WANT_DEC_FP__
#include <math.h>

int islessequal(real-floating x, real-floating y);  /* C only */
int islessequal(decimal-floating x, decimal-floating y);  /* C only */
bool islessequal(real-floating x, real-floating y);  /* C++ only */
bool islessequal(decimal-floating x, decimal-floating y);  /* C++ only */

#define _TR1_C99
#include <math.h>

bool islessequal(real-floating x, real-floating y);  /* C++ only */
```

General description

The islessequal() macro or function template determines whether the argument *x* is less than or equal to *y*. It is equivalent to (*x*) <= (*y*), but no exception is raised if *x* or *y* are NaN.

Function	Hex	IEEE
islessequal	X	X

Notes:

1. To use IEEE decimal floating-point, the hardware must have the Decimal Floating-Point Facility installed.
2. This function works in IEEE decimal floating-point format. See "IEEE Decimal Floating-Point" for more information.

Returned value

The islessequal() macro returns 1 if the value of *x* is less than or equal to *y*, else returns 0. The C++ function template returns true if the value of x is less than or equal to y, else returns false.

Related information

- "math.h" on page 44

islessgreater() — Determines if X is less or greater than Y

Standards

Standards / Extensions	C or C++	Dependencies
C99 Single UNIX Specification, Version 3 C/C++ DFP C++ TR1 C99	both	z/OS V1R8

Format

```
#define _ISOC99_SOURCE
#include <math.h>

int islessgreater(real-floating x, real-floating y);

#define __STDC_WANT_DEC_FP__
#include <math.h>

int islessgreater(real-floating x, real-floating y);  /* C only */
int islessgreater(decimal-floating x, decimal-floating y);  /* C only */
bool islessgreater(real-floating x, real-floating y);  /* C++ only */
bool islessgreater(decimal-floating x, decimal-floating y);  /* C++ only */

#define _TR1_C99
#include <math.h>

bool islessgreater(real-floating x, real-floating y);  /* C++ only */
```

General description

The islessgreater() macro or function template determines whether the argument *x* is less or greater than *y*. It is equivalent to (*x*) || (*y*), but no exception is raised if *x* or *y* are NaN.

Function	Hex	IEEE
islessgreater	X	X

Notes:

1. To use IEEE decimal floating-point, the hardware must have the Decimal Floating-Point Facility installed.
2. This function works in IEEE decimal floating-point format. See "IEEE Decimal Floating-Point" for more information.

Returned value

The islessgreater() macro returns 1 if the value of *x* is less or greater than *y*, else returns 0. The C++ function template returns true if the value of x is less or greater than y, else returns false.

Related information

- "math.h" on page 44

islower() — Test for lowercase

The information for this function is included in "isalnum() to isxdigit() — Test integer value" on page 895.

ismccollel() — Identify a multicharacter collating element

Standards

Standards / Extensions	C or C++	Dependencies
Language Environment	both	

Format

```
#include <collate.h>

int ismccollel(collel_t c);
```

General description

Determines whether a character is a multicharacter collating element. A collating element is a glyph, usually a character, that has a value used to define its order in a collating sequence. A multicharacter collating element is a sequence of two or more characters that are to be collated as one entity.

Returned value

ismccollel() returns:

1 if collel_t represents a multicharacter collating element

0 if collel_t represents a single-character collating element

-1 if collel_t is out of range, or otherwise not valid

Example

CELEBI05

```
/* CELEBI05

   This example prints all of the collating elements in the
   collating sequence, by using the &ismc. function to determine
   if the collating element is a multi-character collating
   element.

 */
#include <collate.h>
#include <locale.h>
#include <stdio.h>
#include <wchar.h>
#include <wctype.h>

main(int argc, char *argv[]) {
   collel_t e, *rp;
   int i;

   setlocale(LC_ALL, "");
   i = collorder(&rp);
   for (; i-- > 0; rp++) {
      if (ismccollel(*rp))
         printf("'%s' ", colltostr(*rp));
      else if (iswprint(*rp))
         printf("'%lc' ", *rp);
      else
         printf("'%x' ", *rp);
   }
}
```

Related information

- “collate.h” on page 19
- “cclass() — Return characters in a character class” on page 243
- “collequiv() — Return a list of equivalent collating elements” on page 295
- “collorder() — Return list of collating elements” on page 297
- “collrange() — Calculate the range list of collating elements” on page 298
- “colltostr() — Return a string for a collating element” on page 299
- “getmccoll() — Get next collating element from string” on page 734
- “getwmccoll() — Get next collating element from wide string” on page 803
- “maxcoll() — Return maximum collating element” on page 1039
- “strtocoll() — Return collating element for string” on page 1763

isnan() — Test for NaN

Standards

Standards / Extensions	C or C++	Dependencies
XPG4 XPG4.2 Single UNIX Specification, Version 3 C/C++ DFP C++ TR1 C99	both	z/OS V1R8

Format

```
#define  _XOPEN_SOURCE
#include <math.h>

int isnan(double x);

#define __STDC_WANT_DEC_FP__
#include <math.h>

int isnan(real-floating x);  /* C only */
int isnan(decimal-floating x);  /* C only */
bool isnan(real-floating x);  /* C++ only */
bool isnan(decimal-floating x);  /* C++ only */

#define _TR1_C99
#include <math.h>

bool isnan(double x);  /* C++ only */
```

General description

The isnan() function tests whether *x* is NaN (not a number).

isnan() is available as a macro. For better performance, the macro form is recommended over the functional form. For C compiles only, to use the functional form, do one of the following:

- Do not include math.h.
- Specify `#undef isnan` after the inclusion of math.h.
- Enclose the call statement in parentheses.

Notes:

1. To use IEEE decimal floating-point, the hardware must have the Decimal Floating-Point Facility installed.
2. This function works in IEEE decimal floating-point format. See "IEEE Decimal Floating-Point" for more information.

Returned value

For IEEE Decimal Floating Point numbers and Binary Floating Point numbers, a non-zero value is returned if x is a NAN. The C++ function template returns true if x is a NAN.

Special behavior in Hex

The isnan() macro returns zero. The C++ function template returns false.

Related information

- "math.h" on page 44

isnormal() — Determines if X is normal

Standards

Standards / Extensions	C or C++	Dependencies
C99 Single UNIX Specification, Version 3 C/C++ DFP C++ TR1 C99	both	z/OS V1R8

Format

```
#define _ISOC99_SOURCE
#include <math.h>

int isnormal(real-floating x);

#define __STDC_WANT_DEC_FP__
#include <math.h>

int isnormal(real-floating x);  /* C only */
int isnormal(decimal-floating x);  /* C only */
bool isnormal(real-floating x);  /* C++ only */
bool isnormal(decimal-floating x);  /* C++ only */

#define _TR1_C99
#include <math.h>

bool isnormal(real-floating x);  /* C++ only */
```

General description

The isnormal() macro or function template determines if its argument value is normal, that is, not zero, infinity, subnormal or a NaN.

Function	Hex	IEEE
isnormal	X	X

Notes:

1. To use IEEE decimal floating-point, the hardware must have the Decimal Floating-Point Facility installed.
2. This function works in IEEE decimal floating-point format. See "IEEE Decimal Floating-Point" for more information.

Returned value

The isnormal() macro returns 1 if the argument value is normal, else returns 0. The C++ template returns true if the argument value is normal, else returns false.

Special behavior in Hex: For normalized numbers, isnormal() returns one. For zero or an unnormalized number, isnormal() returns zero. The C++ function template for normalized numbers, isnormal() returns true. For zero or a number that is not normalized, isnormal() returns false.

Related information

- "math.h" on page 44

__isPosixOn() — Test for POSIX runtime option

Standards

Standards / Extensions	C or C++	Dependencies
z/OS UNIX	both	

Format

```
#include <unistd.h>

int __isPosixOn(void);
```

General description

The __isPosixOn() function returns 1 if the kernel is active and the POSIX runtime option is in effect for the calling process.

Returned value

The __isPosixOn() function returns 1 if the POSIX runtime option is in effect for the calling process and returns 0 otherwise.

If POSIX is in effect, then the kernel is active, although the kernel may be active without POSIX being in effect.

There are no errno values defined.

Related information

- "unistd.h" on page 82

isprint() — Test for printable character classification

The information for this function is included in "isalnum() to isxdigit() — Test integer value" on page 895.

ispunct() — Test for punctuation classification

The information for this function is included in "isalnum() to isxdigit() — Test integer value" on page 895.

isspace() — Test for space character classification

The information for this function is included in "isalnum() to isxdigit() — Test integer value" on page 895.

isunordered() — Determine if either X or Y is unordered

Standards

Standards / Extensions	C or C++	Dependencies
C99 Single UNIX Specification, Version 3 C/C++ DFP C++ TR1 C99	both	z/OS V1R8

Format

```
#define _ISOC99_SOURCE
#include <math.h>

int isunordered(real-floating x, real-floating y);

#define __STDC_WANT_DEC_FP__
#include <math.h>

int isunordered(real-floating x, real-floating y);  /* C only */
int isunordered(decimal-floating x, decimal-floating y);  /* C only */
bool isunordered(real-floating x, real-floating y);  /* C++ only */
bool isunordered(decimal-floating x, decimal-floating y);  /* C++ only */

#define _TR1_C99
#include <math.h>

bool isunordered(real-floating x, real-floating y);  /* C++ only */
```

General description

The isunordered() macro or function template determines if either *x* or *y* is unordered, that is if *x* or *y* is a NaN.

Function	Hex	IEEE
isunordered	X	X

Notes:

1. To use IEEE decimal floating-point, the hardware must have the Decimal Floating-Point Facility installed.
2. This function works in IEEE decimal floating-point format. See "IEEE Decimal Floating-Point" for more information.

Returned value

The isunordered() macro returns 1 if either *x* or *y* is unordered, else returns 0. The C++ function template returns true if either x or y is unordered, else returns false.

Special behavior in hex: The isunordered() macro always returns 0. The C++ function template always returns false.

Related information

- “math.h” on page 44

isupper() — Test for uppercase letter classification

The information for this function is included in “isalnum() to isxdigit() — Test integer value” on page 895.

iswalnum() to iswxdigit() — Test wide integer value

Standards

Standards / Extensions	C or C++	Dependencies
ISO C Amendment XPG4 XPG4.2 C99 Single UNIX Specification, Version 3	both	

Format

```
#include <wctype.h>

int iswalnum(wint_t wc);
int iswalpha(wint_t wc);
int iswcntrl(wint_t wc);
int iswdigit(wint_t wc);
int iswgraph(wint_t wc);
int iswlower(wint_t wc);
int iswprint(wint_t wc);
int iswpunct(wint_t wc);
int iswspace(wint_t wc);
int iswupper(wint_t wc);
int iswxdigit(wint_t wc);
```

General description

The functions listed above, which are all declared in wctype.h, test a given wide integer value. These functions are sensitive to locale. For locale descriptions, see “Internationalization: Locales and Character Sets” in *z/OS XL C/C++ Programming Guide*. Here are descriptions of each function in this group.

iswalnum()
Test for a wide alphanumeric character, as defined in the `alnum` locale source file and in the `alnum` class of the LC_CTYPE category of the current locale.

iswalpha()
Test for a wide alphabetic character, as defined in the `alpha` locale source file and in the `alpha` class of the LC_CTYPE category of the current locale.

iswcntrl()
Test for a wide control character, as defined in the `cntrl` locale source file and in the `cntrl` class of the LC_CTYPE category of the current locale.

iswdigit()
Test for a wide decimal-digit character: 0 through 9, as defined in the `digit` locale source file and in the `digit` class of the LC_CTYPE category of the current locale.

iswgraph()
Test for a wide printing character, not a space. as defined in the `graph` locale source file and in the `graph` class of the LC_CTYPE category of the current locale.

iswlower()
Test for a wide lowercase letter, as defined in the `lower` locale source file and in the `lower` class of the LC_CTYPE category of the current locale.

iswprint()
Test for any wide printing character, as defined in the `print` locale source file and in the `print` class of the LC_CTYPE category of the current locale.

iswpunct()
Test for a wide nonalphanumeric, nonspace character, as defined in the `punct` locale source file and in the `punct` class of the LC_CTYPE category of the current locale.

iswspace()
Test for a wide white space character, as defined in the `space` locale source file and in the `space` class of the LC_CTYPE category of the current locale.

iswupper()
Test for a wide uppercase letter, as defined in the `upper` locale source file and in the `upper` class of the LC_CTYPE category of the current locale.

iswxdigit()
Test for a wide hexadecimal digit 0 through 9, a through f, or A through F, as defined in the `xdigit` locale source file and in the `xdigit` class of the LC_CTYPE category of the current locale.

The behavior of these wide-character function are affected by the LC_CTYPE category of the current locale. The space, uppercase, and lowercase characters can be redefined by their respective class of the LC_CTYPE in the current locale. If you change the category, undefined results can occur.

Returned value

If the wide integer satisfies the test value, these functions return nonzero.

If the wide integer does not satisfy the test value, these functions return 0.

The value for *wc* must be representable as a wide unsigned character. WEOF is a valid input value.

Example

CELEBI06

```
/* CELEBI06

 This example tests for various wide integer values and prints a result.

 */
#include <stdio.h>
#include <wctype.h>

int main(void)
{
   wint_t wc;

   for (wc=0; wc <= 0xFF; wc++) {
      printf("%3d", wc);
      printf(" %#4x ", wc);
      printf("%3s", iswalnum(wc)  ? "AN" : " ");
      printf("%2s", iswalpha(wc)  ? "A"  : " ");
      printf("%2s", iswcntrl(wc)  ? "C"  : " ");
      printf("%2s", iswdigit(wc)  ? "D"  : " ");
      printf("%2s", iswgraph(wc)  ? "G"  : " ");
      printf("%2s", iswlower(wc)  ? "L"  : " ");
      printf(" %c", iswprint(wc)  ? wc   : ' ');
      printf("%3s", iswpunct(wc)  ? "PU" : " ");
      printf("%2s", iswspace(wc)  ? "S"  : " ");
      printf("%3s", iswprint(wc)  ? "PR" : " ");
      printf("%2s", iswupper(wc)  ? "U"  : " ");
      printf("%2s", iswxdigit(wc) ? "X"  : " ");

      putchar('\n');
   }
}
```

Related information

- "wctype.h" on page 86

iswblank() — Test for blank character classification

Standards

Standards / Extensions	C or C++	Dependencies
Language Environment C99 Single UNIX Specification, Version 3 C++ TR1 C99	both	

Format

```
#include <wctype.h>

int iswblank(wint_t wc);
```

C99:

```
#define _ISOC99_SOURCE
#include <wctype.h>

int iswblank(wint_t wc);
```

General description

Tests for a wide blank character.

The space, uppercase, and lowercase characters can be redefined by their respective classes of the LC_CTYPE in the current locale.

For use as a C library function: To avoid infringing on the user's name space, this nonstandard function has two names. One name, __iswblk(), and the other as shown above. The name shown above is exposed only when you use the compiler option LANGLVL(EXTENDED) or define the _EXT feature test macro.

For use as a z/OS UNIX function: Define the _OPEN_SYS feature test macro.

Note: The iswblank() function has a dependency on the level of the Enhanced ASCII Extensions. See "Enhanced ASCII support" on page 2113 for details.

Returned value

If the wide integer satisfies the test value, iswblank() returns nonzero.

If the wide integer does not satisfy the test value, iswblank() returns 0.

The value for *wc* must be representable as a wide unsigned char. WEOF is a valid input value.

The behavior of iswblank() is affected by the LC_CTYPE category of the current locale. If you change the category, undefined results can occur.

Related information

- "wctype.h" on page 86
- "isblank() — Test for blank character classification" on page 905
- "iswalnum() to iswxdigit() — Test wide integer value" on page 920

iswcntrl() — Test for control classification

The information for this function is included in "iswalnum() to iswxdigit() — Test wide integer value" on page 920.

iswctype() — Test for character property

Standards

Standards / Extensions	C or C++	Dependencies
ISO C Amendment XPG4 XPG4.2 C99 Single UNIX Specification, Version 3	both	

Format

```
#include <wctype.h>

int iswctype(wint_t wc, wctype_t wc_prop);
```

General description

Determines whether the wide character *wc* has the property *wc_prop*. If the value of *wc* is neither WEOF nor any value of the wide character that corresponds to a multibyte character, the behavior is undefined. If the value of *wc_prop* is not valid (that is, not obtained by a previous call to wctype(), or *wc_prop* has been invalidated by a subsequent call to setlocale() that has affected category LC_CTYPE), the behavior is undefined.

These twelve strings are reserved for the standard (basic) character classes: alnum, alpha, blank, cntrl, digit, graph, lower, print, punct, space, upper, and xdigit.

The functions are shown below with their equivalent isw*() function:

```
iswctype(wc, wctype("alnum"));  - iswalnum(wc);
iswctype(wc, wctype("alpha"));  - iswalpha(wc);
iswctype(wc, wctype("blank"));  - iswblank(wc);
iswctype(wc, wctype("cntrl"));  - iswcntrl(wc);
iswctype(wc, wctype("digit"));  - iswdigit(wc);
iswctype(wc, wctype("graph"));  - iswgraph(wc);
iswctype(wc, wctype("lower"));  - iswlower(wc);
iswctype(wc, wctype("print"));  - iswprint(wc);
iswctype(wc, wctype("punct"));  - iswpunct(wc);
iswctype(wc, wctype("space"));  - iswspace(wc);
iswctype(wc, wctype("upper"));  - iswupper(wc);
iswctype(wc, wctype("xdigit")); - iswxdigit(wc);
```

Returned value

iswctype() returns nonzero (true) if the wide character *wc* has the property *wc_prop*.

Example

CELEBI07

```
/* CELEBI07

   This example test various wide characters for certain properties and
   prints the result.

 */
#include <stdio.h>
#include <wchar.h>
#include <wctype.h>

int main(void)
{
   int wc;

   for (wc=0; wc <= 0xFF; wc++) {
      printf("%3d", wc);
      printf(" %#4x ", wc);
      printf("%3s", iswctype(wc, wctype("alnum"))  ? "AN" : " ");
      printf("%2s", iswctype(wc, wctype("alpha"))  ? "A"  : " ");
      printf("%2s", iswctype(wc, wctype("cntrl"))  ? "C"  : " ");
      printf("%2s", iswctype(wc, wctype("digit"))  ? "D"  : " ");
      printf("%2s", iswctype(wc, wctype("graph"))  ? "G"  : " ");
      printf("%2s", iswctype(wc, wctype("lower"))  ? "L"  : " ");
      printf(" %c", iswctype(wc, wctype("print"))  ? wc   : ' ');
      printf("%3s", iswctype(wc, wctype("punct"))  ? "PU" : " ");
      printf("%2s", iswctype(wc, wctype("space"))  ? "S"  : " ");
      printf("%3s", iswctype(wc, wctype("print"))  ? "PR" : " ");
      printf("%2s", iswctype(wc, wctype("upper"))  ? "U"  : " ");
      printf("%2s", iswctype(wc, wctype("xdigit")) ? "X"  : " ");

      putchar('\n');
   }
}
```

Related information

- “wctype.h” on page 86
- “wctype() — Obtain handle for character property classification” on page 2056

iswdigit() — Test for hexadecimal-digit classification

The information for this function is included in “iswalnum() to iswxdigit() — Test wide integer value” on page 920.

iswgraph() — Test for graphic classification

The information for this function is included in “iswalnum() to iswxdigit() — Test wide integer value” on page 920.

iswlower() — Test for lowercase

The information for this function is included in “iswalnum() to iswxdigit() — Test wide integer value” on page 920.

iswprint() — Test for printable character classification

The information for this function is included in “iswalnum() to iswxdigit() — Test wide integer value” on page 920.

iswpunct() — Test for punctuation classification

The information for this function is included in “iswalnum() to iswxdigit() — Test wide integer value” on page 920.

iswspace() — Test for space character classification

The information for this function is included in “iswalnum() to iswxdigit() — Test wide integer value” on page 920.

iswupper() — Test for uppercase letter classification

The information for this function is included in “iswalnum() to iswxdigit() — Test wide integer value” on page 920.

iswxdigit() — Test for hexadecimal-digit classification

The information for this function is included in “iswalnum() to iswxdigit() — Test wide integer value” on page 920.

isxdigit() — Test for hexadecimal-digit classification

The information for this function is included in “isalnum() to isxdigit() — Test integer value” on page 895.

itoa() — Convert int into a string

Standards

Standards / Extensions	C or C++	Dependencies
z/OS UNIX	both	z/OS V1R5

Format

```
#define _OPEN_SYS_ITOA_EXT
#include <stdlib.h>

char * itoa(int n, char * buffer, int radix);
```

General description

The itoa() function coverts the integer n into a character string. The string is placed in the buffer passed, which must be large enough to hold the output. The radix values can be OCTAL, DECIMAL, or HEX. When the radix is DECIMAL, itoa() produces the same result as the following statement:

```
(void) sprintf(buffer, "%d", n);
```

with buffer the returned character string. When the radix is OCTAL, itoa() formats integer n into an unsigned octal constant. When the radix is HEX, itoa() formats integer n into an unsigned hexadecimal constant. The hexadecimal value will include lower case abcdef, as necessary.

Returned value

String pointer (same as buffer) will be returned. When passed a non-valid radix argument, function will return NULL and set errno to EINVAL.

Portability considerations

This is a non-standard function. Even though the prototype given is commonly used by compilers on other platforms, there is no guarantee that this function will behave the same on all platforms, in all cases. You can use this function to help port applications from other platforms, but you should avoid using it when writing new applications, in order to ensure maximum portability.

Example

CELEBI12

```
/* CELEBI12

   This example reads an int and formats it to decimal, unsigned
   octal, and unsigned hexadecimal constants converted to a
   character string.

*/

#define _OPEN_SYS_ITOA_EXT
#include <stdio.h>
#include <stdlib.h>

int main ()
{
   int i;
   char buffer [sizeof(int)*8+1];
   printf ("Enter a number: ");
   if (scanf ("%d",&i) == 1) {
      itoa (i,buffer,DECIMAL);
      printf ("decimal: %s\n",buffer);
      itoa (i,buffer,HEX);
      printf ("hexadecimal: %s\n",buffer);
      itoa (i,buffer,OCTAL);
      printf ("octal: %s\n",buffer);
   }
   return 0;
}
```

Output

If the input is 1234, then the output should be:

```
decimal: 1234
hexadecimal: 4d2
octal: 2322
```

Related information

- “stdlib.h” on page 70
- “lltoa() — Convert long long into a string” on page 979
- “ltoa() — Convert long into a string” on page 1027
- “ulltoa() — Convert unsigned long long into a string” on page 1943
- “ultoa() — Convert unsigned long into a string” on page 1945
- “utoa() — Convert unsigned int into a string” on page 1972

JoinWorkUnit() — Join a WLM work unit

Standards

Standards / Extensions	C or C++	Dependencies
z/OS UNIX	both	

Format

```
#include <sys/__wlm.h>

int JoinWorkUnit(wlmetok_t *enclavetoken);
```

General description

The JoinWorkUnit function provides the ability for an application to join a WLM work unit.

******enclavetoken*
: Points to a work unit enclave token that was returned from a call to either CreateWorkUnit() or ContinueWorkUnit().

Returned value

If successful, JoinWorkUnit() returns 0.

If unsuccessful, JoinWorkUnit() returns -1 and sets errno to one of the following values:

Error Code
Description

EFAULT
An argument of this function contained an address that was not accessible to the caller.

EINVAL
An argument of this function contained an incorrect value.

EMVSSAF2ERR
An error occurred in the security product.

EMVSWLMERROR
The WLM join enclave failed. Use __errno2() to obtain the WLM service reason code for the failure.

EPERM
The calling thread's address space is not permitted to the BPX.WLMSERVER Facility class. The caller's address space must be permitted to the BPX.WLMSERVER Facility class if it is defined. If BPX.WLMSERVER is not defined, the calling process is not defined as a superuser (UID=0).

Related information

- “sys/__wlm.h” on page 77
- “CheckSchEnv() — Check WLM scheduling environment” on page 271
- “ConnectServer() — Connect to WLM as a server manager” on page 313
- “ConnectWorkMgr() — Connect to WLM as a work manager” on page 315
- “ContinueWorkUnit() — Continue WLM work unit” on page 323
- “CreateWorkUnit() — Create WLM work unit” on page 344
- “DeleteWorkUnit() — Delete a WLM work unit” on page 378
- “DisconnectServer() — Disconnect from WLM server” on page 383
- “LeaveWorkUnit() — Leave a WLM work unit” on page 945
- “QueryMetrics() — Query WLM system information” on page 1368
- “QuerySchEnv() — Query WLM scheduling environment” on page 1369

jrand48() — Pseudo-random number generator

Standards

Standards / Extensions	C or C++	Dependencies
XPG4 XPG4.2 Single UNIX Specification, Version 3	both	

Format

```
#define _XOPEN_SOURCE
#include <stdlib.h>

long int jrand48(unsigned short int x16v[3]);
```

General description

The drand48(), erand48(), jrand48(), lrand48(), mrand48() and nrand48() functions generate uniformly distributed pseudo-random numbers using a linear congruential algorithm and 48-bit integer arithmetic.

The functions drand48() and erand48() return nonnegative, double-precision, floating-point values, uniformly distributed over the interval [0.0,1.0).

The functions lrand48() and nrand48() return nonnegative, long integers, uniformly distributed over the interval [0,2**31).

The functions mrand48() and jrand48() return signed long integers, uniformly distributed over the interval [-2**31,2**31).

The jrand48() function generates the next 48-bit integer value in a sequence of 48-bit integer values, X(i), according to the linear congruential formula:

```
X(n+1) = (aX(n) + c)mod(2**48)     n>=0
```

The jrand48() function uses storage provided by the argument array, *x16v*[3], to save the most recent 48-bit integer value in the sequence, X(i). The jrand48() function uses *x16v*[0] for the low-order (rightmost) 16 bits, *x16v*[1] for the middle-order 16 bits, and *x16v*[2] for the high-order 16 bits of this value.

The initial values of a, and c are:

```
a   = 5deece66d (base 16)
c   = b         (base 16)
```

The values a and c, may be changed by calling the lcong48() function. The initial values of a and c are restored if either the seed48() or srand48() function is called.

Special behavior for z/OS UNIX Services: You can make the jrand48() function and other functions in the drand48 family thread-specific by setting the environment variable _RAND48 to the value THREAD before calling any function in the drand48 family.

If you do not request thread-specific behavior for the drand48 family, C/370 serializes access to the storage for X(n), a and c by functions in the drand48 family when they are called by a multithreaded application.

If thread-specific behavior is requested and the jrand48() function is called from thread t, the jrand48() function generates the next 48-bit integer value in a sequence of 48-bit integer values, X(t,i), for the thread according to the linear congruential formula:

```
X(t,n+1) = (a(t)X(t,n) + c(t))mod(2**48)     n>=0
```

The jrand48() function uses storage provided by the argument array, *x16v*[3], to save the most recent 48-bit integer value in the sequence, X(t,i). The jrand48() function uses *x16v*[0] for the low-order (rightmost) 16 bits, *x16v*[1] for the middle-order 16 bits, and *x16v*[2] for the high-order 16 bits of this value.

The initial values of a(t) and c(t) on the thread t are:

```
a(t)   = 5deece66d (base 16)
c(t)   = b         (base 16)
```

The values a(t) and c(t) may be changed by calling the lcong48() function from the thread t. The initial values of a(t) and c(t) are restored if either the seed48() or srand48() function is called from the thread.

Returned value

jrand48() saves the generated 48-bit value, X(n+1), in storage provided by the argument array, *x16v*[3]. jrand48() transforms the generated 48-bit value to a signed long integer value on the interval [-2**31,2**31) and returns this transformed value.

Special behavior for z/OS UNIX Services: If thread-specific behavior is requested for the drand48 family and the jrand48() function is called on thread t, the jrand48() function saves the generated 48-bit value, X(t,n+1), in storage provided by the argument array, *x16v*[3]. The jrand48() function transforms the generated 48-bit value to a signed long integer value on the interval [-2**31,2**31) and returns this transformed value.

Related information

- “stdlib.h” on page 70
- “drand48() — Pseudo-random number generator” on page 402
- “erand48() — Pseudo-random number generator” on page 426
- “lcong48() — Pseudo-random number initializer” on page 939
- “lrand48() — Pseudo-random number generator” on page 1010
- “mrand48() — Pseudo-random number generator” on page 1101
- “nrand48() — Pseudo-random number generator” on page 1148
- “seed48() — Pseudo-random number initializer” on page 1467
- “srand48() — Pseudo-random number initializer” on page 1718

j0(), j1(), jn() — Bessel functions of the first kind

Standards

Standards / Extensions	C or C++	Dependencies
SAA XPG4 XPG4.2 Single UNIX Specification, Version 3	both	

Format

```
#include <math.h>

double j0(double x);

double j1(double x);

double jn(int n, double x);
```

Compiler option: LANGLVL(SAA), LANGLVL(SAAL2), or LANGLVL(EXTENDED)

General description

The j0(), j1(), and jn() functions are Bessel functions of the *first kind*, for orders 0, 1, and *n*, respectively. Bessel functions are solutions to certain types of differential equations. The argument *x* must be positive. The argument *n* should be greater than or equal to 0. If *n* is less than 0, there will be a negative exponent in the result.

Note: This function works in both IEEE Binary Floating-Point and hexadecimal floating-point formats. See "IEEE binary floating-point" on page 94 for more information about IEEE Binary Floating-Point.

Returned value

If successful, the calculated value is returned.

For j0(), j1(), y0(), or y1(), if the absolute value of *x* is too large, the function sets errno to ERANGE to indicate a value that is out of range, and returns 0.

Special behavior for IEEE: If *x* is negative, y0(), y1(), and yn() return the value NaNQ. If *x* is 0, y0(), y1(), and yn() return the value -HUGE_VAL. In all cases, errno remains unchanged.

Example

CELEBJ01

```
/* CELEBJ01

   This example computes y to be the order 0 Bessel function of
   the first kind for x, and z to be the order 3 Bessel function
   of the second kind for x.

 */
#include <math.h>
#include <stdio.h>
```

```
int main(void)
{
   double x, y, z;
   x = 4.27;

   y = j0(x);      /* y = -0.3660 is the order 0 bessel */
                    /* function of the first kind for x  */
   z = yn(3,x);    /* z = -0.0875 is the order 3 bessel */
                    /* function of the second kind for x */
   printf("x = %f\n  y = %f\n  z = %f\n", x, y, z);
}
```

Related information

- "math.h" on page 44
- "erf(), erfc(), erff(), erfl(), erfcf(), erfcl() — Calculate error and complementary error functions" on page 428
- "gamma() — Calculate gamma function" on page 682
- "y0(), y1(), yn() — Bessel functions of the second kind" on page 2098

kill() — Send a signal to a process

Standards

Standards / Extensions	C or C++	Dependencies
POSIX.1 XPG4 XPG4.2 Single UNIX Specification, Version 3	both	POSIX(ON)

Format

```
#define _POSIX_SOURCE
#include <signal.h>

int kill(pid_t pid, int sig);
```

General description

Sends a signal to a process or process group. A process has permission to send a signal if the real or effective user ID of the sender is the same as the real or effective user ID of the intended recipient. A process can also send signals if it has appropriate privileges. If _POSIX_SAVED_IDS is defined in the unistd.h header file, the saved set user ID of the intended recipient is checked instead of its effective user ID.

Regardless of user ID, a process can always send a SIGCONT signal to a process that is a member of the same session (same session ID) as the sender.

You can use either signal() or sigaction() to specify how a signal will be handled when kill() is invoked.

A process can use kill() to send a signal to itself. If the signal is not blocked or ignored, at least one pending unblocked signal is delivered to the sender before kill() returns. If there are no other pending unblocked signals, the delivered signal is *sig*.

pid can be used to specify these processes:

pid_t *pid*;
Specifies the processes that the caller wants to send a signal to:

- If *pid* is greater than 0, kill() sends its signal to the process whose ID is equal to *pid*.
- If *pid* is equal to 0, kill() sends its signal to all processes whose process group ID is equal to that of the sender, except for those that the sender does not have appropriate privileges to send a signal to.
- If *pid* is -1, kill() returns -1.
- **Special behavior for XPG4.2:** If *pid* is -1, kill() sends the signal, *sig*, to all processes, except for those to which the sender does not have appropriate privileges to send a signal.
- If *pid* is less than -1, kill() sends its signal to all processes whose process group ID is equal to the absolute value of *pid*, except for those that the sender does not have appropriate privileges to send a signal to.

int *sig*;
The signal that should be sent to the processes specified by *pid*. (For a list of signals, see Table 48 on page 1612.) This must be 0 or one of the signals defined in the signal.h header file. If *sig* is 0, kill() performs error checking but does not send a signal. You can code *sig* as 0 to check whether the *pid* argument is valid.

This function is supported only in a POSIX program. You can use it to pass SIGIOERR.

Usage notes

The use of the `SIGTHSTOP` and `SIGTHCONT` signal is not supported with this function.

Returned value

kill() returns 0 if it has permission to send *sig* to any of the processes specified by *pid*.

If kill() fails to send a signal, it returns -1 and sets errno to one of the following values:

Error Code
Description

EINVAL
The value of *sig* is incorrect or is not the number of a supported signal.

EPERM
The caller does not have permission to send the signal to any process specified by *pid*.

ESRCH
There are no processes or process groups corresponding to *pid*.

Example

CELEBK01

```
/* CELEBK01 */
#define _POSIX_SOURCE
#include <signal.h>
#include <stdio.h>
#include <sys/types.h>
```

```
#include <unistd.h>
#include <sys/wait.h> /*FIX: used to be <wait.h>*/

main() {
  sigset_t sigset;
  int    p[2], status;
  char   c='z';
  pid_t  pid;

  if (pipe(p) != 0)
    perror("pipe() error");
  else {
    if ((pid = fork()) == 0) {
      sigemptyset(&sigset);
      puts("child is letting parent know he's ready for signal");
      write(p[1], &c, 1);
      puts("child is waiting for signal");
      sigsuspend(&sigset);
      exit(0);
    }

    puts("parent is waiting for child to say he's ready for signal");
    read(p[0], &c, 1);
    puts("child has told parent he's ready for signal");

    kill(pid, SIGTERM);

    wait(&status);
    if (WIFSIGNALED(status))
      if (WTERMSIG(status) == SIGTERM)
        puts("child was ended with a SIGTERM");
      else
        printf("child was ended with a %d signal\n", WTERMSIG(status));
    else puts("child was not ended with a signal");

    close(p[0]);
    close(p[1]);
  }
}
```

Output

```
parent is waiting for child to say he's ready for signal
child is letting parent know he's ready for signal
child is waiting for signal
child has told parent he's ready for signal
child was ended with a SIGTERM
```

Related information

- "signal.h" on page 63
- "unistd.h" on page 82
- "bsd_signal() — BSD version of signal()" on page 219
- "getpid() — Get the process ID" on page 752
- "killpg() — Send a signal to a process group" on page 934
- "pthread_kill() — Send a signal to a thread" on page 1283
- "raise() — Raise signal" on page 1371
- "setsid() — Create session, set process group ID" on page 1576
- "sigaction() — Examine or change a signal action" on page 1611
- "sighold() — Add a signal to a thread" on page 1636
- "sigignore() — Set disposition to ignore a signal" on page 1637
- "signal() — Handle interrupts" on page 1643
- "sigprocmask() — Examine or change a thread" on page 1651
- "sigrelse() — Remove a signal from a thread" on page 1656
- "sigset() — Change a signal action or a thread" on page 1656

killpg() — Send a signal to a process group

Standards

Standards / Extensions	C or C++	Dependencies
XPG4.2 Single UNIX Specification, Version 3	both	POSIX(ON)

Format

```
#define _XOPEN_SOURCE_EXTENDED 1
#include <signal.h>

int killpg(pid_t pgrp, int sig);
```

General description

The killpg() function sends a signal to a process group.

A process has permission to send a signal if the real or effective user ID of the sender is the same as the real or effective user ID of the intended recipient. A process can also send signals if it has appropriate privileges. If _POSIX_SAVED_IDS is defined in the <unistd.h> include file, the saved set user ID of the intended recipient is checked instead of its effective user ID.

Regardless of user ID, a process can always send a SIGCONT signal to a process group that is a member of the same session (same session ID) as the sender.

pid_t *pgrp*;
Specifies the process group that the caller wants to send a signal to:

- If *pgrp* is greater than one, killpg() sends the signal, *sig*, to the process whose process group ID is equal to *pgrp* and which the sender has appropriate privileges to send a signal.
- If *pgrp* is equal to or less than one, killpg() returns a -1 and sets errno to EINVAL.

int *sig*;
The signal that should be sent to the processes specified by *pid*. (For a list of signals, see Table 48 on page 1612.) This must be zero, or one of the signals defined in the <signal.h> include file. If *sig* is zero, killpg() performs error checking but doesn't really send a signal. You can code *sig* as zero to check whether the *pid* argument is valid.

This function is supported only in a POSIX program.

Usage notes

The use of the SIGTHSTOP and SIGTHCONT signal is not supported with this function.

Returned value

If successful, killpg() returns 0 if it has permission to send *sig* to any of the processes in the process group ID specified by *pgrp*.

If unsuccessful, killpg() returns -1 and sets errno to one of the following values:

Error Code
Description

EINVAL
The value of *sig* is incorrect or is not the number of a supported signal, or the value of *pgrp* is less than or equal to one.

EPERM
The caller does not have permission to send the signal to any process in the process group ID specified by *pgrp*.

ESRCH
There are no process groups corresponding to *pgrp*.

Related information

- “signal.h” on page 63
- “bsd_signal() — BSD version of signal()” on page 219
- “getpgid() — Get process group ID” on page 750
- “getpid() — Get the process ID” on page 752
- “kill() — Send a signal to a process” on page 931
- “raise() — Raise signal” on page 1371
- “setsid() — Create session, set process group ID” on page 1576
- “sigaction() — Examine or change a signal action” on page 1611
- “sighold() — Add a signal to a thread” on page 1636
- “sigignore() — Set disposition to ignore a signal” on page 1637
- “signal() — Handle interrupts” on page 1643
- “sigprocmask() — Examine or change a thread” on page 1651
- “sigrelse() — Remove a signal from a thread” on page 1656
- “sigset() — Change a signal action or a thread” on page 1656

labs() — Calculate long absolute value

Standards

Standards / Extensions	C or C++	Dependencies
ISO C XPG4 XPG4.2 C99 Single UNIX Specification, Version 3	both	

Format

```
#include <stdlib.h>

long int labs(long int n);
```

General description

Calculates the absolute value of its long integer argument *n*. The result is undefined when the argument is equal to LONG_MIN, the smallest available long integer (-2 147 483 648). The value LONG_MIN is defined in the limits.h header file.

Returned value

Returns the absolute value of the long integer argument *n*.

Example

CELEBL01

```
/* CELEBL01

   This example computes y as the absolute value of
   the long integer -41567.

 */
#include <stdlib.h>
#include <stdio.h>

int main(void)
{
   long x, y;

   x = -41567L;
   y = labs(x);

   printf("The absolute value of %ld is %ld\n", x, y);
}
```

Output

```
The absolute value of -41567 is 41567
```

Related information

- "stdlib.h" on page 70
- "abs(), absf(), absl() — Calculate integer absolute value" on page 103
- "fabs(), fabsf(), fabsl() — Calculate floating-point absolute value" on page 460

__lchattr() — Change the attributes of a file or directory when they point to a symbolic or external link

Standards

Standards / Extensions	C or C++	Dependencies
z/OS UNIX	both	z/OS V1R5

Format

```
#define _OPEN_SYS_FILE_EXT 1
#include <sys/stat.h>

int __lchattr (char *pathname, attrib_t *attributes, int attributes_len);
```

General description

The __lchattr() function modifies the attributes that are associated with a file. The pathname specifies a symbolic or external link (a pointer to another file, directory, or data set).

The __lchattr() service changes the attributes of the symbolic link itself, provided the attributes requested can apply to a symbolic link. Only the owner and security label can be changed for a symbolic link, all other attributes do not apply and will be ignored.

The attributes argument is the address of an attrib_t structure which is used to identify the attributes to be modified and the new values desired. The attrib_t type is an f_attributes structure as defined in <sys/stat.h> for use with the __lchattr() function. For proper behavior, the user should ensure that this structure has been initialized to zeros before it is populated. The f_attributes structure is defined as indicated in Table 20 on page 261.

Returned value

If successful, __lchattr() returns 0.

If unsuccessful, __lchattr() returns -1 and sets errno to one of the following values:

EACCES
The calling process did not have appropriate permissions. Possible reasons include:

- The calling process was attempting to set access time or modification time to current time, and the effective UID of the calling process does not match the owner of the file; the process does not have write permission for the file; or the process does not have appropriate privileges.
- The calling process was attempting to truncate the file, and it does not have write permission for the file.

EFBIG
The calling process was attempting to change the size of a file, but the specified length is greater than the maximum file size limit for the process.

EINVAL
The attributes structure containing the requested changes is not valid.

ELOOP
A loop exists in symbolic links that were encountered during resolution of the pathname argument. This error is issued if more than 24 symbolic links are detected in the resolution of pathname.

ENAMETOOLONG
pathname is longer than 1023 characters, or a component of the pathname is longer than 255 characters (Filename truncation is not supported).

ENOENT
No file named pathname was found.

ENOTDIR
Some component of pathname is not a directory.

EPERM
The operation is not permitted for one of the following reasons:

- The calling process was attempting to change the mode or the file format, but the effective UID of the calling process does not match the owner of the file, and the calling process does not have appropriate privileges.
- The calling process was attempting to change the owner, but it does not have appropriate privileges.
- The calling process was attempting to change the general attribute bits, but it does not have write permission for the file.
- The calling process was attempting to set a time value (not current time), but the effective user ID does not match the owner of the file, and it does not have appropriate privileges.

- The calling process was attempting to set the change time or reference time to current time, but it does not have write permission for the file.
- The calling process was attempting to change auditing flags, but the effective UID of the calling process does not match the owner of the file, and the calling process does not have appropriate privileges.
- The calling process was attempting to change the Security Auditor's auditing flags, but the user does not have auditor authority.
- Attributes indicate that the security label is to be set, and one or more of the following conditions applies:
  - The calling process does not have RACF SPECIAL authorization and appropriate privileges.
  - The security label currently associated with the file is already set.

EROFS
pathname specifies a file that is on a read-only file system.

Related information

- "sys/stat.h" on page 75
- "__fchattr() — Change the attributes of a file or directory by file descriptor" on page 465
- "__chattr() — Change the attributes of a file or directory" on page 261

lchown() — Change owner and group of a file

Standards

Standards / Extensions	C or C++	Dependencies
XPG4.2 Single UNIX Specification, Version 3	both	

Format

```
#define _XOPEN_SOURCE_EXTENDED 1
#include <unistd.h>

int lchown(const char *path, uid_t owner, gid_t group);
```

General description

The lchown() function has the same effect as chown() except in the case where the named file is a symbolic link. In this case lchown() changes the ownership of the symbolic link file itself, while chown() changes the ownership of the file or directory to which the symbolic link refers.

Returned value

If successful, lchown() returns 0.

If unsuccessful, lchown() returns -1 and sets errno to one of the following values:

Error Code
Description

EACCES
Search permission is denied on a component of the path prefix of path.

EINVAL
The owner or group id is not a value supported by the implementation.

ELOOP
Too many symbolic links were encountered in resolving *path*

ENAMETOOLONG
The length of a pathname exceeds **PATH_MAX** or a pathname component is longer than **NAME_MAX**.

ENOENT
A component of *path* does not name an existing file or *path* is an empty string.

ENOTDIR
A component of the path prefix of *path* is not a directory.

EOPNOTSUPP
The *path* argument names a symbolic link and the implementation does not support setting the owner or group of a symbolic link.

EPERM
The effective user ID does not match the owner of the file and the process does not have appropriate privileges.

EROFS
The file resides on a read-only file system.

The lchown() function may fail if:

Error Code
Description

EINTR
A signal was caught during execution of the function.

EIO An I/O error occurred while reading or writing to the file system.

ENAMETOOLONG
Pathname resolution of a symbolic link produced an intermediate result whose length exceeds **PATH_MAX**.

Related information

- “unistd.h” on page 82
- “chown() — Change the owner or group of a file or directory” on page 275
- “symlink() — Create a symbolic link to a path name” on page 1795

lcong48() — Pseudo-random number initializer

Standards

Standards / Extensions	C or C++	Dependencies
XPG4 XPG4.2 Single UNIX Specification, Version 3	both	

Format

```
#define _XOPEN_SOURCE
#include <stdlib.h>

void lcong48(unsigned short int param[7]);
```

General description

The drand48(), erand48(), jrand48(), lrand48(), mrand48() and nrand48() functions generate uniformly distributed pseudo-random numbers using a linear congruential algorithm and 48-bit integer arithmetic.

The lcong48(), seed48(), and srand48() functions are initialization functions, one of which should be invoked before either the drand48(), lrand48() or mrand48() function is called.

The drand48(), lrand48() and mrand48() functions generate a sequence of 48-bit integer values, X(i), according to the linear congruential formula:

```
X(n+1) = (aX(n) + c)mod(2**48)     n>=0
```

The initial values of X, a, and c are:

```
X(0)= 1
a   = 5deece66d (base 16)
c   = b         (base 16)
```

C/370 provides storage to save the most recent 48-bit integer value of the sequence, X(i). This storage is shared by the drand48(), lrand48() and mrand48() functions. The lcong48() function is used to reinitialize the most recent 48-bit value in this storage. The lcong48() function replaces the low-order (rightmost) 16 bits of this storage with *param*[0], the middle-order 16 bits with *param*[1], and the high-order 16 bits with *param*[2].

The values a and c, may also be changed by calling the lcong48() function. The lcong48() function replaces the low-order (rightmost) 16 bits of a with *param*[3], the middle-order 16 bits with *param*[4], and the high-order 16 bits with *param*[5]. The lcong48() function replaces c with *param*[6].

Special behavior for z/OS UNIX Services: You can make the lcong48() function and other functions in the drand48 family thread-specific by setting the environment variable _RAND48 to the value THREAD before calling any function in the drand48 family.

If you do not request thread-specific behavior for the drand48 family, C/370 serializes access to the storage for X(n), a and c by functions in the drand48 family when they are called by a multithreaded application.

If thread-specific behavior is requested, calls to the drand48(), lrand48() and mrand48() functions from thread t generate a sequence of 48-bit integer values, X(t,i), according to the linear congruential formula:

```
X(t,n+1) = (a(t)X(t,n) + c(t))mod(2**48)     n>=0
```

C/370 provides thread-specific storage to save the most recent 48-bit integer value of the sequence, X(t,i). When the lcong48() function is called from thread t, it reinitializes the most recent 48-bit value in this storage. The lcong48() function replaces the low-order (rightmost) 16 bits of this storage with *param*[0], the middle-order 16 bits with *param*[1], and the high-order 16 bits with *param*[2].

The lcong48() function may also be used to change values of a(t) and c(t) for the thread t. The lcong48() function replaces the low-order (rightmost) 16 bits of a(t) with *param*[3], the middle-order 16 bits with *param*[4], and the high-order 16 bits with *param*[5]. The lcong48() function replaces c(t) with *param*[6].

Returned value

After lcong48() has used values from the argument array, *param*[7], to change the values of a and c and to reinitialized storage for the most recent 48-bit integer value in the sequence, X(i), it returns.

Special behavior for z/OS UNIX Services: If thread-specific behavior is requested for the drand48 family and lcong48() is called on thread t, it uses the argument array, *param*[7], to change the values of a(t) and c(t) and to reinitialize storage for the most recent 48-bit integer value in the sequence, X(t,i), for the thread. Then it returns.

Related information

- “stdlib.h” on page 70
- “drand48() — Pseudo-random number generator” on page 402
- “erand48() — Pseudo-random number generator” on page 426
- “jrand48() — Pseudo-random number generator” on page 928
- “lrand48() — Pseudo-random number generator” on page 1010
- “mrand48() — Pseudo-random number generator” on page 1101
- “nrand48() — Pseudo-random number generator” on page 1148
- “seed48() — Pseudo-random number initializer” on page 1467
- “srand48() — Pseudo-random number initializer” on page 1718

ldexp(), ldexpf(), ldexpl() — Multiply by a power of two

Standards

Standards / Extensions	C or C++	Dependencies
ISO C POSIX.1 XPG4 XPG4.2 ISO/ANSI C++ C99 Single UNIX Specification, Version 3 C++ TR1 C99	both	

Format

```
#include <math.h>

double ldexp(double x, int exp);
float ldexp(float x, int exp);               /* C++ only */
long double ldexp(long double x, int exp);   /* C++ only */
float ldexpf(float x, int exp);
long double ldexpl(long double x, int exp);
```

General description

Calculates the value of $x*(2^{exp})$.

Restriction: The ldexpf() and ldexpl() functions do not support the _FP_MODE_VARIABLE feature test macro.

Returned value

Returns the calculated value.

Otherwise, if the correct calculated value is outside the range of representable values, ±HUGE_VAL is returned, according to the sign of the value. The value ERANGE is stored in errno to indicate that the result was out of range.

Special behavior for XPG4.2:

Error Code
Description

ERANGE
The result underflowed. ldexp() returns 0.0.

Example

CELEBL02

```
/* CELEBL02

   This example computes  y = 1.5*&lpar.2**5&rpar.

 */
#include <math.h>
#include <stdio.h>

int main(void)
{
   double x, y;
   int p;

   x = 1.5;
   p = 5;
   y = ldexp(x,p);

   printf("%lf times 2 to the power of %d is %Lf\n", x, p, y);
}
```

Output

```
1.500000 times 2 to the power of 5 is 48.000000
```

Related information

- “math.h” on page 44
- “frexp(), frexpf(), frexpl() — Extract mantissa and exponent of the floating-point value” on page 623
- “modf(), modff(), modfl() — Extract fractional and integral parts of floating-point value” on page 1086

ldexpd32(), ldexpd64(), ldexpd128() — Multiply by a power of ten

Standards

Standards / Extensions	C or C++	Dependencies
C/C++ DFP	both	z/OS V1.8

Format

```
#define __STDC_WANT_DEC_FP__
#include <math.h>

_Decimal32  ldexpd32(_Decimal32 x, int exp);
_Decimal64  ldexpd64(_Decimal64 x, int exp);
_Decimal128 ldexpd128(_Decimal128 x, int exp);
```

```
_Decimal32  ldexp(_Decimal32 x, int exp);   /* C++ only */
_Decimal64  ldexp(_Decimal64 x, int exp);   /* C++ only */
_Decimal128 ldexp(_Decimal128 x, int exp);  /* C++ only */
```

General description

Calculates the value of $x*10^{exp}$.

Notes:

1. To use IEEE decimal floating-point, the hardware must have the Decimal Floating-Point Facility installed.
2. These functions work in IEEE decimal floating-point format. See "IEEE Decimal Floating-Point" for more information.

Returned value

Returns the calculated value.

Otherwise, if the correct calculated value is outside the range of representable values, ±HUGE_VAL_D32, ±HUGE_VAL_D64, or ±HUGE_VAL_D128 is returned, according to the sign of the value. The value ERANGE is stored in errno to indicate that the result was out of range.

Example

```
/* CELEBL19

   This example illustrates the ldexpd32() function.

   This example computes  y = 1.5*10**5

*/

#define  __STDC_WANT_DEC_FP__
#include <math.h>
#include <stdio.h>

int main(void)
{
  _Decimal32 x, y;
  int p;

  x = 1.5DF;
  p = 5;
  y = ldexpd32(x, p);

  printf("%Hf times 10 to the power of %d is %Hf\n", x, p, y);
}
```

Related information

- “math.h” on page 44
- “frexpd32(), frexpd64(), frexpd128() — Extract mantissa and exponent of the decimal floating-point value” on page 624
- “ldexp(), ldexpf(), ldexpl() — Multiply by a power of two” on page 941
- “modfd32(), modfd64(), modfd128() — Extract fractional and integral parts of decimal floating-point value” on page 1087

ldiv() — Compute quotient and remainder of integral division

Standards

Standards / Extensions	C or C++	Dependencies
ISO C XPG4 XPG4.2 C99 Single UNIX Specification, Version 3	both	

Format

```
#include <stdlib.h>

ldiv_t ldiv(long int numerator, long int denominator);
```

General description

Calculates the quotient and remainder of the division of *numerator* by *denominator*.

Returned value

Returns a structure of type `ldiv_t`, containing both the quotient `long int quot` and the remainder `long int rem`.

If the value cannot be represented, the returned value is undefined. If *denominator* is 0, a divide by 0 exception is raised.

Example

CELEBL03

```
/* CELEBL03

   This example uses the &ldiv. function to calculate the
   quotients and remainders for a set of two dividends and two
   divisors.

 */
#include <stdio.h>
#include <stdlib.h>

int main(void)
{
   long int num[2] = {45,-45};
   long int den[2] = {7,-7};
   ldiv_t ans;   /* ldiv_t is a struct type containing two long ints:
                    'quot' stores quotient; 'rem' stores remainder */
   short i,j;

   printf("Results of long division:\n");
   for (i = 0; i < 2; i++)
      for (j = 0; j < 2; j++)
      {
         ans = ldiv(num[i], den[j]);
         printf("Dividend: %6ld  Divisor: %6ld", num[i], den[j]);
         printf("  Quotient: %6ld  Remainder: %6ld\n", ans.quot, ans.rem);
      }
}
```

Output

```
Results of long division:
Dividend:  45  Divisor:   7  Quotient:   6  Remainder:   3
Dividend:  45  Divisor:  -7  Quotient:  -6  Remainder:   3
Dividend: -45  Divisor:   7  Quotient:  -6  Remainder:  -3
Dividend: -45  Divisor:  -7  Quotient:   6  Remainder:  -3
```

Related information

- “stdlib.h” on page 70
- “div() — Calculate quotient and remainder” on page 384

LeaveWorkUnit() — Leave a WLM work unit

Standards

Standards / Extensions	C or C++	Dependencies
z/OS UNIX	both	

Format

```
#include <sys/__wlm.h>

int LeaveWorkUnit(wlmetok_t *enclavetoken);
```

General description

The LeaveWorkUnit() function provides the ability for an application to leave a WLM work unit.

**enclavetoken*
Points to a work unit enclave token that was returned from a call to CreateWorkUnit() or ContinueWorkUnit().

Returned value

If successful, LeaveWorkUnit() returns 0.

If unsuccessful, LeaveWorkUnit() returns -1 and sets errno to one of the following values:

Error Code
Description

EFAULT
An argument of this function contained an address that was not accessible to the caller.

EINVAL
An argument of this function contained an incorrect value.

EMVSSAF2ERR
An error occurred in the security product.

EMVSWLMERROR
The WLM leave enclave failed. Use __errno2() to obtain the WLM service reason code for the failure.

EPERM
The calling thread's address space is not permitted to the BPX.WLMSERVER Facility class. The caller's address space must be

permitted to the BPX.WLMSERVER Facility class if it is defined. If BPX.WLMSERVER is not defined, the calling process is not defined as a superuser (UID=0).

Related information

- "sys/__wlm.h" on page 77
- "CheckSchEnv() — Check WLM scheduling environment" on page 271
- "ConnectServer() — Connect to WLM as a server manager" on page 313
- "ConnectWorkMgr() — Connect to WLM as a work manager" on page 315
- "ContinueWorkUnit() — Continue WLM work unit" on page 323
- "CreateWorkUnit() — Create WLM work unit" on page 344
- "DeleteWorkUnit() — Delete a WLM work unit" on page 378
- "DisconnectServer() — Disconnect from WLM server" on page 383
- "JoinWorkUnit() — Join a WLM work unit" on page 927
- "QueryMetrics() — Query WLM system information" on page 1368
- "QuerySchEnv() — Query WLM scheduling environment" on page 1369

__le_ceegtjs() — Retrieve the value of an exported JCL symbol

Standards

Standards / Extensions	C or C++	Dependencies
Language Environment	both	AMODE 64

Format

```
#include <__le_api.h>

void __le_ceegtjs(_INT4 * function_code,
                  _VSTRING * symbol_name,
                  _CHAR255 * symbol_value,
                  _INT4 * value_length,
                  _FEEDBACK * fc);
```

General description

The __le_ceegtjs() function retrieves and returns to the caller the symbol value and length of the requested exported JCL symbol.

Parameter
Description

function_code
A fullword integer containing the function code of the following value:

1 Retrieve the value and its associate length of an exported JCL symbol.

symbol_name
A halfword length-prefixed character string(VSTRING), representing the name of an exported JCL symbol to be retrieved.

symbol_value
A 255-byte fixed-length string. On return from this service, the *symbol_value* contains the value of the exported JCL symbol. If the length of the exported JCL symbol is shorter than 255 characters, the returned string is padded with blanks.

value_length
A fullword integer containing the length of the value of the specified JCL symbol.

fc A 16-byte Feedback Code indicating the results of this function.

Table 34. Feedback Codes for __le_ceegtjs()

Code	Severity	Message Number	Message Text
CEE000	0	- -	The function completed successfully.
CEE3L9	0	3753	The input symbol cannot be found in the current job step.
CEE3LA	3	3754	Incorrect parameters detected.
CEE3QS	1	3932	The system service failed with return code *return_code* and reason code *reason_code*.

Usage notes

1. Lower case characters in the *symbol_name* will be converted to upper case by the __le_ceegtjs function.
2. For more information about JCL symbols and their usage, see "Using System Symbols and JCL symbols" in *z/OS MVS JCL Reference*.

Example

CELEBL31

```
/* CELEBL31

   This example retrieves the value of an exported JCL symbol.

*/
#include <stdio.h>
#include <string.h>
#include <__le_api.h>

int main()
{
   _FEEDBACK fc;
   _INT4 funcode;
   _CHAR255 symvalue;
   _VSTRING symname;
   _INT4 valuelen;
   char *symbol="SYM1";

   /* Setting the function code */
   funcode=1;

   /* Preparing the JCL symbol name */
   symname.length=strlen(symbol);
   memcpy(symname.string, symbol,strlen(symbol));

   /* Retrieving the value of the JCL symbol */
   __le_ceegtjs(&funcode,&symname,symvalue,&valuelen,&fc);
   if( fc.tok_sev > 0) {
      printf("__le_ceegtjs failed with message number %d\n",
             fc.tok_msgno);
      exit(1);
   }
   symvalue[valuelen]='\0';
   printf("The value of JCL symbol %s is %s. The length
          of the value is %d\n",symbol,symvalue,valuelen);
}
```

Output

Use the following JCL to run CELEBL31:

```
//JOB1     JOB    FELE,MSGLEVEL=(2,0)
//STEP1    EXEC   PGM=CELEBL31
//E1       EXPORT SYMLIST=(SYM1,SYM2,SYM3)
//S1       SET    SYM1=XXXX
//S2       SET    SYM2=YYYY
//STEPLIB  DD     DSN=USER.LOADLIB,DISP=SHR
//SYSPRINT DD     SYSOUT=*
//SYSOUT   DD     SYSOUT=*
```

Running this example would produce the following output:

```
The value of JCL symbol SYM1 is XXXX. The length of the value is 4.
```

Related information

- "__le_api.h" on page 38

__le_cib_get() — Get condition information block

Standards

Standards / Extensions	C or C++	Dependencies
Language Environment	both	AMODE 64

Format

```
#include <__le_api.h>

struct cib  *__le_cib_get(void);
```

General description

Returns the Condition Information Block (CIB) structure associated with the current signal.

Notes:

1. This function is valid when called while a Language Environment exception handler is running.
2. This function is valid when called while a POSIX(OFF) signal catcher is running.
3. This function is valid when called while a POSIX(ON) signal catcher is running, if the signal is generated and caught immediately to the same thread. __le_cib_get() will fail if called from POSIX(ON) signal catchers that are driven as a result of signals generated by another thread or process. It may also fail when called from a catcher, if the caught signal is from the same thread but was delayed by blocking or by other signals being delivered at the same time.

Returned value

If there is an active condition the returned value is a pointer to the currently active CIB. If there is more than one active condition, the returned CIB will be for the most recent (most deeply nested) condition.

NULL is returned there is no active CIB, and the errno will be set to EMVSERR.

Error Code
Description

EMVSERR
No active CIB is available.

__le_condition_token_build() — Build a Language Environment condition token

Standards

Standards / Extensions	C or C++	Dependencies
Language Environment	both	AMODE 64

Format

```
#include <__le_api.h>

void  *__le_condition_token_build(_INT2 * c_1, _INT2 * c_2,
                                  _INT2 * format, _INT2 * severity,
                                  _INT2 * control, _CHAR3 facility_ID,
                                  _INT4 * i_s_info,
                                  _FEEDBACK * cond_token,
                                  _FEEDBACK * fc);
```

General description

Dynamically constructs a 16–byte Language Environment condition token. The condition token is only to be used to retrieve messages from a Language Environment message file.

Parameter
Description

c_1 *c_1* is a 2–byte binary integer representing the value of the first 2 bytes of the 4–byte condition_ID. *c_1* and *c_2* make up the condition_ID portion of the condition token.

c_2 *c_2* is a 2–byte binary integer representing the value of the second 2 bytes of the 4–byte condition_ID.

For format 1, this is the Msg_No; for format 2, the cause_code.

format A 2–byte binary integer defining the format of the condition_ID portion of the token.

severity
A 2–byte binary integer indicating the condition's severity. In both format 1 and 2 conditions, this field is used to test the condition's severity. For format 1 conditions, the value of this field is the same as the severity value specified in the condition_ID.

Possible *severity* Values:

0 = Information only, if entire token is 0 there is no information.
1 = Warning
2 = Error
3 = Severe Error
4 = Critical Error

control A 2–byte binary integer containing flags describing or controlling various aspects of condition handling. Valid values for the control field are 1 and 0. 1 indicates the *facility_ID* assigned by IBM, 0 indicates the *facility_ID* assigned by the user.

facility_ID
A 3 character field containing three alphanumeric characters (A-Z, a-z, and 0-9) identifying the product or component of a product generating this condition or feedback information, for example, CEE.

The *facility_ID* is associated with the repository of the runtime messages. If a unique ID is required (for IBM and non-IBM products), an ID can be obtained by contacting an IBM project office.

If you creat a new *facility_ID* to use with a message table, created using the CEEBLDTX utility, be aware that the *facility_ID* must be part of the Language Environment message table name. For more information about the CEEBLDTX utility, see *z/OS Language Environment Programming Guide*. It is important to follow the naming guidelines below in order to have a module name that does not cause your application to abend.

First, begin a non-IBM assigned product *facility_ID* with letters J through Z. (See the *control* parameter above to indicate whether the *facility_ID* has been assigned by IBM.) Secondly, special characters, including blank spaces, cannot be used in a *facility_ID*. Lastly, there are no other constraints (besides the alphanumeric requirement) on a non-IBM assigned *facility_ID*.

i_s_info
A fullword binary integer identifying the ISI, that contains insert data.

cond_token
A 16–byte representation of the constructed condition token.

fc A 16–byte Feedback Code indicating the results of this function.

Table 35. Feedback Codes for __le_condition_token_build()

Code	Severity	Message Number	Message Text
CEE000	0	- -	The function completed successfully.
CEE0CH	3	401	A non-valid case code *case-code* was passed to routine *routine-name*.
CEE0CI	3	402	A non-valid control code *control-code* was passed to routine *routine-name*.
CEE0CJ	3	403	A non-valid severity code *severity-code* was passed to routine *routine-name*.
CEE0CK	3	404	Facility ID, *facility-id*, with non-alphanumeric characters was passed to routine *routine-name*.
CEE0E4	1	452	An invalid facility ID *facility-id* was passed to routine *routine-name*.

Usage notes

1. The structure of the condition token (type _FEEDBACK) is described in the "__le_api.h" header file shipped with Language Environment. You can assign values directly to the fields of the token in the header file without using the __le_condition_token_build() function.
2. This condition token is **only** to be used to retrieve messages from a Language Environment message table.

Related information

- “__le_api.h” on page 38

__le_debug_set_resume_mch() — Move the resume cursor to a predefined location represented by a machine state

Standards

Standards / Extensions	C or C++	Dependencies
Language Environment	both	AMODE 64

Format

```
#include <__le_api.h>

void  __le_debug_set_resume_mch(__mch_t * position, _FEEDBACK * fc);
```

General description

Moves the resume cursor to a predefined location represented by the machine state.

Parameter
Description

position
A pointer to a valid machine state block to which the resume cursor is moved.

fc A 16–byte Feedback Code indicating the results of this function.

Table 36. Feedback Codes for __le_debug_set_resume_mch()

Code	Severity	Message Number	Message Text
CEE000	0	- -	The function completed successfully.
CEE07V	3	255	*Position* parameter not a machine state

Related information

- “__le_api.h” on page 38

__le_msg_add_insert() — Add insert to a Language Environment message

Standards

Standards / Extensions	C or C++	Dependencies
Language Environment	both	AMODE 64

Format

```
#include <__le_api.h>

void  *__le_msg_add_insert(_FEEDBACK * cond_token,
                           _INT4 * insert_seq_num,
                           _VSTRING * insert_data,
                           _FEEDBACK * fc);
```

General description

Copies message insert data and loads the address of that data into the Instance Specific Information (ISI) associated with the condition being processed. The number of ISIs per thread is limited to 15.

Parameter
Description

cond_token
A 16–byte condition token that defines the condition for which the q_data_token is retrieved.

insert_seq_num
A 4–byte integer that contains the insert sequence number (such as insert 1 insert 2). It corresponds to an insert number specified with an :ins. tag in the message source file created by the CEEBLDTX utility. For more information about the CEEBLDTX utility see *z/OS Language Environment Programming Guide*.

insert_data
A halfword-prefixed length string, used without truncation, that represents the insert data. DBCS strings must be enclosed within shift-out (0x0E) and shift-in (0x0F) characters.

Note: The maximum size for an individual insert item is 254 bytes.

fc A 16–byte Feedback Code indicating the results of this function.

Table 37. Feedback Codes for __le_msg_add_insert()

Code	Severity	Message Number	Message Text
CEE000	0	- -	The function completed successfully.
CEE0EB	3	459	Not enough storage was available to create a new Instance Specific Information block.
CEE0EC	1	460	Multiple instances of the condition token with message number *message-number* and facility ID *facility-id* were detected.

Table 37. Feedback Codes for __le_msg_add_insert() (continued)

Code	Severity	Message Number	Message Text
CEE0ED	3	461	The maximum number of unique message insert blocks was reached. This condition token had it's I_S_info field set to 1.
CEE0EE	3	462	Instance Specific Information for the condition token with message number *message-number* and facility ID *facility-id* could not be found.
CEE0EF	3	463	The maximum size for an insert data item was exceeded be located.
CEE0H9	3	553	An internal error was detected in creating the inserts for a condition.

Usage notes

1. z/OS UNIX System Services consideration – In multithreaded applications, __le_msg_add_insert() applies to message insert data for only the invoking thread.

Related information

- "__le_api.h" on page 38
- "__le_msg_get() — Get a Language Environment message"
- "__le_msg_get_and_write() — Get and output a Language Environment message" on page 955
- "__le_msg_write() — Output a Language Environment message to stderr" on page 956

__le_msg_get() — Get a Language Environment message

Standards

Standards / Extensions	C or C++	Dependencies
Language Environment	both	AMODE 64

Format

```
#include <__le_api.h>

void  *__le_msg_get(_FEEDBACK * cond_token,
                    _CHAR80 message_area,
                    _INT4 * msg_index,
                    _FEEDBACK * fc);
```

General description

Retrieves, formats, and stores, in a passed message area, a Language Environment message corresponding to a user supplied condition token. The caller can later retrieve the message to modify or to write as output.

Parameter
Description

cond_token
A 16–byte condition token supplied by the invoker.

message_area
A fixed-length 80 character string, where the message is placed.

Note: The message is left-justified and padded on the right with blanks.

msg_index
A 4–byte binary integer returned to the invoker.

The *msg_index* should be set to zero on the first invocation of __le_msg_get(). If a message is too large to be contained in the *message_area*, *msg_index* is returned as an index into the message. This index is used on subsequent invocations to retrieve the remaining portion of the message. Feedback Code is also returned, indicating the message has been truncated. When the entire message is returned, *msg_index* is zero.

msg_idex contains different results based on the length of the message.

- If a message contains fewer than 80 characters, the entire message is returned on the first invocation. *msg_index* contains 0.
- If a message contains exactly 80 characters, the entire message is returned on the first invocation. *msg_index* contains 0.
- If the message is more than 80 characters it is split into segments. The *msg_index* does not contain the cumulative index for the entire message returned, but contains only the index of the segment that was just returned. It is up to the user to maintain the cumulative count if needed. When a message is too long, the following can occur:
  - If a message contains more than 80 characters and at least one blank is contained in the first 80 characters, the string up to and including the last blank is returned on the first invocation.
  - If the 80th character is non-blank (even if the 81st character is a blank), *msg_index* contains the index of the last blank (something less than 80), and the next invocation starts with the next character.
  - If the 80th character is a blank, *msg_index* contains 80 and the next invocation starts with the 81st character, blank or non-blank.
  - If a message contains more than 80 characters and at least the first 80 are all non-blank, the first 80 are returned. The next invocation does not add any blanks and starts with the 81st character. *msg_index* contains 80.

fc
A 16–byte Feedback Code indicating the results of this function.

Table 38. Feedback Codes for __le_msg_get()

Code	Severity	Message Number	Message Text
CEE000	0	- -	The function complete successfully.
CEE036	3	102	An unrecognized condition token was passed to the function and could not be used.
CEE0E2	3	450	The message inserts for the condition token with message number *message-number* and facility ID *facility-id* could not be located.
CEE0E6	3	454	The message number *message-number* could not be found for facility ID *facility-id*.

Table 38. Feedback Codes for __le_msg_get() (continued)

Code	Severity	Message Number	Message Text
CEE0E7	1	455	The message with message number *message-number* and facility ID *facility-id* was truncated.
CEE0EA	1	458	The message repository *repository-name* could not be located.

Usage notes

1. z/OS UNIX System Services consideration – In multithreaded applications, __le_msg_get() affects only the invoking thread. However, __le_msg_get() uses the NATLANG value of the enclave. Any subsequent calls to __le_msg_get(), for a given condition, use the NATLANG value in effect at the time of the first invocation.

Related information

- "__le_api.h" on page 38
- "__le_msg_add_insert() — Add insert to a Language Environment message" on page 952
- "__le_msg_get_and_write() — Get and output a Language Environment message"
- "__le_msg_write() — Output a Language Environment message to stderr" on page 956

__le_msg_get_and_write() — Get and output a Language Environment message

Standards

Standards / Extensions	C or C++	Dependencies
Language Environment	both	AMODE 64

Format

```
#include <__le_api.h>

void  *__le_msg_get_and_write(_FEEDBACK * cond_token,
                              _INT4 * destination_code,
                              _FEEDBACK * fc);
```

General description

Retrieves, formats, and stores, in a passed message area, a Language Environment message corresponding to a user supplied condition token. The caller can later retrieve the message to modify or to write as output.

Parameter
  Description

cond_token
  A 16–byte condition token supplied by the invoker.

destination_code
A 4–byte binary integer written to 'stderr'. The only acceptable value for is 2.

fc A 16–byte Feedback Code indicating the results of this function.

Table 39. Feedback Codes for __le_msg_get_and_write()

Code	Severity	Message Number	Message Text
CEE000	0	- -	The function completed successfully.
CEE0E2	3	450	The message inserts for the condition token with message number *message-number* and facility ID *facility-id* could not be located.
CEE0E3	3	451	An invalid destination code *destination-code* was passed to routine *routine*.
CEE0E6	3	454	The message number *message-number* could not be found for facility ID *facility-id*.
CEE0E9	1	457	The message file destination *ddname* could not be located.
CEE0EA	1	458	The message repository *repository-name* could not be located.
CEE3CT	3	3,485	An internal message service error occurred while locating the message number within a message file.
CEE3CU	3	3,486	An internal message service error occurred while formatting a message.
CEE3CV	3	3,487	An internal message service error occurred while locating a message number within the ranges specified in the repository.

Usage notes

1. z/OS UNIX System Services consideration – In multithreaded applications, __le_msg_get_and_write() affects only the invoking thread. When multiple threads write to 'stderr' the output is interwoven by line. To group lines of output, serialize 'stderr' access (for example, by using a mutex).

Related information

- "__le_api.h" on page 38
- "__le_msg_add_insert() — Add insert to a Language Environment message" on page 952
- "__le_msg_get() — Get a Language Environment message" on page 953
- "__le_msg_write() — Output a Language Environment message to stderr"

__le_msg_write() — Output a Language Environment message to stderr

Standards

Standards / Extensions	C or C++	Dependencies
Language Environment	both	AMODE 64

Format

```
#include <__le_api.h>

void  *__le_msg_write(_VSTRING * message_string,
                      _INT4 * destination_code,
                      _FEEDBACK * fc);
```

General description

Writes a user-defined Language Environment message string to 'stderr'.

Parameter
  Description

message_string
A halfword-prefixed printable character string containing a message. DBCS characters must be enclosed within shift-out (0x0F) and shift-in (0x0E) characters.

Insert data cannot be placed in the message with __le_msg_write(). The halfword-prefixed message string must contain only printable characters and be a length greater than zero. Unpredictable results will occur if the byte following the halfword prefix is 0x00.

destination_code
A 4–byte binary integer written to 'stderr'. The only acceptable value is 2.

fc A 16–byte Feedback Code indicating the results of this function.

Table 40. Feedback Codes for __le_msg_write()

Code	Severity	Message Number	Message Text
CEE000	0	- -	The function completed successfully.
CEE0E3	3	451	An invalid destination code *destination-code* was passed to routine *routine*.
CEE0E9	3	457	The message file destination *ddname* could not be located.

Usage notes

1. z/OS UNIX System Services consideration – In multithreaded applications, __le_msg_write() affects only the invoking thread. When multiple threads write to 'stderr' the output is interwoven by line. To group lines of output, serialize 'stderr' access (for example, by using a mutex).

Related information

- "__le_api.h" on page 38
- "__le_msg_add_insert() — Add insert to a Language Environment message" on page 952
- "__le_msg_get() — Get a Language Environment message" on page 953
- "__le_msg_get_and_write() — Get and output a Language Environment message" on page 955

__le_record _dump() — Record information for the active condition

Standards

Standards / Extensions	C or C++	Dependencies
Language Environment	both	AMODE 64

Format

```
#include <__le_api.h>

int __le_record _dump(int function_code,
                      void *information);
```

General description

The __le_record _dump() function records information of the active condition, so that the information can be retrieved from the CIB later. Use this function only during condition handling.

Parameter
Description

function_code
A fullword integer that contains the function code of the following value:

1 Record the dataset name of a dump for the active condition.

information
The information to be recorded. When the value of *function_code* is 1, the information is a halfword length-prefixed EBCDIC character string, which is expected to be the dataset name of a dump for the active condition. Language Environment will validate that the length is positive and it does not exceed 44.

Returned value

If successful, __le_record_dump() returns 0.

If unsuccessful, __le_record_dump() returns nonzero and sets errno to one of the following values:

Error Code
Description

EMVSERR
No active CIB is available.

EINVAL
Incorrect parameters detected.

Usage notes

1. After the condition handling functions return control to your application, CIB that represents the condition is no longer valid, therefore the recorded information is no longer accessible.
2. If __le_record_dump() is called more than once for the same condition, the information of the last call is recorded.

__le_traceback() — Call chain traceback service

Standards

Standards / Extensions	C or C++	Dependencies
Language Environment	both	z/OS V1.9 AMODE 64

Format

```
#include <__le_api.h>

void  __le_traceback(int cmd, void* cmd_parms, _FEEDBACK *fc);
```

General description

The __le_traceback() function assists in tracing the call chain. It identifies the language, program unit, entry point, current location, caller's DSA, and other information from the address of a DSA for a program unit. This is essential for creating meaningful traceback messages.

Parameter
Description

cmd The __le_traceback() command to be used. The following commands can be used:

__TRACEBACK_FIELDS
Information that can be used to create a traceback message is returned in individual fields.

cmd_parms
A pointer to a structure that contains additional command specific parameters. For the command __TRACEBACK_FIELDS, this parameter must point to a __tf_parms_t.

fc A 16-byte feedback code indicating the results of this function.

Table 41. Feedback Codes for __le_traceback()

Code	Severity	Message number	Message text
CEE000	0	- -	The service completed successfully.
CEE310	3	3104S	Information could not be successfully extracted for this DSA. It is likely that the dsaptr parameter does not point to an actual DSA or save area.
CEE316	2	3110E	The *cmd* parameter is not a valid command for __le_traceback().
CEE3NS	1	3836W	A statement number is not available for this DSA. DWARF data in the load module is corrupted.
CEE3NT	1	3837W	Statement numbers are not available. The explicit DLL load of DLL CDAEQED failed with feedback code *fc*.

Table 41. Feedback Codes for __le_traceback() (continued)

Code	Severity	Message number	Message text
CEE3NU	1	3838W	Statement numbers are not available. The explicit DLL load of DLL CDAEQDPI failed with feedback code *fc*.
CEE3NV	1	3839W	Statement numbers are not available. The explicit DLL load of DLL CELQDSNF failed with feedback code *fc*.
CEE3O0	1	3840	Statement numbers are not available. dllqueryfn() failed for a function in the DLL CELQDSNF.
CEE3O1	1	3841	A statement number is not available for this DSA. An internal routine failed with return code *return-code* and reason code *reason-code*.

The __tf_parms_s structure is defined as follows:

```
typedef struct __tf_string_s {
     size_t     __tf_bufflen;

     char*   __tf_buff;
} __tf_string_t;

typedef struct __tf_parms_s {
     /*****************************************************************/
     /* Input                                                         */
     /*****************************************************************/
     void*   __tf_dsa_addr;
     void*   __tf_caa_addr;
     void*   __tf_call_instruction;
     /*****************************************************************/
     /* Output related to input DSA                                   */
     /*****************************************************************/
     void*   __tf_ pu_addr;
     void*   __tf_ entry_addr;
     struct __cib*   __tf_cib_addr;
     uint8_t     __tf_member_id;
     int     __tf_is_main:1;
     int     :23;
     int     :32;
     __tf_string_t __tf_pu_name;
     __tf_string_t __tf_entry_name;
     __tf_string_t __tf_statement_id;
     /*****************************************************************/
     /* Output related to caller's DSA                                */
     /*****************************************************************/
     void*   __tf_caller_dsa_addr;
     void*   __tf_caller_call_instruction;
} __tf_parms_t;
```

The following are members of the structure:

Member
Description

void* __tf_dsa_addr
The address of the DSA for the current routine in the traceback. When this field is zero on input, the address of the DSA for the caller of

__le_traceback() will be used and the address will be returned. No attempt is made to verify that the input is a DSA. Incorrect input can lead to unpredictable results.

void* __tf_caa_addr
The address of the CAA associated with the DSA. When this field is zero on input, the address of the CAA for the current thread will be used and the address will be returned. No attempt is made to verify that the input is a CAA. Incorrect input can lead to unpredictable results.

void* __tf_call_instruction
The address of the instruction that caused transfer out of the routine. This is either the address of a BASR, BRAS or BRASL instruction if transfer was made by subroutine call, or the address of the interrupted statement if transfer was caused by an exception. When multiple calls are made to __le_traceback() to scan the call chain, the callers_call_instruction (described below) returned from the previous call can be used here. If the address is not known, this field should be set to zero. When this field is zero on input and the address can be determined, it will be returned.

void* __tf_pu_addr
The address of the start of the program unit for the routine associated with the DSA is returned in this field. If the program unit address cannot be determined, this field is set to zero.

void* __tf_entry_addr
The address of the entry point into the routine associated with the DSA is returned in this field. If the entry point address cannot be determined, this parameter is set to zero.

struct __cib* __tf_cib_addr
The address of the CIB (struct __cib) associated with the DSA, if an exception occurred, is returned in this field. If no exception occurred, this field is set to zero. Note that if an exception caused transfer out of the routine, the state of the registers after the last instruction ran in the routine is saved in the CIB, rather than in the DSA.

uint8_t __tf_member_id
The member identifier for the routine associated with the DSA will be returned in this field. If the member ID cannot be determined, this field is set to negative one.

int __tf_is_main:1
One of two values is returned in this field: 0 (the routine associated with the DSA is not the main program) or 1 (the routine associated with the DSA is the main program).

__tf_string_t __tf_pu_name
A structure that will be used to return the name of the program unit containing the routine associated with the DSA. The structure has the following fields:

char* __tf_buff
The address of a buffer in which the program unit name will be returned. The name will be returned in the buffer as a null terminated string.

size_t __tf_bufflen
The size of the buffer

If the program unit name cannot be determined, the buffer is set to a null string. If the program unit name cannot fit within the supplied string, it is truncated. (Truncation of DBCS preserves even byte count and SI/SO pairing.) If __tf_buff is NULL or __tf_bufflen is zero, the program unit name is not returned.

__tf_string_t __tf_entry_name
A structure that will be used to return the name of the entry point into the routine associated with the DSA. The structure has the following fields:

char* __tf_buff
The address of a buffer in which the entry point name will be returned. The name will be returned in the buffer as a null terminated string.

size_t __tf_bufflen
The size of the buffer

If the entry point name cannot be determined, the buffer is set to a null string. If the entry point name cannot fit within the supplied string, it is truncated. (Truncation of DBCS preserves even byte count and SI/SO pairing.) If __tf_buff is NULL or __tf_bufflen is zero, the entry point name is not returned

__tf_string_t __tf_statement_id
A structure that will be used to return the identifier of the statement containing the instruction which caused transfer out of the routine associated with the DSA. The structure has the following fields:

char* __tf_buff
The address of a buffer in which the statement id will be returned. The statement id will be returned in the buffer as a null terminated string.

size_t __tf_bufflen
The size of the buffer

If the statement id cannot be determined, the buffer is set to a null string. If the statement id cannot fit within the supplied string, it is truncated. (Truncation of DBCS preserves even byte count and SI/SO pairing.) If __tf_buff is NULL or __tf_bufflen is zero, the statement id is not returned

void* __tf_callers_dsa_addr
The address of the DSA for the caller is returned in this field. If the address of the caller's DSA cannot be determined or is not valid (points to inaccessible storage), then this field is set to zero.

void* __tf_callers_call_instruction
The address of the instruction that caused transfer out of the caller is returned in this field. This is either the address of a BASR, BRAS or BRASL instruction if transfer was made by subroutine call, or the address of the interrupted statement if transfer was caused by an exception. If the address cannot be determined, this parameter is set to zero.

Example

```
#include <__le_api.h>
#include <stdlib.h>

int main() {
  __tf_parms_t   tbck_parms;
  char                           pu_name[256];
  char                           entry_name[256];
  char                              statement_id[256];
  _FEEDBACK                         fc;
  int                               rc;


  tbck_parms.__tf_pu_name. __tf_bufflen = sizeof(pu_name);
  tbck_parms.__tf_entry_name. __tf_bufflen = sizeof(entry_name);
  tbck_parms.__tf_statement_id. __tf_bufflen = sizeof(statement_id);

  tbck_parms.__tf_pu_name. __tf_buff = pu_name;
  tbck_parms.__tf_entry_name. __tf_buff = entry_name;
  tbck_parms.__tf_statement_id. __tf_buff = statement_id;

  tbck_parms.__tf_dsa_addr = 0;
  tbck_parms.__tf_caa_addr = 0;
  tbck_parms.__tf_call_instruction = 0;

  do {

    _le_traceback(__TRACEBACK_FIELDS, &tbck_parms, &fc);

    if ( fc.tok_sev >= 2 ) {
      printf("Error: __le_traceback() failed.\n");
      break;
    }

    printf("Entry=%s Offset=%c%x Line=%s\n",
           tbck_parms.__tf_entry_name.__tf_buff,
           tbck_parms.__tf_call_instruction
           < tbck_parms.__tf_entry_addr ? '-' : '+',
           abs((int)((long)tbck_parms.__tf_call_instruction
           - (long)tbck_parms.__tf_entry_addr)),
           tbck_parms.__tf_statement_id.__tf_buff
          );

    tbck_parms.__tf_dsa_addr = tbck_parms.__tf_caller_dsa_addr;
    tbck_parms.__tf_call_instruction =
       tbck_parms.__tf_caller_call_instruction;


  } while (!tbck_parms.__tf_is_main);

  return 0;
}
```

Output

```
Entry=main Offset=+da Line=28
Entry=CELQINIT Offset=+134c Line=
```

Related information

- “__le_api.h” on page 38

lfind() — Linear search routine

Standards

Standards / Extensions	C or C++	Dependencies
XPG4 XPG4.2 Single UNIX Specification, Version 3	both	

Format

```
#define _XOPEN_SOURCE
#include <search.h>

void *lfind(const void *key, const void *base, size_t *nelp,
           size_t width, int (*compar)(const void *, const void *));
```

General description

The lfind() function is the same as a lsearch() except that if the entry is not found, it is not added to the table. Instead, a NULL pointer is returned.

Special behavior for C++: Because C and C++ linkage conventions are incompatible, lfind() cannot receive a C++ function pointer as the comparator argument. If you attempt to pass a C++ function pointer to lfind(), the compiler will flag it as an error. You can pass a C or C++ function to lfind() by declaring it as extern "C".

Returned value

If the searched-for entry is found, lfind() returns a pointer to it.

If not found, lfind() returns a NULL pointer.

No errors are defined.

Related information

- "search.h" on page 62
- "bsearch() — Search arrays" on page 220
- "hsearch() — Search hash tables" on page 817
- "lsearch() — Linear search and update" on page 1019
- "tsearch() — Binary tree search" on page 1920

lgamma(), lgammaf(), lgammal() — Log gamma function

Standards

Standards / Extensions	C or C++	Dependencies
XPG4 XPG4.2 C99 Single UNIX Specification, Version 3 C++ TR1 C99	both	

Format

```
#define _XOPEN_SOURCE
#include <math.h>

double lgamma(double x);
extern int signgam;
int *__signgam(void);
```

C99:

```
#define _ISOC99_SOURCE
#include <math.h>

double lgamma(double x);
float lgammaf(float x);
long double lgammal(long double x);
```

C++ TR1 C99:

```
#define _TR1_C99
#include <math.h>

float lgamma(float x);
long double lgamma(long double x);
```

General description

The lgamma() function computes the

$$\log_e | \Gamma(x) |$$

where

$$\Gamma(x)$$

is defined as

$$\int_0^{\infty} e^{-t} \; t^{(x - 1)} \, dt$$

The sign of

$$\Gamma(x)$$

is returned in the external integer *signgam*. The argument *x* may not be a non-positive integer.

In a multithreaded process, each thread has its own instance of the *signgam* variable. Threads access their instances of the variable by calling the __signgam() function. See "__signgam() — Return signgam reference" on page 1648. The math.h header (see "math.h" on page 44) redefines the string "*signgam*" to an invocation of the __signham function. The actual *signgam* external variable is used to store the *signgam* value for the IPT.

Note: The following table shows the viable formats for these functions. See "IEEE binary floating-point" on page 94 for more information about IEEE Binary Floating-Point.

Function	Hex	IEEE
lgamma	X	X
lgammaf	X	X
lgammal	X	X

Returned value

If successful, lgamma() returns the above function of its argument.

lgamma() will fail under the following conditions:

- If the result overflows, the function will return HUGE_VAL and set errno to ERANGE.
- If *x* is a non-positive integer and _XOPEN_SOURCE is defined, lgamma() returns HUGE_VAL and sets errno to EDOM.
- If *x* is a non-positive integer and _ISOC99_SOURCE is defined, lgamma() returns HUGE_VAL and sets errno to ERANGE.

Note: If both _XOPEN_SOURCE and _ISOC99_SOURCE are defined, the _ISOC99_SOURCE behavior will take precedence.

Special behavior for IEEE: Even when _XOPEN_SOURCE is defined and _ISOC99_SOURCE not, lgamma() returns HUGE_VAL and sets errno to ERANGE.

Example

```
/*
   This example uses lgamma() to calculate ln(|G(x)|), where x = 42.
 */
#include <math.h>
#include <stdio.h>

int main(void)
{
   double x=42, g_at_x;

   g_at_x = exp(lgamma(x));       /* g_at_x = 3.345253e+49 */
   printf ("The value of G(%4.2f) is %7.2e\n", x, g_at_x);
}
```

Output

```
The value of G(42.00) is 3.35e+49
```

Related information

- “math.h” on page 44
- “exp(), expf(), expl() — Calculate exponential function” on page 448
- “isnan() — Test for NaN” on page 916
- “__signgam() — Return signgam reference” on page 1648

lgammad32(), lgammad64(), lgammad128() - Log gamma function

Standards

Standards / Extensions	C or C++	Dependencies
C/C++ DFP	both	z/OS V1.10

Format

```
#define __STDC_WANT_DEC_FP__
#include <math.h>

_Decimal32  lgammad32(_Decimal32 x);
_Decimal64  lgammad64(_Decimal64 x);
_Decimal128 lgammad128(_Decimal128 x);
```

```
_Decimal32  lgamma(_Decimal32 x);     /* C++ only */
_Decimal64  lgamma(_Decimal64 x);     /* C++ only */
_Decimal128 lgamma(_Decimal128 x);    /* C++ only */
```

General description

The lgamma() function computes the

$$\log_e | \Gamma (x) |$$

is defined as

$$\int_{o}^{\infty} e^{-t} \; t^{(x-1)} \, dt$$

The sign of

$$\Gamma (x)$$

is returned in the external integer signgam. The argument *x* may not be a non-positive integer.

In a multithreaded process, each thread has its own instance of the *signgam* variable. Threads access their instances of the variable by calling the __signgam() function. The math.h header redefines the string *signgam* to an invocation of the __siggam function. The actual signgam external variable is used to store the signgam value for the IPT.

These functions work in IEEE decimal floating-point format. See “IEEE decimal floating-point” on page 95 for more information.

Note: To use IEEE decimal floating-point, the hardware must have the Decimal Floating-Point Facility installed.

Returned value

If successful, lgamma() returns the above function of its argument.

lgamma() will fail under the following conditions:
- If the result overflows, the function will return +HUGE_VAL_D32, +HUGE_VAL_D64 or +HUGE_VAL_D128 and set errno to ERANGE.
- If *x* is a non-positive integer, lgamma() returns +HUGE_VAL_D32, +HUGE_VAL_D64 or +HUGE_VAL_D128 and sets errno to ERANGE.

Example

CELEBL26

```
/* CELEBL26

   This exaxmple illustrates the lgammad64() function.

*/
```

```
#define __STDC_WANT_DEC_FP__
#include <math.h>
#include <stdio.h>
int main(void)
{
   _Decimal64 x, y;

   x = 42.0DD;
   y = lgammad64(x);

   printf ("lgammad64(%Df) = %Df\n", x, y);
}
```

Related information

- “math.h” on page 44
- “expd32(), expd64(), expd128() — Calculate exponential function” on page 449
- “isnan() — Test for NaN” on page 916
- “__signgam() — Return signgam reference” on page 1648

__librel() — Query release level

Standards

Standards / Extensions	C or C++	Dependencies
Language Environment	both	

Format

```
#include <stdlib.h>

int __librel(void);
```

General description

The __librel() function provides the release level of the z/OS XL C/C++ library. To use this function, you must compile your application with LANGLVL(EXTENDED).

Returned value

The __librel() function returns the z/OS XL C/C++ library release level that your application is using. The value is printed in hexadecimal format. The first byte of the value returned contains the product and version, second byte the release, and the third and fourth bytes contain the modification level.

Figure 4 shows the format of the hexadecimal value returned by the __librel() function.

```
0xPVRRMMMM
  ||||||||
  ||||++++ Modification Level (M)
  ||++---- Release (R)
  |+------ Version (V)
  +------- Product Number (P)
```

Figure 4. Format of the __librel() function return value

Table 42 on page 969 correlates library release level with the corresponding __librel() return value.

Table 42. Library release level and value returned by the __librel() function

Library release level	Return value	Notes
z/OS V2.2	0x42020000	
z/OS V2.1	0x42010000	
z/OS V1.13	0x410D0000	
z/OS V1.12	0x410C0000	
z/OS V1.11	0x410B0000	
z/OS V1.10	0x410A0000	
z/OS V1.9	0x41090000	
z/OS V1.8	0x41080000	
z/OS V1.7	0x41070000	
z/OS V1.6	0x41060000	
z/OS V1.5	0x41050000	
z/OS V1.4	0x41040000	
z/OS V1.3	0x41030000	
z/OS V1.2	0x41020000	
z/OS V1.1	0x220A0000	When running under z/OS V1.1, the __librel() function returns the same value as returned for OS/390 V2.10.
OS/390 V2.10	0x220A0000	
OS/390 V2.9	0x22090000	
OS/390 V2.8	0x22080000	
C/370 V2.2	0x02020000	For application running under the C/370 specific library (the common library version), the product designation is 0.

Example

CELEBL04

```
/* CELEBL04

   This example calls the __librel() function that returns
   the library release level your program is currently
   using in the following hexadecimal format 0xPVRRMMMM.

 */
#include <stdio.h>
#include <stdlib.h>

int main(void)
{
   printf("The current release of the library is: %X\n",__librel());
}
```

Output

```
The current release of the library is: 42020000
```

Related information

- “stdlib.h” on page 70

link() — Create a link to a file

Standards

Standards / Extensions	C or C++	Dependencies
POSIX.1 XPG4 XPG4.2 Single UNIX Specification, Version 3	both	

Format

```
#define _POSIX_SOURCE
#include <unistd.h>

int link(const char *oldfile, const char *newname);
```

General description

Provides an alternative pathname for the existing file, so that the file can be accessed by either the old or the new name. link() creates a link from the pathname *newname* to an existing file, with the pathname *oldfile*. The link can be stored in the same directory as the original file or in a completely different one.

Links are allowed to files only, not to directories.

This is a hard link, which ensures the existence of a file even after its original name has been removed.

If link() successfully creates the link, it increments the *link count* of the file. The link count tells how many links there are to the file. At the same time, link() updates the change time of the file, and the change time and modification time of the directory that contains *newname* (that is, the directory that holds the link). If link() fails, the link count is not incremented.

If *oldfile* names a symbolic link, link() creates a link that refers to the file that results from resolving the pathname contained in the symbolic link. If *newname* names a symbolic link, link() fails and sets errno to EEXIST.

Returned value

If successful, link() returns 0.

If unsuccessful, link() returns -1 and sets errno to one of the following values:

Error Code
Description

EACCES
The process did not have appropriate permissions to create the link. Possible reasons include no search permission on a pathname component of *oldfile* or *newname*, no write permission on the directory intended to contain the link, or no permission to access *oldfile*.

EEXIST
Either *newname* refers to a symbolic link, or a file or directory with the name *newname* already exists.

EINVAL
Either *oldfile* or *newname* is incorrect, because it contains a NULL.

ELOOP
A loop exists in symbolic links. This error is issued if the number of symbolic links encountered during resolution of *oldfile* or *newname* is greater than POSIX_SYMLOOP.

EMLINK
oldfile already has its maximum number of links. The maximum number of links to a file is given by **LINK_MAX**, which you can determine by using pathconf() or fpathconf().

ENAMETOOLONG
oldfile or *newname* is longer than **PATH_MAX**, or a component of one of the pathnames is longer than **NAME_MAX** while _POSIX_NO_TRUNC is in effect. For symbolic links, the length of the pathname string substituted for a symbolic link in *oldfile* or *newname* exceeds **PATH_MAX**. The **PATH_MAX** and **NAME_MAX** values can be determined using pathconf().

ENOENT
A pathname component of *oldfile* or *newname* does not exist, or *oldfile* itself does not exist, or one of the two arguments is an empty string.

ENOSPC
The directory intended to contain the link cannot be extended to contain another entry.

ENOTDIR
A pathname component of one of the arguments is not a directory.

EPERM
oldfile is the name of a directory, and links to directories are not supported.

EROFS
Creating the link would require writing on a read-only file system.

EXDEV
oldfile and *newname* are on different file systems.

Example

```
#define _POSIX_SOURCE
#include <fcntl.h>
#include <sys/stat.h>
#include <sys/types.h>
#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>

main() {
  char fn[]="link.example.file";
  char ln[]="link.example.link";
  int fd;

  if ((fd = creat(fn, S_IWUSR)) < 0)
    perror("creat() error");
  else {
    close(fd);
    if (link(fn, ln) != 0) {
      perror("link() error");
      unlink(fn);
    }
    else {
      unlink(fn);
      unlink(ln);
    }
  }
}
```

Related information

- “unistd.h” on page 82
- “rename() — Rename file” on page 1433
- “symlink() — Create a symbolic link to a path name” on page 1795
- “unlink() — Remove a directory entry” on page 1962

listen() — Prepare the server for incoming client requests

Standards

Standards / Extensions	C or C++	Dependencies
XPG4.2 Single UNIX Specification, Version 3	both	

Format

X/Open:

```
#define _XOPEN_SOURCE_EXTENDED 1
#include <sys/socket.h>

int listen(int socket, int backlog);
```

Berkeley sockets:

```
#define _OE_SOCKETS
#include <sys/socket.h>

int listen(int socket, int backlog);
```

General description

The listen() function applies only to stream sockets. It indicates a readiness to accept client connection requests, and creates a connection request queue of length *backlog* to queue incoming connection requests. Once full, additional connection requests are rejected.

Parameter
Description

socket The socket descriptor.

backlog Defines the maximum length for the queue of pending connections.

The listen() call indicates a readiness to accept client connection requests. It transforms an active socket into a passive socket. Once called, *socket* can never be used as an active socket to initiate connection requests. Calling listen() is the third of four steps that a server performs to accept a connection. It is called after allocating a stream socket with socket(), and after binding a name to *socket* with bind(). It must be called before calling accept().

If the backlog is less than 0, *backlog* is set to 0. If the backlog is greater than SOMAXCONN, as defined in **sys/socket.h**, *backlog* is set to SOMAXCONN.

For AF_UNIX sockets, this value is variable and can be set in the application. For AF_INET and AF_INET6 sockets, the value cannot exceed the maximum number of connections allowed by the installed TCP/IP.

Special behavior for C++: To use this function with C++, you must use the _XOPEN_SOURCE_EXTENDED 1 feature test macro.

Returned value

If successful, listen() returns 0.

If unsuccessful, listen() returns -1 and sets errno to one of the following values:

Error Code
  Description

EBADF
  The *socket* parameter is not a valid socket descriptor.

EDESTADDRREQ
  The socket is not bound to a local address, and the protocol does not support listening on an unbound socket.

EINVAL
  An invalid argument was supplied. The socket is not named (a bind() has not been done), or the socket is ready to accept connections (a listen() has already been done). The socket is already connected.

ENOBUFS
  Insufficient system resources are available to complete the call.

ENOTSOCK
  The descriptor is for a file, not for a socket.

EOPNOTSUPP
  The *socket* parameter is not a socket descriptor that supports the listen() call.

Related information

- “sys/socket.h” on page 74
- “accept() — Accept a new connection on a socket” on page 105
- “bind() — Bind a name to a socket” on page 210
- “connect() — Connect a socket” on page 308
- “socket() — Create a socket” on page 1688

llabs() — Calculate absolute value of long long integer

Standards

Standards / Extensions	C or C++	Dependencies
z/OS UNIX C99 Single UNIX Specification, Version 3 C++ TR1 C99	both	OS/390 V2R10

Format

```
#include <stdio.h>

long long llabs(long long int n);
```

Compile requirement: Use of this function requires the long long data type. See *z/OS XL C/C++ Language Reference* for information on how to make long long available.

General description

Calculates the absolute value of its long long integer argument *n*. The result is undefined when the argument is equal to `LONGLONG_MIN`, the smallest available long long integer (-9 223 372 036 854 775 808). The value `LONGLONG_MIN` is defined in the limits.h header file.

Returned value

Returns the absolute value of the long long integer argument *n*.

Related information

- "stdio.h" on page 68
- "stdlib.h" on page 70
- "limits.h" on page 39
- "labs() — Calculate long absolute value" on page 935

lldiv() — Compute quotient and remainder of integral division for long long type

Standards

Standards / Extensions	C or C++	Dependencies
z/OS UNIX C99 Single UNIX Specification, Version 3 C++ TR1 C99	both	OS/390 V2R10

Format

```
#include <stdio.h>

long long lldiv(long long numer, long long denom);
```

Compile requirement: Use of this function requires the long long data type. See *z/OS XL C/C++ Language Reference* for information on how to make long long available.

General description

Calculates the quotient and remainder of the division of numerator by denominator.

Returned value

Returns a structure of type `lldiv_t`, containing both the quotient `long long quot` and the remainder `long long rem`.

If the value cannot be represented, the returned value is undefined. If *denominator* is 0, a divide by 0 exception is raised.

Example

```
/*
   This example uses the
   lldiv() function to calculate the quotients and
   remainders for a set of two dividends and two divisors.
 */
#define _LONG_LONG 1
#include <stdio.h>
#include <stdlib.h>

int main(void)
{
   long long  num[2] = {45,-45};
   long long  den[2] = {7,-7};
   lldiv_t ans;   /* lldiv_t is a struct type containing
                     two long long int fields:
                    'quot' stores quotient; 'rem' stores remainder */
   short i,j;

   printf("Results of long division:\n");
   for (i = 0; i < 2; i++)
      for (j = 0; j < 2; j++)
      {
         ans = lldiv(num[i], den[j]);
         printf("Dividend: %6lld  Divisor: %6lld", num[i], den[j]);
         printf("  Quotient: %6lld  Remainder: %6lld\n", ans.quot,
                                                          ans.rem);
      }
}
```

Output

```
Results of long division:
Dividend:  45  Divisor:   7  Quotient:   6  Remainder:   3
Dividend:  45  Divisor:  -7  Quotient:  -6  Remainder:   3
Dividend: -45  Divisor:   7  Quotient:  -6  Remainder:  -3
Dividend: -45  Divisor:  -7  Quotient:   6  Remainder:  -3
```

Related information

- "stdio.h" on page 68
- "stdlib.h" on page 70
- "div() — Calculate quotient and remainder" on page 384
- "ldiv() — Compute quotient and remainder of integral division" on page 944

llround(), llroundf(), llroundl() — Round to the nearest integer

Standards

Standards / Extensions	C or C++	Dependencies
C99 Single UNIX Specification, Version 3 C++ TR1 C99	both	z/OS V1R7

Format

```
#define _ISOC99_SOURCE
#include <math.h>

long long int llround(double x);
long long int llroundf(float x);
long long int llroundl(long double x);
```

C++ TR1 C99:

```
#define _TR1_C99
#include <math.h>

long long int llround(float x);
long long int llround(long double x);
```

Compile requirement: Use of this function requires the long long data type. See *z/OS XL C/C++ Language Reference* for information on how to make long long available.

General description

The llround() family of functions round *x* to the nearest integer, rounding halfway cases away from zero, regardless of the current rounding mode.

Note: The following table shows the viable formats for these functions. See "IEEE binary floating-point" on page 94 for more information about IEEE Binary Floating-Point.

Function	Hex	IEEE
llround	X	X
llroundf	X	X
llroundl	X	X

Returned value

If successful, they return the rounded integer. If the correct value is positive or negative and too large to represent as a **long long**, a domain error will occur and an unspecified value is returned.

Example

```
/*
 * This program illustrates the use of llround() function
 *
 */
#define _ISOC99_SOURCE
#include <math.h>
#include <_Ieee754.h>  /* fpc functions */
#include <stdio.h>


void main() {

  _FP_fpcreg_t   save_rmode;
  long long int  rnd2nearest;
  double         number;

  printf("Illustrates the llround() function\n\n");
  save_rmode.rmode = _RMODE_RZ;
  __fpc_sm(save_rmode.rmode);  /* set rounding mode to round to zero */

  number=501.1;
  rnd2nearest = llround(number);
  printf ("llround(%.1f) = %lli\n",number, rnd2nearest);

  number=1.5;
  rnd2nearest = llround(number);
  printf ("llround(%.1f) = %lli\n",number, rnd2nearest);

  number=-2.5;
  rnd2nearest = llround(number);
  printf ("llround(%.1f) = %lli\n",number, rnd2nearest);
}
```

```
Output

Illustrates the llround() function

llround(501.1) = 501
llround(1.5) = 2
llround(-2.5) = -3
```

Related information

- "math.h" on page 44
- "ceil(), ceilf(), ceill() — Round up to integral value" on page 249
- "floor(), floorf(), floorl() — Round down to integral value" on page 548
- "lrint(), lrintf(), lrintl() and llrint(), llrintf(), llrintl() — Round the argument to the nearest integer" on page 1012
- "lround(), lroundf(), lroundl() — Round a decimal floating-point number to its nearest integer" on page 1016
- "nearbyint(), nearbyintf(), nearbyintl() — Round the argument to the nearest integer" on page 1132
- "rint(), rintf(), rintl() — Round to nearest integral value" on page 1450
- "round(), roundf(), roundl() — Round to the nearest integer" on page 1455
- "trunc(), truncf(), truncl() — Truncate an integer value" on page 1915

llroundd32(), llroundd64(), llroundd128() — Round to the nearest integer

Standards

Standards / Extensions	C or C++	Dependencies
C/C++ DFP	both	z/OS V1.8

Format

```
#define __STDC_WANT_DEC_FP__
#include <math.h>

long long int llroundd32(_Decimal32 x);
long long int llroundd64(_Decimal64 x);
long long int llroundd128(_Decimal128 x);

long long int llround(_Decimal32 x);   /* C++ only */
long long int llround(_Decimal64 x);   /* C++ only */
long long int llround(_Decimal128 x);  /* C++ only */
```

Compile requirement: Use of this function requires the long long data type. See *z/OS XL C/C++ Language Reference* for information on how to make long long available.

General description

The llround() family of functions round *x* to the nearest integer, rounding halfway cases away from zero, regardless of the current rounding mode.

Notes:

1. To use IEEE decimal floating-point, the hardware must have the Decimal Floating-Point Facility installed.
2. These functions work in IEEE decimal floating-point format. See "IEEE Decimal Floating-Point" for more information.

Returned value

If successful, they return the rounded integer. If the correct value is positive or negative and too large to represent as a long long, a domain error will occur and an unspecified value is returned.

Example

```
/* CELEBL21

   This example illustrates the llroundd32() function.

*/

#pragma strings(readonly)

#define __STDC_WANT_DEC_FP__
#include <fenv.h>
#include <math.h>
#include <stdio.h>

static void try_rm(int, _Decimal32);

/* pass back printable rounding mode */

static
char *rm_str(int rm)
{
  char *s = "undetermined";

  switch (rm)
  {
    case    FE_DEC_TONEAREST            :
      s =  "FE_DEC_TONEAREST"           ;  break;
    case    FE_DEC_TOWARDZERO           :
      s =  "FE_DEC_TOWARDZERO"          ;  break;
    case    FE_DEC_UPWARD               :
      s =  "FE_DEC_UPWARD"              ;  break;
    case    FE_DEC_DOWNWARD             :
      s =  "FE_DEC_DOWNWARD"            ;  break;
    case    FE_DEC_TONEARESTFROMZERO    :
      s =  "FE_DEC_TONEARESTFROMZERO"   ;  break;
    case   _FE_DEC_TONEARESTTOWARDZERO  :
      s = "_FE_DEC_TONEARESTTOWARDZERO" ;  break;
    case   _FE_DEC_AWAYFROMZERO         :
      s = "_FE_DEC_AWAYFROMZERO"        ;  break;
    case   _FE_DEC_PREPAREFORSHORTER    :
      s = "_FE_DEC_PREPAREFORSHORTER"   ;  break;
   }

   return s;
}


/* Try out one passed-in number with rounding mode */

static void try_rm(int rm, _Decimal32 d32)
{
  long long int ll;

  (void)fe_dec_setround(rm);

  ll = llroundd32(d32);

  printf("llroundd32(%+.2HF)  = %+lld - rounding mode = %s\n",
         d32 , ll, rm_str(rm)
        );
  return;
}


int main()
{
  try_rm( FE_DEC_TONEAREST          ,  501.50DF);
  try_rm( FE_DEC_TOWARDZERO         ,  501.50DF);
  try_rm( FE_DEC_UPWARD             , -501.51DF);
  try_rm( FE_DEC_DOWNWARD           , -501.49DF);
  try_rm( FE_DEC_TONEARESTFROMZERO  ,  500.50DF);
  try_rm(_FE_DEC_TONEARESTTOWARDZERO, -501.50DF);
  try_rm(_FE_DEC_AWAYFROMZERO       ,  500.49DF);
  try_rm(_FE_DEC_PREPAREFORSHORTER  ,  501.50DF);

  return 0;
}
```

Related information

- “math.h” on page 44
- “ceild32(), ceild64(), ceild128() — Round up to integral value” on page 250
- “floord32(), floord64(), floord128() — Round down to integral value” on page 549
- “llround(), llroundf(), llroundl() — Round to the nearest integer” on page 975
- “lrintd32(), lrintd64(), lrintd128() and llrintd32(), llrintd64(), llrintd128() — Round the argument to the nearest integer” on page 1014
- “lroundd32(), lroundd64(), lroundd128() — Round a floating-point number to its nearest integer” on page 1017
- “nearbyintd32(), nearbyintd64(), nearbyintd128() — Round the argument to the nearest integer” on page 1134
- “rintd32(), rintd64(), rintd128() — Round to nearest integral value” on page 1451
- “roundd32(), roundd64(), roundd128() — Round to the nearest integer” on page 1456
- “truncd32(), truncd64(), truncd128() — Truncate an integer value” on page 1916

lltoa() — Convert long long into a string

Standards

Standards / Extensions	C or C++	Dependencies
z/OS UNIX	both	z/OS V1R5

Format

```
#define _OPEN_SYS_ITOA_EXT
#include <stdlib.h>

char * lltoa(int64_t ll, char * buffer, int radix);
```

Compile requirement: Use of this function requires the long long data type. See *z/OS XL C/C++ Language Reference* for information on how to make long long available.

General description

The lltoa() function coverts the int64_t *ll* into a character string. The string is placed in the buffer passed, which must be large enough to hold the output. The radix values can be OCTAL, DECIMAL, or HEX. When the radix is DECIMAL, lltoa() produces the same result as the following statement:

```
(void) sprintf(buffer, "%lld", ll);
```

with buffer the returned character string. When the radix is OCTAL, lltoa() formats int64_t *ll* into an unsigned octal constant. When the radix is HEX, lltoa() formats int64_t *ll* into an unsigned hexadecimal constant. The hexadecimal value will include lower case abcdef, as necessary.

Returned value

String pointer (same as buffer) will be returned. When passed an invalid radix argument, function will return NULL and set errno to EINVAL.

Usage notes

1. This is a non-standard function. Even though the prototype given is commonly used by compilers on other platforms, there is no guarantee that this function will behave the same on all platforms, in all cases. You can use this function to help port applications from other platforms, but you should avoid using it when writing new applications, in order to ensure maximum portability.

Example

CELEBL30

```
/* CELEBL30

   This example reads an int64_t and formats it to decimal, unsigned
   octal, unsigned hexadecimal constants converted to a character
   string.

*/

#define _OPEN_SYS_ITOA_EXT
#include <stdio.h>
#include <stdlib.h>

int main ()
{
   int64_t i;
   char buffer [sizeof(int64_t)*8+1];
   printf ("Enter a number: ");
   if (scanf ("%lld",&i) == 1) {
      lltoa (i,buffer,DECIMAL);
      printf ("decimal: %s\n",buffer);
      lltoa (i,buffer,HEX);
      printf ("hexadecimal: %s\n",buffer);
      lltoa (i,buffer,OCTAL);
      printf ("octal: %s\n",buffer);
   }
   return 0;
}
```

Output

If the input is 1234, then the output should be:

```
decimal: 1234
hexadecimal: 4d2
octal: 2322
```

Related information

- "stdlib.h" on page 70
- "itoa() — Convert int into a string" on page 925
- "ltoa() — Convert long into a string" on page 1027
- "ulltoa() — Convert unsigned long long into a string" on page 1943
- "ultoa() — Convert unsigned long into a string" on page 1945
- "utoa() — Convert unsigned int into a string" on page 1972

localdtconv() — Date and time formatting convention inquiry

Standards

Standards / Extensions	C or C++	Dependencies
Language Environment	both	

Format

```
#include <locale.h>

struct dtconv *localdtconv(void);
```

General description

Determines the date/time format information of the current locale.

To avoid infringing on the user's name space, this nonstandard function has two names. One name is prefixed with two underscore characters, and one name is not. The name without the prefix underscore characters is exposed only when you use LANGLVL(EXTENDED).

To use this function, you must either invoke the function using its external entry point name (that is, the name that begins with two underscore characters), or compile with LANGLVL(EXTENDED). When you use LANGLVL(EXTENDED) any relevant information in the header is also exposed.

Returned value

Returns the address of the *dtconv* structure:

```
struct dtconv {
        char *abbrev_month_names[12];   /* Abbreviated month names      */
        char *month_names[12];          /* full month names             */
        char *abbrev_day_names[7];      /* Abbreviated day names        */
        char *day_names[7];             /* full day names               */
        char *date_time_format;         /* date and time format         */
        char *date_format;              /* date format                 */
        char *time_format;              /* time format                 */
        char *am_string;                /* AM string                   */
        char *pm_string;                /* PM string                   */
        char *time_format_ampm;         /* long date format            */
        char *iso_std8601_2000;         /* ISO 8601:2000 std date format*/
};
```

The dtconv structure is an IBM extension that stores values from the LC_TIME category of the current locale. It is initialized by the setlocale() function and copied to the user-supplied dtconv when localdtconv() is called.

The dtconv structure can be overwritten by subsequent calls to localdtconv() and setlocale() with LC_ALL or LC_TIME.

Related information

- Se the topic about internationalization of locales and character sets in *z/OS XL C/C++ Programming Guide*
- "locale.h" on page 40
- "time.h" on page 79
- "asctime(), asctime64() — Convert time to character string" on page 182
- "asctime_r(), asctime64_r() — Convert date and time to a character string" on page 184
- "ctime(), ctime64() — Convert time to character string" on page 359
- "ctime_r(), ctime64_r() — Convert time value to date and time character string" on page 362
- "gmtime(), gmtime64() — Convert time to broken-down UTC time" on page 810
- "gmtime_r(), gmtime64_r() — Convert a time value to broken-down UTC time" on page 812
- "localeconv() — Query numeric conventions"
- "localtime(), localtime64() — Convert time and correct for local time" on page 984
- "localtime_r(), localtime64_r() — Convert time value to broken-down local time" on page 986
- "mktime(), mktime64() — Convert local time" on page 1078
- "setlocale() — Set locale" on page 1552
- "strftime() — Convert to formatted time" on page 1743
- "time(),time64() — Determine current UTC time" on page 1878
- "tzset() — Set the time zone" on page 1936

localeconv() — Query numeric conventions

Standards

Standards / Extensions	C or C++	Dependencies
ISO C XPG4 XPG4.2 C99 Single UNIX Specification, Version 3	both	

Format

```
#include <locale.h>

struct lconv *localeconv(void);
```

General description

Sets the components of a structure having type `struct lconv` to values appropriate for the current locale. The structure may be overwritten by another call to localeconv() or by calling setlocale() and passing LC_ALL, LC_MONETARY, or LC_NUMERIC.

For a list of the elements in the `lconv` structure, see Table 6 on page 40.

Pointers to strings with a value of "" indicate that the value is not available in the C locale or is of 0 length. char types with a value of UCHAR_MAX indicate that the value is not available in the current locale.

Returned value

Returns a pointer to the structure.

Example

CELEBL06

```
/* CELEBL06

   This example prints out the default decimal point for your locale and
   then the decimal point for the Fr_CA locale.

 */
#include <stdio.h>
#include <locale.h>

int main(void)
{
  char * string;
  struct lconv * mylocale;
  mylocale = localeconv();
  /* Display default decimal point */
  printf( "Default decimal point is a %s\n",
          mylocale->decimal_point );

  if (NULL != (string = setlocale(LC_ALL, "Fr_CA.IBM-1047" )))
  {
     mylocale = localeconv();
     /* A comma is set to be the decimal point
        when the locale is Fr_CA.IBM-1047 */
  printf( "French-speaking Canadian decimal point is a %s\n",
          mylocale->decimal_point );
   }
   else {
      printf("setlocale(LC_ALL, Fr_CA.IBM-1047) returned <NULL>\n");
   }
   return 0;
}
```

Output

```
Default decimal-point is a .
French-speaking Canadian decimal-point is a ,
```

Related information

- See the topic about internationalization of locales and character sets in *z/OS XL C/C++ Programming Guide*
- “locale.h” on page 40
- “localdtconv() — Date and time formatting convention inquiry” on page 981

- "localtime(), localtime64() — Convert time and correct for local time"
- "localtime_r(), localtime64_r() — Convert time value to broken-down local time" on page 986
- "setlocale() — Set locale" on page 1552

localtime(), localtime64() — Convert time and correct for local time

Standards

Standards / Extensions	C or C++	Dependencies
ISO C POSIX.1 XPG4 XPG4.2 C99 Single UNIX Specification, Version 3 Language Environment	both	

Format

```
#include <time.h>

struct tm *localtime(const time_t *timeval);
```

```
#define _LARGE_TIME_API
#include <time.h>

struct tm *localtime64(const time64_t *timeval);
```

Compile requirement: Use of localtime64() function requires the long long data type. See *z/OS XL C/C++ Language Reference* for information on how to make long long available.

General description

The localtime() funtion converts the calendar time pointed to by *timeval* to a broken-down time expressed in local time. Calendar time is usually obtained by a call to the time() function.

The localtime64() function behaves exactly like localtime() except it will break down a time64_t value pointing to a calendar time beyond 03:14:07 UTC on January 19, 2038 with a limit of 23:59:59 UTC on December 31, 9999.

Usage notes

1. This function is sensitive to time zone information which is provided by:
   - The TZ environmental variable when POSIX(ON) and TZ is correctly defined, or by the _TZ environmental variable when POSIX(OFF) and _TZ is correctly defined.
   - The LC_TOD category of the current locale if POSIX(OFF) or TZ is not defined.

   The time zone external variables `tzname`, `timezone`, and `daylight` declarations remain feature test protected in time.h.
2. The ctime(), localtime(), and mktime() functions now return Coordinated Universal Time (UTC) unless customized locale information is made available, which includes setting the `timezone_name` variable.

3. In POSIX you can supply the necessary information by using environment variables.
4. In non-POSIX applications, you can supply customized locale information by setting time zone and daylight information in LC_TOD.
5. By customizing the locale, you allow the time functions to preserve both time and date, correctly adjusting for daylight time on a given date.
6. The gmtime() and localtime() functions may use a common, statically allocated structure for the conversion. Each call to one of these functions will alter the result of the previous call.
7. Calendar time returned by the time() function begins at the epoch, which was at 00:00:00 Coordinated Universal Time (UTC), January 1, 1970.
8. The localtime() function converts calendar time (that is, seconds elapsed since the epoch) to broken-down time, expressed as local time, using time zone information provided by the TZ or _TZ environment variable or the LC_TOD category of the current locale:
   - When neither TZ nor _TZ is defined, the current locale is interrogated for time zone information. If neither TZ nor _TZ is defined and LC_TOD time zone information is not present in the current locale, a default value is applied to local time. POSIX programs simply default to Coordinated Universal Time (UTC), while non-POSIX programs establish an offset from UTC based on the setting of the system clock. For more information about customizing a time zone to work with local time, see “Customizing a time zone” in *z/OS XL C/C++ Programming Guide*.

Returned value

Returns a pointer to a *tm* structure containing the broken-down time, expressed as a local time, and corresponding to the calendar time pointed to by *timeval*. If the calendar time cannot be converted, localtime() returns a NULL pointer. See “time.h” on page 79 for a description of the fields of the *tm* structure.

Error Code
Description

EOVERFLOW
The result cannot be represented.

Example

CELEBL07

```
/* CELEBL07

   This example queries the system clock and displays the local time.

 */
#include <time.h>
#include <stdio.h>

int main(void)
{
   struct tm *newtime;
   time_t ltime;

   time(&ltime);
   newtime = localtime(&ltime);
   printf("The date and time is %s", asctime(newtime));
}
```

Output

This output would occur if the local time is 3:00 p.m. June 16, 2006):

```
The date and time is Fri Jun 16 15:00:00 2006
```

Related information

- See the topic about internationalization of locales and character sets in *z/OS XL C/C++ Programming Guide* for a description of LC_TOD, which is a nonstandard, z/OS XL C/C++ proprietary locale category.
- "locale.h" on page 40
- "time.h" on page 79
- "asctime(), asctime64() — Convert time to character string" on page 182
- "asctime_r(), asctime64_r() — Convert date and time to a character string" on page 184
- "ctime(), ctime64() — Convert time to character string" on page 359
- "ctime_r(), ctime64_r() — Convert time value to date and time character string" on page 362
- "gmtime(), gmtime64() — Convert time to broken-down UTC time" on page 810
- "gmtime_r(), gmtime64_r() — Convert a time value to broken-down UTC time" on page 812
- "localdtconv() — Date and time formatting convention inquiry" on page 981
- "localtime_r(), localtime64_r() — Convert time value to broken-down local time"
- "mktime(), mktime64() — Convert local time" on page 1078
- "strftime() — Convert to formatted time" on page 1743
- "time(),time64() — Determine current UTC time" on page 1878
- "tzset() — Set the time zone" on page 1936

localtime_r(), localtime64_r() — Convert time value to broken-down local time

Standards

Standards / Extensions	C or C++	Dependencies
Single UNIX Specification, Version 2 Single UNIX Specification, Version 3 Language Environment	both	OS/390 V2R8

Format

```
#define _XOPEN_SOURCE 500
#include <time.h>

struct tm *localtime_r(const time_t *__restrict__ clock,
                       struct tm *__restrict__ result);
```

```
#define _LARGE_TIME_API
#include <time.h>

struct tm *localtime64_r(const time64_t *__restrict__ clock,
                       struct tm *__restrict__ result);
```

Compile requirement: Use of localtime64_r() function requires the long long data type. See *z/OS XL C/C++ Language Reference* for information on how to make long long available.

General description

The localtime_r() function converts the calendar time pointed to by *clock* into a broken-down time stored in the structure to which *result* points. The localtime_r() function also returns a pointer to that same structure.

Unlike localtime(), the reentrant version is not required to set tzname.

The localtime64_r() function behaves exactly like localtime_r() except it will break down a time64_t value pointing to a calendar time beyond 03:14:07 UTC on January 19, 2038 with a limit of 23:59:59 UTC on December 31, 9999.

Returned value

If successful, localtime_r() returns a pointer to the structure pointed to by the argument *result*.

If an error is detected, localtime() returns a null pointer and set errno to indicate the error.

Error Code
Description

EOVERFLOW
The result cannot be represented.

Related information

- See the topic about internationalization of locales and character sets in *z/OS XL C/C++ Programming Guide*
- "locale.h" on page 40
- "time.h" on page 79
- "asctime(), asctime64() — Convert time to character string" on page 182
- "asctime_r(), asctime64_r() — Convert date and time to a character string" on page 184
- "ctime(), ctime64() — Convert time to character string" on page 359
- "ctime_r(), ctime64_r() — Convert time value to date and time character string" on page 362
- "gmtime(), gmtime64() — Convert time to broken-down UTC time" on page 810
- "gmtime_r(), gmtime64_r() — Convert a time value to broken-down UTC time" on page 812
- "localdtconv() — Date and time formatting convention inquiry" on page 981
- "localtime(), localtime64() — Convert time and correct for local time" on page 984
- "mktime(), mktime64() — Convert local time" on page 1078
- "strftime() — Convert to formatted time" on page 1743
- "time(),time64() — Determine current UTC time" on page 1878
- "tzset() — Set the time zone" on page 1936

lockf() — Record locking on files

Standards

Standards / Extensions	C or C++	Dependencies
XPG4.2 Single UNIX Specification, Version 3	both	

Format

```
#define _XOPEN_SOURCE_EXTENDED 1
#include <unistd.h>

int lockf(int filedes, int function, off_t size);
```

General description

The lockf() function allows sections of a file to be locked with advisory-mode locks. Calls to lockf() from other processes which attempt to lock the locked file section will either return an error value or block until the section becomes unlocked. All the locks for a process are removed when the process terminates. Record locking with lockf() is supported for regular files.

The *filedes* argument is an open file descriptor. The file descriptor must have been opened with a write-only permission (O_WRONLY) or with read/write permission (O_RDWR) to establish a lock with this function.

The *function* argument is a control value which specifies the action to be taken. The permissible values for *function* are defined in <unistd.h> as follows:

Function	Description
F_ULOCK	unlock locked sections
F_LOCK	lock a section for exclusive use
F_TLOCK	test and lock a section for exclusive use
F_TEST	test a section for locks by other processes

F_TEST detects if a lock by another process is present on the specified section; F_LOCK and F_TLOCK both lock a section of a file if the section is available; F_ULOCK removes locks from a section of the file.

The *size* argument is the number of contiguous bytes to be locked or unlocked. The section to be locked or unlocked starts at the current offset in the file and extends forward for a positive size or backward for a negative size (the preceding bytes up to but not including the current offset). If *size* is 0, the section from the current offset through the largest possible file offset is locked (that is, from the current offset through the present or any future End Of File (EOF)). An area need not be allocated to the file to be locked because locks may exist past the End Of File.

The sections locked with F_LOCK or F_TLOCK may, in whole or in part, contain or be contained by a previously locked section for the same process. When this occurs, or if adjacent locked sections would occur, the sections are combined into a single locked section. If the request would cause the number of locks to exceed a system-imposed limit, the request will fail.

F_LOCK and F_TLOCK requests differ only by the action taken if the section is not available. F_LOCK blocks the calling process until the section is available. F_TLOCK makes the function fail if the section is already locked by another process.

File locks are released on first close by the locking process of any file descriptor for the file.

F_ULOCK requests may release (wholly or in part) one or more locked sections controlled by the process. Locked sections will be unlocked starting at the current file offset through *size* bytes or to the End Of File (EOF) if *size* is (off_t)0. When all of a locked section is not released (that is, when the beginning or end of the area to be unlocked falls within a locked section), the remaining portions of that section are still locked by the process. Releasing the center portion of a locked section will cause the remaining locked beginning and end portions to become two separate locked sections. If the request would cause the number of locks in the system to exceed a system-imposed limit, the request will fail.

A potential for deadlock occurs if a process controlling a locked section is blocked by accessing another process's locked section. If the system detects that a deadlock would occur, lockf() will fail with an EDEADLK error.

Locks obtained by lockf() are controlled by the same facility controlling locks obtained by fcntl().

The interaction between fcntl() and lockf() locks is unspecified.

Blocking on a section is interrupted by any signal.

Large file support for z/OS UNIX files: Large z/OS UNIX files are supported automatically for AMODE 64 C/C++ applications. AMODE 31 C/C++ applications must be compiled with the option LANGLVL(LONGLONG) and define the _LARGE_FILES feature test macro before any headers are included to enable this function to operate on z/OS UNIX files that are larger than 2 GB in size. File size and offset fields are enlarged to 63 bits in width. Therefore, any other function operating on the file is required to define the _LARGE_FILES feature test macro as well.

Returned value

If successful, lockf() returns 0.

If unsuccessful, existing locks are not changed, lockf() returns -1, and sets errno to one of the following values:

Error Code
: **Description**

EACCES or EAGAIN
: The *function* argument is F_TLOCK or F_TEST and the section is already locked by another process

EBADF
: The *filedes* argument is not a valid open file descriptor; or *function* is F_LOCK or F_TLOCK and *filedes* is not a valid file descriptor open for writing.

EDEADLK
: The *function* argument is F_LOCK and a deadlock is detected.

EINTR
: A signal was caught during execution of the function.

EOVERFLOW
: The offset of the first, or if size is not 0 then the last, byte in the requested section cannot be represented correctly in an object of type off_t.

Related information

- "unistd.h" on page 82

log(), logf(), logl() — Calculate natural logarithm

Standards

Standards / Extensions	C or C++	Dependencies
ISO C POSIX.1 XPG4 XPG4.2 ISO/ANSI C++ C99 Single UNIX Specification, Version 3 C++ TR1 C99	both	

Format

```
#include <math.h>

double log(double x);
float log(float x);                     /* C++ only */
long double log(long double x);         /* C++ only */
float logf(float x);
long double logl(long double x);
```

General description

Calculates the natural logarithm (base e) of *x*, for *x* greater than 0.

Note: These functions work in both IEEE Binary Floating-Point and hexadecimal floating-point formats. See "IEEE binary floating-point" on page 94 for more information about IEEE Binary Floating-Point.

Returned value

Returns the computed value.

If *x* is negative, the function sets errno to EDOM and returns -HUGE_VAL. If *x* is 0.0, the function returns -HUGE_VAL and sets errno to ERANGE. If the correct value would cause an underflow, 0 is returned and the value ERANGE is stored in errno.

Special behavior for IEEE: If *x* greater than 0, the function returns the natural logarithm (base e) of *x*.

If *x* is negative, the function sets errno to EDOM and returns NaNQ. If *x* is 0.0, the function returns -HUGE_VAL and errno remains unchanged.

Note: When environment variable _EDC_SUSV3 is set to 2, and if *x* is 0.0, the function returns -HUGE_VAL and sets errno to ERANGE.

Example

CELEBL08

```
/* CELEBL08

  This example calculates the natural logarithm of 1000.0.

 */
```

```
#include <math.h>
#include <stdio.h>

int main(void)
{
   double x = 1000.0, y;

   y = log(x);

   printf("The natural logarithm of %lf is %lf\n", x, y);
}
```

Output

```
The natural logarithm of 1000.000000 is 6.907755
```

Related information

- “math.h” on page 44
- “exp(), expf(), expl() — Calculate exponential function” on page 448
- “log10(), log10f(), log10l() — Calculate base 10 logarithm” on page 1000
- “pow(), powf(), powl() — Raise to power” on page 1193

logb(), logbf(), logbl() — Unbiased exponent

Standards

Standards / Extensions	C or C++	Dependencies
XPG4.2 C99 Single UNIX Specification, Version 3 C++ TR1 C99	both	z/OS V1R7 for logbf(), logbl()

Format

```
#define _XOPEN_SOURCE_EXTENDED 1
#include <math.h>

double logb(double x);
```

C99:

```
#define _ISOC99_SOURCE
#include <math.h>

float logbf(float x);
long double logbl(long double x);
```

C++ TR1 C99:

```
#define _TR1_C99
#include <math.h>

float logb(float x);
long double logb(long double x);
```

General description

Returns the exponent of its argument *x*, as a signed integer value in floating-point mode. If x is subnormal, it is treated as a normalized number.

Note: The following table shows the viable formats for these functions. See “IEEE binary floating-point” on page 94 for more information about IEEE Binary Floating-Point.

Function	Hex	IEEE
logb	X	X
logbf	X	X
logbl	X	X

Returned value

If successful, logb() returns the exponent of *x*.

logb() will fail under the following condition: If *x* is equal to 0.0, logb() will return -HUGE_VAL and set errno to EDOM.

Example

```
/*
 * This program illustrates the use of logb() function
 *
 */
#define _ISOC99_SOURCE
#include <math.h>
#include <float.h>  /* Needed for FLT_RADIX */
#include <stdio.h>


void main() {

  int i;
  union {
    double number;
    unsigned char  uchars [sizeof(double)];
  } dblval;
  double logbx;

  printf("Illustrates the logb() function");

  #ifdef __BFP__
    printf(" (IEEE version)\n\n");
  #else
    printf(" (HFP version)\n\n");
  #endif

  /* generate the smallest possible double number */
  for (i=0; i<sizeof(double); i++)
    dblval.uchars[i] = 0;
  dblval.uchars[1] = 0x10;

  logbx = logb(dblval.number);

  printf("x = %g\n",dblval.number);
  printf("logb(x) = %f\n\n", logbx);

  printf("pow(FLT_RADIX, logb(x) ) should equal x\n");
  printf("pow(%d,%f) = %g\n",FLT_RADIX, logbx, pow(FLT_RADIX, logbx));
}
```

```
Output

Illustrates the logb() function (IEEE version)
```

```
x = 2.22507e-308
logb(x) = -1022.000000

pow(FLT_RADIX, logb(x) ) should equal x
pow(2,-1022.000000) = 2.22507e-308
```

Related information

- “math.h” on page 44
- “ilogb(), ilogbf(), ilogbl() — Integer unbiased exponent” on page 835

logbd32(), logbd64(), logbd128() — Unbiased exponent

Standards

Standards / Extensions	C or C++	Dependencies
C/C++ DFP	both	z/OS V1.8

Format

```
#define __STDC_WANT_DEC_FP__
#include <math.h>

_Decimal32  logbd32(_Decimal32 x);
_Decimal64  logbd64(_Decimal64 x);
_Decimal128 logbd128(_Decimal128 x);

_Decimal32  logb(_Decimal32 x);     /* C++ only */
_Decimal64  logb(_Decimal64 x);     /* C++ only */
_Decimal128 logb(_Decimal128 x);    /* C++ only */
```

General description

Returns the unbiased exponent of its argument *x* as a signed integer value in decimal floating-point mode. For typical numbers, the value returned is the logarithm of |*x*| rounded down (toward -INF) to the nearest integer value.

Notes:

1. To use IEEE decimal floating-point, the hardware must have the Decimal Floating-Point Facility installed.
2. These functions work in IEEE decimal floating-point format. See "IEEE Decimal Floating-Point" for more information.

Returned value

If successful, these functions return the unbiased exponent of *x* as a signed integer value in decimal floating-point mode.

These functions will fail under the following condition: If *x* is equal to 0.0, -HUGE_VAL_D32, -HUGE_VAL_D64, or -HUGE_VAL_D128 is returned and errno is set to EDOM.

Example

```
/* CELEBL24

   This program illustrates the use of logbd32() function

*/
```

```
#define  __STDC_WANT_DEC_FP__
#include <math.h>
#include <stdio.h>
#include <stdlib.h>

void main()
{
  _Decimal32 x, logbx;

  printf("Illustrates the logbd32() function\n");


  /* Generate the smallest possible positive _Decimal32 number       */

  x = strtod32("0000001.E-101", NULL);

  logbx = logbd32(x);

  printf("x       = %Hg\n"  , x    );
  printf("logb(x) = %Hf\n\n", logbx);

  printf("powd32(10.0, logb32(x)) should equal x\n");
  printf("powd32(%Hf, %Hf) = %Hg\n",
         10.0DF, logbx, powd32(10.0DF, logbx));
}
```

Related information

- “math.h” on page 44
- “ilogbd32(), ilogbd64(), ilogbd128() — Integer unbiased exponent” on page 837
- “logb(), logbf(), logbl() — Unbiased exponent” on page 991

logd32(), logd64(), logd128() — Calculate natural logarithm

Standards

Standards / Extensions	C or C++	Dependencies
C/C++ DFP	both	z/OS V1.8

Format

```
#define __STDC_WANT_DEC_FP__
#include <math.h>

_Decimal32  logd32(_Decimal32 x);
_Decimal64  logd64(_Decimal64 x);
_Decimal128 logd128(_Decimal128 x);

_Decimal32  log(_Decimal32 x);    /* C++ only */
_Decimal64  log(_Decimal64 x);    /* C++ only */
_Decimal128 log(_Decimal128 x);   /* C++ only */
```

General description

Calculates the natural logarithm (base e) of *x*, for *x* greater than 0.

Notes:

1. To use IEEE decimal floating-point, the hardware must have the Decimal Floating-Point Facility installed.
2. These functions work in IEEE decimal floating-point format. See "IEEE Decimal Floating-Point" for more information.

Returned value

If *x* greater than 0, the function returns the natural logarithm (base e) of *x*.

If *x* is negative, the function sets errno to EDOM and returns NaNQ.

If *x* is 0.0, the function returns -HUGE_VAL_D32, -HUGE_VAL_D64, or -HUGE_VAL_D128 and errno remains unchanged.

Example

```
/* CELEBL22

  This example illustrates the logd64() function.

*/

#define  __STDC_WANT_DEC_FP__
#include <math.h>
#include <stdio.h>

int main(void)
{
   _Decimal64 x = 1000.0DD, y;

   y = logd64(x);

   printf("The natural logarithm of %Df is %Df\n", x, y);
}
```

Related information

- "math.h" on page 44
- "expd32(), expd64(), expd128() — Calculate exponential function" on page 449
- "log(), logf(), logl() — Calculate natural logarithm" on page 990
- "log10d32(), log10d64(), log10d128() — Calculate base 10 logarithm" on page 1002
- "powd32(), powd64(), powd128() — Raise to power" on page 1194

__login(), __login_applid() — Create a new security environment for process

Standards

Standards / Extensions	C or C++	Dependencies
z/OS UNIX	both	OS/390 V2R6

Format

```
#define  _OPEN_SYS
#include <unistd.h>

int __login(int   function_code,
            int   identity_type,
            int   identity_length,
            void  *identity,
            int   pass_length,
            char  *pass,
            int   certificate_length,
            char  *certificate,
            int   option_flags);
int __login_applid(int function_code,
    int identity_type,
    int identity_length,
    void *identity,
    int pass_length,
    char *pass,
    int certificate_length,
    char *certificate,
    int option_flags,
    const char *applid);
```

General description

The __login() function provides a way for a process to change its identity so as to be different than the address space identity and create a new security environment for the process. Once changed the process should not revert back to a previous identity and security environment. The following rules apply:

- Any single-threaded process can issue a __login to change its security environment.
- If the process is in a multiprocessing or multiple user environment and there is no task level security associated with the process, then the new security environment will be associated with the process.
- If the process is in a multiprocessing or multiple user environment and there is task level security associated with the process, then the old security environment will be replaced by the new security environment.

The __login_applid() function is equivalent to __login() with the added feature that it also allows the application identifier (APPLID) to be supplied that will be passed on to the security product to assist with authentication. This is useful, for example, in situations where a pass ticket is provided and the pass ticket was created with a USERID/APPLID combination. When applid is NULL or a pointer to NULL, no application identifier will be passed on to the security product.

The function has the following parameters:

Parameter
Description

function_code
Specifies the function. Specify __LOGIN_CREATE, as defined in the unistd.h header file, to create a process level security environment for the caller's process.

identity_type
Specifies the format of the the user identity being provided in **identity*. Specify __LOGIN_USERID, as defined in the unistd.h header file. The user ID identity is in the format of a 1-to-8-character userid and is passed as input.

identity_length
Specifies the length of the *identity* as defined by *identity_type*.

**identity*
Specifies the user identity as defined by *identity_type*.

pass_length
Specifies the length of the password or PassTicket, or the password phrase defined by *pass*.

**pass* Specifies a user password or PassTicket, or a password phrase.

certificate_length
Is not used presently and must be set to zero.

certificate
Is not used presently and must point to void.

option_flags
Specifies options used to tailor request. Must be set to 0.

applid Specifies the application identifier that will be used for authentication with the security product.

Usage notes

1. The intent of the __login() service is to provide a way for a process to change its identity so as to be different than the address space identity. The process should either terminate or select a new user ID, but should not try to revert back to the original identity. The user could issue the __login() again with the original user identity, but the task would retain its own security environment and not share the the security environment at the address space level.
2. A security product supporting multiprocessing or multiple user environment must be installed and operational.

Returned value

If successful, __login() returns 0.

If unsuccessful, __login() returns -1 and sets errno to one of the following values:

Error Code
Description

EACCES
Permission is denied.

EINVAL
A parameter is not valid. For example, length of applid exceeds 8 bytes.

EMVSERR
An MVS environmental error or internal occurred.

EMVSEXPIRE
The password or PassTicket, or a password phrase for the specified resource has expired.

EMVSSAF2ERR
An error occurred in the security product. The userid has been revoked or is unable to use the application.

ENOSYS
The function is not implemented.

EPERM
The operation is not permitted. Calling process may not be authorized in BPX.DAEMON facility class. The function is not supported in an address space where a load was done from an uncontrolled library. A required password or PassTicket, or a password phrase was not specified.

ESRCH
The USERID cannot become an OMVS process. The userid provided is not defined to the security product or doesn't have an OMVS segment defined.

Related information

- "unistd.h" on page 82

log1p(), log1pf(), log1pl() — Natural log of x+1

Standards

Standards / Extensions	C or C++	Dependencies
XPG4.2 C99 Single UNIX Specification, Version 3 C++ TR1 C99	both	

Format

```
#define _XOPEN_SOURCE_EXTENDED 1
#include <math.h>

double log1p(double x);
```

C99:

```
#define _ISOC99_SOURCE
#include <math.h>

float log1pf(float x);
long double log1pl(long double x);
```

C++ TR1 C99:

```
#define _TR1_C99
#include <math.h>

float log1p(float x);
long double log1p(long double x);
```

General description

Computes

$$\text{Log}_e (1.0 + x)$$

The value of x must be greater than -1.0.

Note: The following table shows the viable formats for these functions. See "IEEE binary floating-point" on page 94 for more information about IEEE Binary Floating-Point.

Function	Hex	IEEE
log1p	X	X
log1pf	X	X
log1pl	X	X

Returned value

If successful, log1p() returns the value of the above function of *x*.

log1p() will fail under the following conditions:
- If *x* is less than -1.0, log1p() will return -HUGE_VAL and set errno to EDOM.
- If *x* is equal to -1.0, log1p() will return -HUGE_VAL and set errno to ERANGE.

Special behavior for IEEE: If successful, log1p() returns the

$$\mathrm{Log}_e\,(1.0 + x)$$

The value of *x* must be greater than -1.0.

log1p() will fail under the following conditions:
- If *x* is less than -1.0, log1p() will return NaNQ and set errno to EDOM.
- If *x* is equal to -1.0, log1p() will return -HUGE_VAL and errno remains unchanged.

Note: When environment variable _EDC_SUSV3 is set to 2, and if *x* is equal to -1.0, the function returns -HUGE_VAL and sets errno to ERANGE.

Related information
- “math.h” on page 44
- “log(), logf(), logl() — Calculate natural logarithm” on page 990

log1pd32(), log1pd64(), log1pd128() — Natural log of x+1

Standards

Standards / Extensions	C or C++	Dependencies
C/C++ DFP	both	z/OS V1.11

Format

```
#define __STDC_WANT_DEC_FP__
#include <math.h>

_Decimal32  log1pd32(_Decimal32 x);
_Decimal64  log1pd64(_Decimal64 x);
_Decimal128 log1pd128(_Decimal128 x);

_Decimal32  log1p(_Decimal32 x);     /* C++ only */
_Decimal64  log1p(_Decimal64 x);     /* C++ only */
_Decimal128 log1p(_Decimal128 x);    /* C++ only */
```

General description

Computes

$$\mathrm{Log}_e\,(1.0 + x)$$

The value of x must be greater than -1.0.

Notes:

1. These functions work in IEEE decimal floating-point format. See “IEEE decimal floating-point” on page 95IEEE Decimal Floating-Point for more information.
2. To use IEEE decimal floating-point, the hardware must have the Decimal Floating-Point Facility installed.

Returned value

If successful, log1p() returns the Log (1.0 + *x*) The value of *x* must be greater than -1.0.

log1p() will fail under the following conditions:

- If *x* is less than -1.0, log1p() will return NaNQ and set errno to EDOM.
- If *x* is equal to -1.0, log1p() will return -HUGE_VAL_D32, -HUGE_VAL_D64 or -HUGE_VAL_D128 and errno remains unchanged.

Example

```
/* CELEBL27

   This exaxmple illustrates the log1pd128() function.

*/

#define __STDC_WANT_DEC_FP__
#include <math.h>
#include <stdio.h>
void main(void)
{
   _Decimal128 x, y;

   x = 23.2DL;
   y = log1pd128(x);

   printf("log1pd128( %DDf ) = %DDf\n", x , y);
}
```

Related information

- “math.h” on page 44
- “logd32(), logd64(), logd128() — Calculate natural logarithm” on page 994

log10(), log10f(), log10l() — Calculate base 10 logarithm

Standards

Standards / Extensions	C or C++	Dependencies
ISO C POSIX.1 XPG4 XPG4.2 ISO/ANSI C++ C99 Single UNIX Specification, Version 3 C++ TR1 C99	both	

Format

```
#include <math.h>

double log10(double x);
```

```
float log10(float x);                   /* C++ only */
long double log10(long double x);       /* C++ only */
float log10f(float x);
long double log10l(long double x);
```

General description

Calculates the base 10 logarithm of the positive value of *x*.

Note: These functions work in both IEEE Binary Floating-Point and hexadecimal floating-point formats. See "IEEE binary floating-point" on page 94 for more information about IEEE Binary Floating-Point.

Returned value

Returns the computed value.

If *x* is negative, the function sets errno to EDOM and returns -HUGE_VAL. If *x* is 0, the function returns -HUGE_VAL, and sets errno to ERANGE. If the correct value would cause an underflow, 0 is returned and the value ERANGE is stored in errno.

Special behavior for IEEE: If successful, the function returns the base 10 logarithm of the positive value of *x*.

If *x* is negative, the function sets errno to EDOM and returns NaNQ. If *x* is 0, the function returns -HUGE_VAL and errno remains unchanged.

Note: When environment variable _EDC_SUSV3 is set to 2, and if *x* is 0, the function returns -HUGE_VAL and sets errno to ERANGE.

Example

CELEBL09

```
/* CELEBL09

   This example calculates the base 10 logarithm of 1000.0.

 */
#include <math.h>
#include <stdio.h>

int main(void)
{
   double x = 1000.0, y;

   y = log10(x);

   printf("The base 10 logarithm of %lf is %lf\n", x, y);
}
```

Output

```
The base 10 logarithm of 1000.000000 is 3.000000
```

Related information

- "math.h" on page 44
- "exp(), expf(), expl() — Calculate exponential function" on page 448
- "log(), logf(), logl() — Calculate natural logarithm" on page 990
- "pow(), powf(), powl() — Raise to power" on page 1193

log10d32(), log10d64(), log10d128() — Calculate base 10 logarithm

Standards

Standards / Extensions	C or C++	Dependencies
C/C++ DFP	both	z/OS V1.8

Format

```
#define __STDC_WANT_DEC_FP__
#include <math.h>

_Decimal32  log10d32(_Decimal32 x);
_Decimal64  log10d64(_Decimal64 x);
_Decimal128 log10d128(_Decimal128 x);

_Decimal32  log10(_Decimal32 x);    /* C++ only */
_Decimal64  log10(_Decimal64 x);    /* C++ only */
_Decimal128 log10(_Decimal128 x);   /* C++ only */
```

General description

Calculates the base 10 logarithm of the positive value of *x*.

Notes:

1. To use IEEE decimal floating-point, the hardware must have the Decimal Floating-Point Facility installed.
2. These functions work in IEEE decimal floating-point format. See "IEEE Decimal Floating-Point" for more information.

Returned value

If successful, the function returns the base 10 logarithm of the positive value of *x*.

If *x* is negative, the function sets errno to EDOM and returns NaNQ.

If *x* is 0, the function returns -HUGE_VAL_D32, -HUGE_VAL_D64, or -HUGE_VAL_D128 and errno remains unchanged.

Example

```
/* CELEBL23

   This example illustrates the log10d128() function.

*/

#define  __STDC_WANT_DEC_FP__
#include <math.h>
#include <stdio.h>

int main(void)
{
   _Decimal128 x = 1000.0DL, y;

   y = log10d128(x);

   printf("The base 10 logarithm of %DDf is %DDf\n", x, y);
}
```

Related information

- “math.h” on page 44
- “expd32(), expd64(), expd128() — Calculate exponential function” on page 449
- “logd32(), logd64(), logd128() — Calculate natural logarithm” on page 994
- “log10(), log10f(), log10l() — Calculate base 10 logarithm” on page 1000
- “powd32(), powd64(), powd128() — Raise to power” on page 1194

log2(), log2f(), log2l() — Calculate the base-2 logarithm

Standards

Standards / Extensions	C or C++	Dependencies
C99 Single UNIX Specification, Version 3 C++ TR1 C99	both	z/OS V1R5

Format

```
#define _ISOC99_SOURCE
#include <math.h>

double log2(double x);
float log2f(float x);
long double log2l(long double x);
```

C++ TR1 C99:

```
#define _TR1_C99
#include <math.h>

float log2(float x);
long double log2(long double x);
```

General description

The log2 functions compute the base-2 logarithm of *x*.

Note: The following table shows the viable formats for these functions. See “IEEE binary floating-point” on page 94 for more information about IEEE Binary Floating-Point.

Function	Hex	IEEE
log2	X	X
log2f	X	X
log2l	X	X

Returned value

The log2 functions return log2 *x*.

A domain error occurs if *x* is less than zero. A range error may occur if *x* is zero.

Special behavior for IEEE: If *x* is equal to 0, a pole error will occur and errno remains unchanged.

When environment variable _EDC_SUSV3 is set to 2, and if *x* is equal to 0, the function returns -HUGE_VAL and sets errno to ERANGE.

Related information

- "math.h" on page 44

log2d32(), log2d64(), log2d128() — Calculate the base-2 logarithm

Standards

Standards / Extensions	C or C++	Dependencies
C/C++ DFP	both	z/OS V1.11

Format

```
#define __STDC_WANT_DEC_FP__
#include <math.h>

_Decimal32  log2d32(_Decimal32 x);
_Decimal64  log2d64(_Decimal64 x);
_Decimal128 log2d128(_Decimal128 x);

_Decimal32  log2(_Decimal32 x);     /* C++ only */
_Decimal64  log2(_Decimal64 x);     /* C++ only */
_Decimal128 log2(_Decimal128 x);    /* C++ only */
```

General description

The log2() functions compute the base-2 logarithm of *x*.

Notes:

1. These functions work in IEEE decimal floating-point format. See "IEEE decimal floating-point" on page 95IEEE Decimal Floating-Point for more information.
2. To use IEEE decimal floating-point, the hardware must have the Decimal Floating-Point Facility installed.

Returned value

The log2 functions return log2 *x*.

A domain error occurs if *x* is less than zero.

A range error may occur if *x* is zero.

Example

```
/* CELEBL28

   This exaxmple illustrates the log2d32() function.

*/

#define __STDC_WANT_DEC_FP__
#include <math.h>
#include <stdio.h>
void main(void)
{
   _Decimal32 x, y;

   x = 85.7DF;
   y = log2d32(x);

   printf("log2d32( %Hf ) = %Hf\n", x, y);
}
```

Related information

- "math.h" on page 44

longjmp() — Restore stack environment

Standards

Standards / Extensions	C or C++	Dependencies
ISO C POSIX.1 XPG4 XPG4.2 C99 C11 Single UNIX Specification, Version 3	both	

Format

```
#include <setjmp.h>

__noreturn__ void longjmp(jmp_buf env, int value);
```

General description

Restores a stack environment previously saved in *env* by setjmp(). The setjmp() and longjmp() functions provide a way to perform a nonlocal goto. They are often used in signal handlers.

A call to setjmp() causes the current stack environment to be saved in *env*. A subsequent call to longjmp() restores the saved environment, and returns control to a point in the program corresponding to the setjmp() call. Execution resumes as if the setjmp() call had just returned the given *value* of the value argument. All variables that are accessible to the function that receives control contain the values they had when longjmp() was called. The values of register variables are unpredictable. Nonvolatile *auto* variables that are changed between calls to setjmp() and longjmp() are also unpredictable.

Note: Ensure that the function that calls setjmp() does not return before you call the corresponding longjmp() function. Calling longjmp() after the function calling setjmp() returns causes unpredictable program behavior.

The *value* argument passed to longjmp() must be nonzero. If you give a 0 argument for *value*, longjmp() substitutes a 1 in its place.

Notes:

1. If longjmp() is used to jump back into an XPLink routine, any alloca() requests issued by the XPLink routine after the earlier setjmp() (or _setjmp(), sigsetjmp(), getcontext(), and so on) was called and before longjmp() is called are backed out. All storage obtained by these alloca() requests is freed before the XPLink routine is resumed.
2. If longjmp() is used to jump back into a non-XPLink routine, alloca() requests made after setjmp() (and so on) and before longjmp() are not backed out.

Special behavior for POSIX: In a POSIX program, the signal mask is *not* saved. Thus, to save and restore a stack environment that includes the current signal mask, use sigsetjmp() and siglongjmp() instead of setjmp() and longjmp(). The sigsetjmp()—siglongjmp() pair, the setjmp()—longjmp() pair, the _setjmp()—_longjmp() pair, and the getcontext()—setcontext() pair cannot be intermixed. A stack environment saved by setjmp() can be restored only by longjmp().

Special behavior for C++: If setjmp() and longjmp() are used to transfer control in a z/OS XL C++ program, the behavior in terms of the destruction of automatic objects is undefined. Additionally, if any automatic objects would be destroyed by a thrown exception transferring control to another (destination) point in the program, then a call to longjmp() at the throw point that transfers control to the same (destination) point has undefined behavior. This applies to both z/OS XL C++ and z/OS XL C/C++ ILC modules. The use of setjmp() and longjmp() in conjunction with try(), catch(), and throw() is also undefined.

Special behavior for XPG4.2: In a program that was compiled with the feature test macro, _XOPEN_SOURCE_EXTENDED, defined, another pair of functions, _setjmp()—_longjmp() are available. These functions are, on this implementation, functionally identical to setjmp()—longjmp(). Therefore it is possible, but not recommended, to intermix the setjmp()—longjmp() pair with the _setjmp()—_longjmp() pair.

Special behavior for XPLINK-compiled C++: Restrictions concerning setjmp.h and ucontext.h:

1. All XPLINK programs compiled with the Release 10 or later C compilers that are to run with Language Environment Release 10 or later libraries and use the **jmp_buf**, **sigjmp_buf** or **ucontext_t** types must not be compiled with C headers from Language Environment 2.9 or earlier.
2. Non-XPLINK functions compiled with any level of Language Environment headers must not define**jmp_buf**, **sigjmp_buf** or **ucontext_t** data items and pass them to XPLINK functions that call getcontext(), longjmp(), _longjmp(), setjmp(), _setjmp(), setcontext(), sigsetjmp(), or swapcontext() with these passed-in data items.
3. When __XPLINK__ is defined, the Release 10 and later headers define a larger **jmp_buf**, **sigjmp_buf** or **ucontext_t**area that is required by setjmp(), getcontext(), and related functions when they are called from an XPLINK routine. If __XPLINK__ is not defined, the Release 10 and later headers define a shorter **jmp_buf**, **sigjmp_buf** or **ucontext_t** area. The Language Environment headers before Release 10 also define the shorter version of these data areas. If an XPLINK function calls setjmp(), getcontext() or similar functions with a short **jmp_buf**, **sigjmp_buf** or **ucontext_t** area, a storage overlay or program check may occur when the C library tries to store past the end of the passed-in (too short) data area.

Returned value

longjmp() does not use the normal function call and return mechanisms; it returns no values.

Example

This example provides for saving the stack environment at this statement:

```
if(setjmp(mark) != 0) ...
```

When the system first performs the `if` statement, it saves the environment in *mark* and sets the condition to `FALSE` because setjmp() returns a 0 when it saves the environment. The program prints the message: `setjmp has been called`

The subsequent call to function *p* tests for a local error condition, which can cause it to perform the longjmp() function. Then, control returns to the original setjmp() function using the environment saved in *mark*. This time the condition is `TRUE` because -1 is the returned value from the longjmp() function. The example then performs the statements in the block and prints: `longjmp has been called`

It then performs the *recover* function and leaves the program.

```
/*    Illustration of longjmp().     */
#include <stdio.h>
#include <setjmp.h>

jmp_buf mark;

void p(void);
void recover(void);

int main(void)
{
   if (setjmp(mark) != 0)
   {
      printf("longjmp has been called\n");
      recover();
      exit(1);
      }
   printf("setjmp has been called\n");
⋮
   p();
⋮
}

void p(void)
{
   int error = 0;
⋮
   error = 9;
⋮
   if (error != 0)
      longjmp(mark, -1);
⋮
}

void recover(void)
{
⋮
}
```

Related information

- "setjmp.h" on page 62
- "getcontext() — Get user context" on page 694
- "_longjmp() — Nonlocal goto" on page 1008
- "setcontext() — Restore user context" on page 1524
- "setjmp() — Preserve stack environment" on page 1546
- "_setjmp() — Set jump point for a nonlocal goto" on page 1549
- "siglongjmp() — Restore the stack environment and signal mask" on page 1640
- "sigsetjmp() — Save stack environment and signal mask" on page 1659
- "swapcontext() — Save and restore user context" on page 1792

_longjmp() — Nonlocal goto

Standards

Standards / Extensions	C or C++	Dependencies
XPG4.2 Single UNIX Specification, Version 3	both	

Format

```
#define _XOPEN_SOURCE_EXTENDED 1
#include <setjmp.h>

void _longjmp(jmp_buf env, int value);
```

General description

The _longjmp() function restores a stack environment previously saved in *env* by _setjmp(). The _setjmp() and _longjmp() functions provide a way to perform a nonlocal *goto*. They are often used in signal handlers.

A call to _setjmp() causes the current stack environment to be saved in *env*.

A subsequent call to _longjmp() restores the saved environment and returns control to a point in the program corresponding to the _setjmp() call. Execution resumes as if the _setjmp() call had just returned the given *value* of the value argument. All variables that are accessible to the function that receives control contain the values they had when _longjmp() was called. The values of register variables are unpredictable. Nonvolatile *auto* variables that are changed between calls to _setjmp() and _longjmp() are also unpredictable.

The X/Open standard states that _longjmp() and _setjmp() are functionally identical to longjmp() and setjmp(), respectively, with the addition restriction that _longjmp() and _setjmp() do not manipulate the signal mask. However, on this implementation longjmp() and setjmp() do not manipulate the signal mask. So on this implementation _longjmp() and _setjmp() are literally identical to longjmp() and setjmp(), respectively.

To save and restore a stack environment, including the current signal mask, use sigsetjmp() and siglongjmp() instead of _setjmp() and _longjmp(), or setjmp() and longjmp().

The _setjmp()—_longjmp() pair, the setjmp()—longjmp() pair, the sigsetjmp()—siglongjmp() pair, and the getcontext()—setcontext() pair cannot be intermixed. A stack environment saved by _setjmp() can be restored only by _longjmp().

Notes:

1. However, on this implementation, since the _setjmp()—_longjmp() pair are functionally identical to the setjmp()—longjmp() pair it is possible to intermix them, but it is not recommended.
2. Ensure that the function that calls _setjmp() does not return before you call the corresponding _longjmp() function. Calling _longjmp() after the function calling _setjmp() returns causes unpredictable program behavior.
3. If _longjmp() is used to jump back into an XPLink routine, any alloca() requests issued by the XPLink routine after the earlier _setjmp() (or setjmp(), sigsetjmp(),

getcontext() and so on) was called and before _longjmp() is called are backed out. All storage obtained by these alloca() requests is freed before the XPLink routine is resumed.

4. If _longjmp() is used to jump back into a non-XPLink routine, alloca() requests made after _setjmp() (and so on) and before _longjmp() are not backed out.

The *value* argument passed to _longjmp() must be nonzero. If you give a zero argument for *value*, _longjmp() substitutes a 1 in its place.

env An address for a `jmp_buf` structure

value The return value from _setjmp()

Special behavior for C++: If _setjmp() and _longjmp() are used to transfer control in a z/OS XL C++ program, the behavior in terms of the destruction of automatic objects is undefined. Additionally, if any automatic objects would be destroyed by a thrown exception transferring control to another (destination) point in the program, then a call to _longjmp() at the throw point that transfers control to the same (destination) point has undefined behavior. This applies both to z/OS XL C++ and z/OS XL C/C++ ILC modules. The use of _setjmp() and _longjmp() in conjunction with try(), catch(), and throw() is also undefined.

Special behavior for XPLINK-compiled C++: Restrictions concerning setjmp.h and ucontext.h:

1. All XPLINK programs compiled with the Release 10 or later C compilers that are to run with Language Environment Release 10 or later libraries and use the **jmp_buf**, **sigjmp_buf** or **ucontext_t** types must not be compiled with C headers from Language Environment 2.9 or earlier.
2. Non-XPLINK functions compiled with any level of Language Environment headers must not define**jmp_buf**, **sigjmp_buf** or **ucontext_t** data items and pass them to XPLINK functions that call getcontext(), longjmp(), _longjmp(), setjmp(), _setjmp(), setcontext(), sigsetjmp(), or swapcontext() with these passed-in data items.
3. When __XPLINK__ is defined, the Release 10 and later headers define a larger **jmp_buf**, **sigjmp_buf** or **ucontext_t**area that is required by setjmp(), getcontext(), and related functions when they are called from an XPLINK routine. If __XPLINK__ is not defined, the Release 10 and later headers define a shorter **jmp_buf**, **sigjmp_buf** or **ucontext_t** area. The Language Environment headers before Release 10 also define the shorter version of these data areas. If an XPLINK function calls setjmp(), getcontext() or similar functions with a short **jmp_buf**, **sigjmp_buf** or **ucontext_t** area, a storage overlay or program check may occur when the C library tries to store past the end of the passed-in (too short) data area.

Returned value

_longjmp() does not use the normal function call and return mechanisms; it returns no values. When _longjmp() completes, program execution continues as if the corresponding invocation of _setjmp() had just returned the value specified by *value*.

Related information

- “setjmp.h” on page 62
- “getcontext() — Get user context” on page 694
- “longjmp() — Restore stack environment” on page 1005
- “setcontext() — Restore user context” on page 1524

- "setjmp() — Preserve stack environment" on page 1546
- "_setjmp() — Set jump point for a nonlocal goto" on page 1549
- "siglongjmp() — Restore the stack environment and signal mask" on page 1640
- "sigsetjmp() — Save stack environment and signal mask" on page 1659
- "swapcontext() — Save and restore user context" on page 1792

lrand48() — Pseudo-random number generator

Standards

Standards / Extensions	C or C++	Dependencies
XPG4 XPG4.2 Single UNIX Specification, Version 3	both	

Format

```
#define _XOPEN_SOURCE
#include <stdlib.h>

long int lrand48(void);
```

General description

The drand48(), erand48(), jrand48(), lrand48(), mrand48() and nrand48() functions generate uniformly distributed pseudo-random numbers using a linear congruential algorithm and 48-bit integer arithmetic.

The functions drand48() and erand48() return nonnegative, double-precision, floating-point values, uniformly distributed over the interval [0.0,1.0).

The functions lrand48() and nrand48() return nonnegative, long integers, uniformly distributed over the interval [0,2**31).

The functions mrand48() and jrand48() return signed long integers, uniformly distributed over the interval [-2**31,2**31).

The lrand48() function generates the next 48-bit integer value in a sequence of 48-bit integer values, X(i), according to the linear congruential formula:

```
X(n+1) = (aX(n) + c)mod(2**48)     n>=0
```

The initial values of X, a, and c are:

```
X(0) = 1
a   = 5deece66d (base 16)
c   = b         (base 16)
```

C/370 provides storage to save the most recent 48-bit integer value of the sequence, X(i). This storage is shared by the drand48(), lrand48() and mrand48() functions. The value, X(n), in this storage may be reinitialized by calling the lcong48(), seed48() or srand48() function. Likewise, the values of a and c, may be changed by calling the lcong48() function. Thereafter, whenever the seed48() or srand48() function is called to change X(n), the initial values of a and c are also reestablished.

Special behavior for z/OS UNIX Services: You can make the lrand48() function and other functions in the drand48 family thread-specific by setting the environment variable _RAND48 to the value THREAD before calling any function in the drand48 family.

If you do not request thread-specific behavior for the drand48 family, C/370 serializes access to the storage for X(n), a and c by functions in the drand48 family when they are called by a multithreaded application.

If thread-specific behavior is requested, and the lrand48() function is called from thread t, the lrand48() function generates the next 48-bit integer value in a sequence of 48-bit integer values, X(t,i), for the thread t. The sequence of values for a thread is generated according to the linear congruential formula:

```
X(t,n+1) = (a(t)X(t,n) + c(t))mod(2**48)     n>=0
```

The initial values of X(t), a(t) and c(t) for the thread t are:

```
X(t,0) = 1
a(t)   = 5deece66d (base 16)
c(t)   = b         (base 16)
```

C/370 provides storage which is specific to the thread t to save the most recent 48-bit integer value of the sequence, X(t,i), generated by the drand48(), lrand48() or mrand48() function. The value, X(t,n), in this storage may be reinitialized by calling the lcong48(), seed48() or srand48() function from the thread t. Likewise, the values of a(t) and c(t) for thread t may be changed by calling the lcong48() function from the thread. Thereafter, whenever the seed48() or srand48() function is called from the thread t to change X(t,n), the initial values of a(t) and c(t) are also reestablished.

Returned value

lrand48() transforms the generated 48-bit value, X(n+1), to a nonnegative, long integer value on the interval [0,2**31) and returns this transformed value.

Special behavior for z/OS UNIX Services: If thread-specific behavior is requested for the drand48 family and lrand48() is called on thread t, lrand48() transforms the generated 48-bit value, X(t,n+1), to a nonnegative, long integer value on the interval [0,2**31) and returns this transformed value.

Related information

- “stdlib.h” on page 70
- “drand48() — Pseudo-random number generator” on page 402
- “erand48() — Pseudo-random number generator” on page 426
- “jrand48() — Pseudo-random number generator” on page 928
- “lcong48() — Pseudo-random number initializer” on page 939
- “mrand48() — Pseudo-random number generator” on page 1101
- “nrand48() — Pseudo-random number generator” on page 1148
- “seed48() — Pseudo-random number initializer” on page 1467
- “srand48() — Pseudo-random number initializer” on page 1718

lrint(), lrintf(), lrintl() and llrint(), llrintf(), llrintl() — Round the argument to the nearest integer

Standards

Standards / Extensions	C or C++	Dependencies
C99 Single UNIX Specification, Version 3 C++ TR1 C99	both	z/OS V1R7

Format

```
#define _ISOC99_SOURCE
#include <math.h>

long int lrint(double x);
long int lrintf(float x);
long int lrintl(long double x);

long long int llrint(double x);
long long int llrintf(float x);
long long int llrintl(long double x);
```

C++ TR1 C99:

```
#define _TR1_C99
#include <math.h>

long int lrint(float x);
long int lrint(long double x);
long long int llrint(float x);
long long int llrint(long double x);
```

Compile requirement: The llrint() family of functions requires the long long data type. See *z/OS XL C/C++ Language Reference* for information on how to make long long available.

General description

The lrint() and llrint() families of functions round their argument to the nearest integer value according to the current rounding mode. If the rounded value is outside the range of the return type, the numeric result is unspecified. A range error may occur if the magnitude of *x* is too large.

Note: The following table shows the viable formats for these functions. See "IEEE binary floating-point" on page 94 for more information about IEEE Binary Floating-Point.

Function	Hex	IEEE
lrint	X	X
lrintf	X	X
lrintl	X	X
llrint	X	X
llrintf	X	X
llrintl	X	X

Returned value

If successful, they return the rounded integer value. If the correct value is positive or negative and too large to represent as a **long** (lrint() family) or **long long** (llrint() family), a domain error will occur and an unspecified value is returned.

Example

```
/*
 * This program illustrates the use of lrint() function
 *
 * Note: To get the output shown in this book , this program
 *       should be compiled using FLOAT(IEEE)
 *
 */
#define _ISOC99_SOURCE
#include <math.h>
#include <stdio.h>
#include <_Ieee754.h>   /* save/get fpc functions   */

char *RoundStr (_FP_rmode_t rm_type) {
  char *RndStr="undetermined";
   switch (rm_type) {
     case (_RMODE_RN):
        RndStr="round to nearest";
        break;
     case (_RMODE_RZ):
        RndStr="round toward zero";
        break;
     case (_RMODE_RP):
        RndStr="round toward +infinity ";
        break;
     case (_RMODE_RM):
        RndStr="round toward -infinity ";
        break;
   }
   return (RndStr);
}

void main() {

  _FP_fpcreg_t save_rmode, current_rmode;
  long int     rnd2nearest;
  double       number=500.99;

  printf("Illustrates the lrint() function\n");
  __fpc_rd(&current_rmode);     /* get current rounding mode */

  rnd2nearest = lrint(number);
  printf ("When rounding direction is %s:\n lrint(%.2f) = %li\n",
          RoundStr(current_rmode.rmode), number, rnd2nearest);
  save_rmode.rmode = _RMODE_RZ;
  __fpc_sm(save_rmode.rmode);   /* set rounding mode to round to zero */

  rnd2nearest = lrint(number);
  printf ("When rounding direction is %s:\n lrint(%.2f) = %li\n",
          RoundStr(save_rmode.rmode), number, rnd2nearest);
}

Output

Illustrates the lrint() function
When rounding direction is round to nearest:
 lrint(500.99) = 501
When rounding direction is round toward zero:
 lrint(500.99) = 500
```

Related information

- “math.h” on page 44
- “ceil(), ceilf(), ceill() — Round up to integral value” on page 249
- “floor(), floorf(), floorl() — Round down to integral value” on page 548
- “llround(), llroundf(), llroundl() — Round to the nearest integer” on page 975
- “lround(), lroundf(), lroundl() — Round a decimal floating-point number to its nearest integer” on page 1016

- “nearbyint(), nearbyintf(), nearbyintl() — Round the argument to the nearest integer” on page 1132
- “rint(), rintf(), rintl() — Round to nearest integral value” on page 1450
- “round(), roundf(), roundl() — Round to the nearest integer” on page 1455
- “trunc(), truncf(), truncl() — Truncate an integer value” on page 1915

lrintd32(), lrintd64(), lrintd128() and llrintd32(), llrintd64(), llrintd128() — Round the argument to the nearest integer

Standards

Standards / Extensions	C or C++	Dependencies
C/C++ DFP	both	z/OS V1.8

Format

```
#define __STDC_WANT_DEC_FP__
#include <math.h>

long int lrintd32(_Decimal32 x);
long int lrintd64(_Decimal64 x);
long int lrintd128(_Decimal128 x);

long int lrint(_Decimal32 x);   /* C++ only */
long int lrint(_Decimal64 x);   /* C++ only */
long int lrint(_Decimal128 x);  /* C++ only */

long long int llrintd32(_Decimal32 x);
long long int llrintd64(_Decimal64 x);
long long int llrintd128(_Decimal128 x);

long long int llrint(_Decimal32 x);   /* C++ only */
long long int llrint(_Decimal64 x);   /* C++ only */
long long int llrint(_Decimal128 x);  /* C++ only */
```

Compile requirment: Use of this function requires the long long data type. See *z/OS XL C/C++ Language Reference* for information on how to make long long available.

General description

The lrint() and llrint() families of functions round their argument to the nearest integer value according to the current rounding mode. If the rounded value is outside the range of the return type, the numeric result is unspecified. A range error may occur if the magnitude of *x* is too large.

Notes:

1. To use IEEE decimal floating-point, the hardware must have the Decimal Floating-Point Facility installed.
2. These functions work in IEEE decimal floating-point format. See "IEEE Decimal Floating-Point" for more information.

Returned value

If successful, they return the rounded integer value. If the correct value is positive or negative and too large to represent as a long (lrint() family) or long long (llrint() family), a domain error will occur and an unspecified value is returned.

Example

```
/* CELEBL20

   This example illustrates the lrintd64() and llrintd128() functions.

*/

#pragma strings(readonly)

#define __STDC_WANT_DEC_FP__
#include <fenv.h>
#include <math.h>
#include <stdio.h>

static void try_rm(int);
/* pass back printable rounding mode */

static
char *rm_str(int rm)
{
  char *s = "undetermined";

  switch (rm)
  {
    case    FE_DEC_TONEAREST            :
      s =  "FE_DEC_TONEAREST"           ;  break;
    case    FE_DEC_TOWARDZERO           :
      s =  "FE_DEC_TOWARDZERO"          ;  break;
    case    FE_DEC_UPWARD               :
      s =  "FE_DEC_UPWARD"              ;  break;
    case    FE_DEC_DOWNWARD             :
      s =  "FE_DEC_DOWNWARD"            ;  break;
    case    FE_DEC_TONEARESTFROMZERO    :
      s =  "FE_DEC_TONEARESTFROMZERO"   ;  break;
    case   _FE_DEC_TONEARESTTOWARDZERO  :
      s = "_FE_DEC_TONEARESTTOWARDZERO" ;  break;
    case   _FE_DEC_AWAYFROMZERO         :
      s = "_FE_DEC_AWAYFROMZERO"        ;  break;
    case   _FE_DEC_PREPAREFORSHORTER    :
      s = "_FE_DEC_PREPAREFORSHORTER"   ;  break;
   }

   return s;
}


/* Try out one passed-in number with rounding mode */

static void try_rm(int rm)
{
  long int      l;
  long long int ll;
  _Decimal64    d64  = 500.01DD;
  _Decimal128   d128 = 500.99DL;

  (void)fe_dec_setround(rm);

  l  =  lrintd64( d64 );
  ll = llrintd128(d128);

  printf(" lrintd64( %.2DF) = %ld - rounding mode = %s\n",
         d64 , l, rm_str(rm)
        );
  printf("llrintd128(%.2DDF) = %lld - rounding mode = %s\n",
         d128, ll, rm_str(rm)
        );
  return;
}


int main()
{
  try_rm( FE_DEC_TONEAREST          );
  try_rm( FE_DEC_TOWARDZERO         );
  try_rm( FE_DEC_UPWARD             );
  try_rm( FE_DEC_DOWNWARD           );
  try_rm( FE_DEC_TONEARESTFROMZERO  );
  try_rm(_FE_DEC_TONEARESTTOWARDZERO);
  try_rm(_FE_DEC_AWAYFROMZERO       );
  try_rm(_FE_DEC_PREPAREFORSHORTER  );

  return 0;
}
```

Related information

- “math.h” on page 44
- “ceild32(), ceild64(), ceild128() — Round up to integral value” on page 250
- “floord32(), floord64(), floord128() — Round down to integral value” on page 549
- “llroundd32(), llroundd64(), llroundd128() — Round to the nearest integer” on page 977
- “lroundd32(), lroundd64(), lroundd128() — Round a floating-point number to its nearest integer” on page 1017
- “lrint(), lrintf(), lrintl() and llrint(), llrintf(), llrintl() — Round the argument to the nearest integer” on page 1012
- “nearbyintd32(), nearbyintd64(), nearbyintd128() — Round the argument to the nearest integer” on page 1134
- “rintd32(), rintd64(), rintd128() — Round to nearest integral value” on page 1451
- “roundd32(), roundd64(), roundd128() — Round to the nearest integer” on page 1456
- “truncd32(), truncd64(), truncd128() — Truncate an integer value” on page 1916

lround(), lroundf(), lroundl() — Round a decimal floating-point number to its nearest integer

Standards

Standards / Extensions	C or C++	Dependencies
C99 Single UNIX Specification, Version 3 C++ TR1 C99	both	z/OS V1R5

Format

```
#define _ISOC99_SOURCE
#include <math.h>

long int lround(double x);
long int lroundf(float x);
long int lroundl(long double x);
```

C++ TR1 C99:

```
#define _TR1_C99
#include <math.h>

long lround(float x);
long lround(long double x);
```

General description

The lround functions round x to the nearest integer value, rounding halfway cases away from zero, regardless of the current rounding mode.

Note: The following table shows the viable formats for these functions. See "IEEE binary floating-point" on page 94 for more information about IEEE Binary Floating-Point.

Function	Hex	IEEE
lround	X	X
lroundf	X	X
lroundl	X	X

Returned value

The lround functions return the rounded integer value of *x*.

If the rounded value is outside the range of the return type, the numeric result is unspecified. A range error may occur if the magnitude of *x* is too large.

Related information

- "math.h" on page 44

lroundd32(), lroundd64(), lroundd128() — Round a floating-point number to its nearest integer

Standards

Standards / Extensions	C or C++	Dependencies
C/C++ DFP	both	z/OS V1.8

Format

```
#define __STDC_WANT_DEC_FP__
#include <math.h>

long int lroundd32(_Decimal32 x);
long int lroundd64(_Decimal64 x);
long int lroundd128(_Decimal128 x);

long int lround(_Decimal32 x);     /* C++ only */
long int lround(_Decimal64 x);     /* C++ only */
long int lround(_Decimal128 x);    /* C++ only */
```

General description

The lround functions round x to the nearest integer value, rounding halfway cases away from zero, regardless of the current rounding mode.

Notes:

1. To use IEEE decimal floating-point, the hardware must have the Decimal Floating-Point Facility installed.
2. These functions work in IEEE decimal floating-point format. See "IEEE Decimal Floating-Point" for more information.

Returned value

The lround functions return the rounded integer value of x.

If the rounded value is outside the range of the return type, the numeric result is unspecified. A range error may occur if the magnitude of x is too large.

Example

```
/* CELEBL21

   This example illustrates the llroundd32() function.

*/

#pragma strings(readonly)

#define __STDC_WANT_DEC_FP__
#include <fenv.h>
#include <math.h>
#include <stdio.h>

static void try_rm(int, _Decimal32);

/* pass back printable rounding mode */

static
char *rm_str(int rm)
{
  char *s = "undetermined";

  switch (rm)
  {
    case    FE_DEC_TONEAREST            :
      s =  "FE_DEC_TONEAREST"           ;  break;
    case    FE_DEC_TOWARDZERO           :
      s =  "FE_DEC_TOWARDZERO"          ;  break;
    case    FE_DEC_UPWARD               :
      s =  "FE_DEC_UPWARD"              ;  break;
    case    FE_DEC_DOWNWARD             :
      s =  "FE_DEC_DOWNWARD"            ;  break;
    case    FE_DEC_TONEARESTFROMZERO    :
      s =  "FE_DEC_TONEARESTFROMZERO"   ;  break;
    case   _FE_DEC_TONEARESTTOWARDZERO  :
      s = "_FE_DEC_TONEARESTTOWARDZERO" ;  break;
    case   _FE_DEC_AWAYFROMZERO         :
      s = "_FE_DEC_AWAYFROMZERO"        ;  break;
    case   _FE_DEC_PREPAREFORSHORTER    :
      s = "_FE_DEC_PREPAREFORSHORTER"   ;  break;
   }

   return s;
}


/* Try out one passed-in number with rounding mode */

static void try_rm(int rm, _Decimal32 d32)
```

```
{
  long long int ll;

  (void)fe_dec_setround(rm);

  ll = llroundd32(d32);

  printf("llroundd32(%+.2HF)  = %+lld - rounding mode = %s\n",
         d32 , ll, rm_str(rm)
        );
  return;
}


int main()
{
  try_rm( FE_DEC_TONEAREST          ,  501.50DF);
  try_rm( FE_DEC_TOWARDZERO         ,  501.50DF);
  try_rm( FE_DEC_UPWARD             , -501.51DF);
  try_rm( FE_DEC_DOWNWARD           , -501.49DF);
  try_rm( FE_DEC_TONEARESTFROMZERO  ,  500.50DF);
  try_rm(_FE_DEC_TONEARESTTOWARDZERO, -501.50DF);
  try_rm(_FE_DEC_AWAYFROMZERO       ,  500.49DF);
  try_rm(_FE_DEC_PREPAREFORSHORTER  ,  501.50DF);

  return 0;
}
```

Related information

- "math.h" on page 44
- "lround(), lroundf(), lroundl() — Round a decimal floating-point number to its nearest integer" on page 1016

lsearch() — Linear search and update

Standards

Standards / Extensions	C or C++	Dependencies
XPG4 XPG4.2 Single UNIX Specification, Version 3	both	

Format

```
#define _XOPEN_SOURCE
#include <search.h>

void *lsearch(const void *key, void *base, size_t *nelp,
              size_t width,
              int (*compar)(const void *, const void *));
```

General description

The lsearch() function is a linear search routine. It returns a pointer into a table indicating where an entry may be found. If the entry does not occur, it is added at the end of the table. The *key* argument points to the entry to be sought in the table. The base argument points to the first element in the table. The *width* argument is the size of an element in bytes. The *nelp* argument points to an integer containing the current number of elements in the table. The integer to which *nelp* points is incremented if the entry is added to the table. The *compar* argument points to a comparison function which the user must supply (strcmp(), for example). It is called with two arguments that point to the elements being compared. The function must return 0 if the elements are equal and nonzero otherwise.

Special behavior for C++: Because C and C++ linkage conventions are incompatible, lsearch() cannot receive a C++ function pointer as the comparator argument. If you attempt to pass a C++ function pointer to lsearch(), the compiler will flag it as an error. You can pass a C or C++ function to lsearch() by declaring it as extern "C".

Returned value

If the searched for entry is found, lsearch() returns a pointer to it.

If not found, lsearch() returns a pointer to the newly added element. A NULL pointer is returned in case of error.

No errors are defined.

Related information

- “bsearch() — Search arrays” on page 220
- “hsearch() — Search hash tables” on page 817
- “lfind() — Linear search routine” on page 963
- “tsearch() — Binary tree search” on page 1920

lseek() — Change the offset of a file

Standards

Standards / Extensions	C or C++	Dependencies
POSIX.1 XPG4 XPG4.2 Single UNIX Specification, Version 3	both	

Format

```
#define _POSIX_SOURCE
#include <unistd.h>

off_t lseek(int fildes, off_t offset, int pos);
```

General description

Changes the current file offset to a new position in a z/OS UNIX file. The new position is the given byte *offset* from the position specified by *pos*. After you have used lseek() to seek to a new location, the next I/O operation on the file begins at that location.

lseek() lets you specify new file offsets past the current end of the file. If data is written at such a point, read operations in the gap between this data and the old end of the file will return bytes containing zeros. (In other words, the gap is assumed to be filled with zeros.)

Seeking past the end of a file, however, does not automatically extend the length of the file. There must be a write operation before the file is actually extended.

Special behavior for POSIX C: For character special files, lseek() sets the file offset to the specified value. z/OS UNIX services ignore the file offset value during the read/write processing to character special files.

int *fildes***;**
The file whose current file offset you want to change.

off_t *offset***;**
The amount (positive or negative) the byte offset is to be changed. The sign indicates whether the offset is to be moved forward (positive) or backward (negative).

int *pos***;**
One of the following symbols (defined in the unistd.h header file):

SEEK_SET
The start of the file

SEEK_CUR
The current file offset in the file

SEEK_END
The end of the file

Large file support for z/OS UNIX files: Large z/OS UNIX files are supported automatically for AMODE 64 C/C++ applications. AMODE 31 C/C++ applications must be compiled with the option LANGLVL(LONGLONG) and define the _LARGE_FILES feature test macro before any headers are included to enable this function to operate on z/OS UNIX files that are larger than 2 GB in size. File size and offset fields are enlarged to 63 bits in width. Therefore, any other function operating on the file is required to define the _LARGE_FILES feature test macro as well.

Returned value

If successful, lseek() returns the new file offset, measured in bytes from the beginning of the file.

If unsuccessful, lseek() returns -1 and sets errno to one of the following values:

Error Code
Description

EBADF
fildes is not a valid open file descriptor.

EINVAL
pos contained something other than one of the three options, or the combination of the *pos* values would have placed the file offset before the beginning of the file.

EOVERFLOW
The resulting file offset would be a value which cannot be represented correctly in an object of type off_t.

ESPIPE
fildes is associated with a pipe or FIFO special file.

Example

This fragment positions a file (that has at least 10 bytes) to an offset of 10 bytes before the end of the file.

```
lseek(fildes,-10,SEEK_END);
```

Related information

- "unistd.h" on page 82
- "creat() — Create a new file or rewrite an existing one" on page 340
- "dup() — Duplicate an open file descriptor" on page 404
- "fcntl() — Control open file descriptors" on page 474
- "fseek() — Change file position" on page 639
- "fsetpos() — Set file position" on page 649
- "open() — Open a file" on page 1152
- "read() — Read from a file or socket" on page 1376
- "sigaction() — Examine or change a signal action" on page 1611
- "write() — Write data on a file or socket" on page 2085

lstat() — Get status of file or symbolic link

Standards

Standards / Extensions	C or C++	Dependencies
z/OS UNIX XPG4.2 Single UNIX Specification, Version 3	both	

Format

```
#define _POSIX1_SOURCE 2
#include <sys/stat.h>

int lstat(const char *__restrict__ pathname, struct stat *__restrict__ buf);
```

General description

Gets status information about a specified file and places it in the area of memory pointed to by the *buf* argument. You do not need permissions on the file itself, but you must have search permission on all directory components of the *pathname*.

If the named file is a symbolic link, lstat() returns information about the symbolic link itself.

The information is returned in the following stat structure, defined in the sys/stat.h header file.

Table 43. Elements of stat Structure

Structure	Description
mode_t st_mode	A bit string indicating the permissions and privileges of the file. Symbols are defined in the sys/stat.h header file to refer to bits in a mode_t value; these symbols are listed in "chmod() — Change the mode of a file or directory" on page 272.
ino_t st_ino	The serial number of the file.
dev_t st_dev	The numeric ID of the device containing the file.
nlink_t st_nlink	The number of links to the file.

Table 43. Elements of stat Structure (continued)

Structure	Description
uid_t st_uid	The numeric user ID of the file's owner.
gid_t st_gid	The numeric group ID of the file's group.
off_t st_size	For regular files, the file's size in bytes. For symbolic links, the length of the pathname contained therein not counting the trailing NULL. For other kinds of files, the value of this field is unspecified.
time_t st_atime	The most recent time the file was accessed.
time_t st_ctime	The most recent time the status of the file was changed.
time_t st_mtime	The most recent time the contents of the file were changed.

Values for `time_t` are given in terms of seconds that have elapsed since epoch.

If the named file is a symbolic link, lstat() updates the time-related fields before putting information in the `stat` structure.

You can examine properties of a `mode_t` value from the `st_mode` field by using a collection of macros defined in the sys/modes.h header file. If *mode* is a `mode_t` value, and *genvalue* is an `unsigned int` value from the stat structure, then:

S_ISBLK(*mode***)**
Is nonzero for block special files.

S_ISCHR(*mode***)**
Is nonzero for character special files.

S_ISDIR(*mode***)**
Is nonzero for directories.

S_ISEXTL(*mode,genvalue***)**
Is nonzero for external links.

S_ISFIFO(*mode***)**
Is nonzero for pipes and FIFO special files.

S_ISLNK(*mode***)**
Is nonzero for symbolic links.

S_ISREG(*mode***)**
Is nonzero for regular files.

S_ISSOCK(*mode***)**
Is nonzero for sockets.

If lstat() successfully determines all this information, it stores it in the area indicated by the *buf* argument.

Large file support for z/OS UNIX files: Large z/OS UNIX files are supported automatically for AMODE 64 C/C++ applications. AMODE 31 C/C++ applications must be compiled with the option LANGLVL(LONGLONG) and define the _LARGE_FILES feature test macro before any headers are included to enable this function to operate on z/OS UNIX files that are larger than 2 GB in size. File size and offset fields are enlarged to 63 bits in width. Therefore, any other function operating on the file is required to define the _LARGE_FILES feature test macro as well.

Returned value

If successful, lstat() returns 0.

If unsuccessful, lstat() returns -1 and sets errno to one of the following values:

Error Code
Description

EACCES
The process does not have search permission on some component of the *pathname* prefix.

EINVAL
buf contains a NULL.

EIO
Added for XPG4.2: An I/O error occurred while reading from the file system.

ELOOP
A loop exists in symbolic links. This error is issued if the number of symbolic links encountered during resolution of the *pathname* argument is greater than POSIX_SYMLOOP.

ENAMETOOLONG
pathname is longer than **PATH_MAX** characters or some component of *pathname* is longer than **NAME_MAX** characters while _POSIX_NO_TRUNC is in effect. For symbolic links, the length of the pathname string substituted for a symbolic link exceeds **PATH_MAX**. The **PATH_MAX** and **NAME_MAX** values can be determined through pathconf().

ENOENT
There is no file named *pathname*, or *pathname* is an empty string.

ENOTDIR
A component of the *pathname* prefix is not a directory.

EOVERFLOW
The file size in bytes or the number of blocks allocated to the file or the file serial number cannot be represented correctly in the structure pointed to by buf.

Note: Starting with z/OS V1.9, environment variable _EDC_EOVERFLOW can be used to control behavior of lstat() with respect to detecting an EOVERFLOW condition for z/OS UNIX files. By default, lstat() will not set EOVERFLOW when the file size can not be represented correctly in structure pointed to by buf. When _EDC_EOVERFLOW is set to YES, lstat() will check for an overflow condition.

Example

CELEBL12

```
/* CELEBL12

   This example provides status information for a file.

 */
#define _POSIX_SOURCE
#include <fcntl.h>
#include <sys/types.h>
#include <sys/stat.h>
```

```
#include <unistd.h>
#undef _POSIX_SOURCE
#include <stdio.h>
#include <time.h>

main() {
  char fn[]="temp.file", ln[]="temp.link";
  struct stat info;
  int fd;

  if ((fd = creat(fn, S_IWUSR)) < 0)
    perror("creat() error");
  else {
    close(fd);
    if (link(fn, ln) != 0)
      perror("link() error");
    else {
      if (lstat(ln, &info) != 0)
        perror("lstat() error");
      else {
        puts("lstat() returned:");
        printf("  inode:   %d\n",   (int) info.st_ino);
        printf(" dev id:   %d\n",   (int) info.st_dev);
        printf("   mode:   %08x\n",       info.st_mode);
        printf("  links:   %d\n",         info.st_nlink);
        printf("    uid:   %d\n",   (int) info.st_uid);
        printf("    gid:   %d\n",   (int) info.st_gid);
        printf("created:   %s",           ctime(&info.st_createtime));
      }
      unlink(ln);
    }
    unlink(fn);
  }
}
```

Output

```
lstat() returned:
  inode:   3022
 dev id:   1
   mode:   03000080
  links:   2
    uid:   25
    gid:   500
created:   Fri Jun 16 15:00:00 2006
```

Related information

- "sys/stat.h" on page 75
- "sys/types.h" on page 75
- "chmod() — Change the mode of a file or directory" on page 272
- "chown() — Change the owner or group of a file or directory" on page 275
- "creat() — Create a new file or rewrite an existing one" on page 340
- "dup() — Duplicate an open file descriptor" on page 404
- "extlink_np() — Create an external symbolic link" on page 455
- "fcntl() — Control open file descriptors" on page 474
- "fstat() — Get status information about a file" on page 651
- "link() — Create a link to a file" on page 970
- "mkdir() — Make a directory" on page 1068
- "mkfifo() — Make a FIFO special file" on page 1071
- "open() — Open a file" on page 1152
- "pipe() — Create an unnamed pipe" on page 1179
- "read() — Read from a file or socket" on page 1376
- "readlink() — Read the value of a symbolic link" on page 1387
- "remove() — Delete file" on page 1428

- "stat() — Get file information" on page 1720
- "symlink() — Create a symbolic link to a path name" on page 1795
- "unlink() — Remove a directory entry" on page 1962
- "utime() — Set file access and modification times" on page 1967
- "write() — Write data on a file or socket" on page 2085

l64a() — Convert long to base 64 string representation

Standards

Standards / Extensions	C or C++	Dependencies
XPG4.2 Single UNIX Specification, Version 3	both	

Format

```
#define  _XOPEN_SOURCE_EXTENDED 1
#include <stdlib.h>

char *l64a(long value);
```

General description

The l64a() function converts a long integer into its corresponding base 64 character representation. In this notation, long integers are represented by up to 6 characters, each character representing a digit in base 64 notation. The following characters are used to represent digits:

Character	Digit represented
.	0
/	1
0-9	2-11
A-Z	12-37
a-z	38-63

Returned value

l64a() returns a pointer to the base 64 representation of *value*. If *value* is zero, l64a() returns a pointer to a NULL string.

l64a() returns a pointer to a static buffer, which will be overwritten by subsequent calls. Buffers are allocated on a per-thread basis.

There are no errno values defined.

Related information

- "stdlib.h" on page 70
- "a64l() — Convert base 64 string representation to long integer" on page 207
- "strtoul() — Convert string to unsigned integer" on page 1780

ltoa() — Convert long into a string

Standards

Standards / Extensions	C or C++	Dependencies
z/OS UNIX	both	z/OS V1R5

Format

```
#define _OPEN_SYS_ITOA_EXT
#include <stdlib.h>

char * ltoa(long l, char * buffer, int radix);
```

General description

The ltoa() function coverts the long *l* into a character string. The string is placed in the buffer passed, which must be large enough to hold the output. The radix values can be OCTAL, DECIMAL, or HEX. When the radix is DECIMAL, ltoa() produces the same result as the following statement:

```
(void) sprintf(buffer, "%ld", l);
```

with buffer the returned character string. When the radix is OCTAL, ltoa() formats long *l* into an unsigned octal constant. When the radix is HEX, ltoa() formats long *l* into an unsigned hexadecimal constant. The hexadecimal value will include lower case abcdef, as necessary

Usage note

This is a non-standard function. Even though the prototype given is commonly used by compilers on other platforms, there is no guarantee that this function will behave the same on all platforms, in all cases. You can use this function to help port applications from other platforms, but you should avoid using it when writing new applications, in order to ensure maximum portability.

Returned value

String pointer (same as buffer) will be returned. When passed an invalid radix argument, function will return NULL and set errno to EINVAL.

Example

CELEBL29

```
/* CELEBL29

   This example reads a long int and formats it to decimal, unsigned
   octal, unsigned hexadecimal constants converted to a character
   string.

*/

#define _OPEN_SYS_ITOA_EXT
#include <stdio.h>
#include <stdlib.h>

int main ()
{
   long i;
   char buffer [sizeof(long)*8+1];
   printf ("Enter a number: ");
   if (scanf ("%ld",&i) == 1) {
      ltoa (i,buffer,DECIMAL);
      printf ("decimal: %s\n",buffer);
      ltoa (i,buffer,HEX);
      printf ("hexadecimal: %s\n",buffer);
      ltoa (i,buffer,OCTAL);
      printf ("octal: %s\n",buffer);
   }
   return 0;
}
```

Output

If the input is 1234, then the output should be:

```
decimal: 1234
hexadecimal: 4d2
octal: 2322
```

Related information

- “stdlib.h” on page 70
- “itoa() — Convert int into a string” on page 925
- “lltoa() — Convert long long into a string” on page 979
- “ulltoa() — Convert unsigned long long into a string” on page 1943
- “ultoa() — Convert unsigned long into a string” on page 1945
- “utoa() — Convert unsigned int into a string” on page 1972

makecontext() — Modify user context

Standards

Standards / Extensions	C or C++	Dependencies
XPG4.2 Single UNIX Specification, Version 3	both	POSIX(ON)

Format

```
#define _XOPEN_SOURCE_EXTENDED 1
#include <ucontext.h>

void makecontext(ucontext_t *ucp, void (*func)(), int argc, ...);
```

General description

The makecontext() function modifies the context specified by *ucp*, which has been initialized using getcontext(). When this context is resumed using setcontext() or swapcontext(), program execution continues by calling *func()*, passing it the arguments that follow *argc* in the makecontext() call.

The value of *argc* must match the number of integer arguments passed to *func()*, otherwise the behavior is undefined.

The **uc_link** member of **ucontext_t** is used to determine the context that will be resumed when the context being modified by makecontext() returns. If the **uc_link** member is not equal to 0, the process continues as if after a call to setcontext() with the context pointed to by the **uc_link** member. If the **uc_link** member is equal to 0, the process exits as if exit() were called. The **uc_link** member should be initialized before the call to makecontext().

This function is supported only in a POSIX program.

This function is not supported in an AMODE 31 XPLINK environment (for example, one which is in AMODE 31 and in which either the main() function was compiled with the XPLINK option, or the XPLINK(ON) runtime option was specified).

The **<ucontext.h>** header file defines the **ucontext_t** type as a structure that includes the following members:

```
mcontext_t       uc_mcontext     A machine-specific representation
                                 of the saved context.
ucontext_t      *uc_link         Pointer to the context that will
                                 be resumed when this context returns.
sigset_t         uc_sigmask      The set of signals that are blocked
                                 when this context is active.
stack_t          uc_stack        The stack used by this context.
```

Special behavior for C++

Because C and C++ linkage conventions are incompatible, makecontext() cannot receive C++ function pointers. If you attempt to pass a C++ function pointer to makecontext(), the compiler will flag it as an error. To use the C++ makecontext() function, you must ensure that all functions registered for makecontext() have C linkage by declaring them as `extern "C"`. For example:

```
C:   void func(int, int);
     :
     makecontext(&context, func, 2, arg1, arg2);

C++: extern "C" void func();
     :
     makecontext(&context, func, 2, arg1, arg2);
```

AMODE 64 considerations

Storage for the stack must be above the 2GB bar. It may not be storage acquired with the __malloc24() or __malloc31() functions. The stack must be big enough to allow for the creation of a 1M guard page (aligned on a 1M boundary).

The environment variable _EDC_CONTEXT_GUARD can be used to control when the stack is guarded and unguarded. For details on the _EDC_CONTEXT_GUARD environment variable, see the "Using Environment Variables" chapter in *z/OS XL C/C++ Programming Guide*.

Returned value

makecontext() returns no values.

If unsuccessful, makecontext() sets errno to one of the following values:

Error Code
    Description

EINVAL
    The context being modified is using an alternate stack, and the target function entry point is not a valid Language Environment or C entry point.

The *argc* argument specifies a value less than 0.

ENOMEM
The *ucp* argument does not have enough stack left to complete the operation. Or more than 15 arguments are passed to the target function, and there is not enough storage to hold all of the arguments.

Note: If the target function is in a DLL that has not yet been loaded, then makecontext() cannot determine the size requirement and assumes that the size required is MINSIGSTKSZ. Therefore, in this case, the stack must be at least the size indicated by MINSIGSTKSZ. If the size required by the target function is more than MINSIGSTKSZ, then you must load the DLL before invoking makecontext().

Example

This example creates a context in main with the getcontext() statement, then modifies the context to have its own stack and to invoke the function *func*. It invokes the function with the setcontext() statement. Since the **uc_link** member is set to 0, the process exits when the function returns.

```
/* This example shows the usage of makecontext().                   */

#define _XOPEN_SOURCE_EXTENDED 1
#include <stdlib.h>
#include <stdio.h>
#include <ucontext.h>
#include <errno.h>

 #ifdef _LP64
  #define STACK_SIZE 2097152+16384 /* large enough value for AMODE 64 */
 #else
  #define STACK_SIZE 16384         /* AMODE 31 addressing */
 #endif

void func(int);

ucontext_t context, *cp = &context;

int main(void) {

  int  value = 1;

  getcontext(cp);
  context.uc_link = 0;
  if ((context.uc_stack.ss_sp = (char *) malloc(STACK_SIZE)) != NULL) {
    context.uc_stack.ss_size = STACK_SIZE;
    context.uc_stack.ss_flags = 0;
             errno = 0;
    makecontext(cp,func,1,value);
              if(errno != 0){
                perror("Error reported by makecontext()");
                return -1;        /* Error occurred exit */ }
  }
  else {
    perror("not enough storage for stack");
    abort();
  }
  printf("context has been built\n");
  setcontext(cp);
  perror("returned from setcontext");
  abort();

}
```

```
void func(int arg) {

  printf("function called with value %d\n",arg);
  printf("process will exit when function returns\n");
  return;

}
```

Output

```
context has been built
function called with value 1
process will exit when function returns
```

Related information

- “ucontext.h” on page 82
- “getcontext() — Get user context” on page 694
- “setcontext() — Restore user context” on page 1524
- “swapcontext() — Save and restore user context” on page 1792

malloc() — Reserve storage block

Standards

Standards / Extensions	C or C++	Dependencies
ISO C POSIX.1 XPG4 XPG4.2 C99 Single UNIX Specification, Version 3	both	

Format

```
#include <stdlib.h>

void *malloc(size_t size);
```

General description

Reserves a block of storage of *size* bytes. Unlike the calloc() function, the content of the storage allocated is indeterminate. The storage to which the returned value points is always aligned for storage of any type of object. Under z/OS XL C only, if 4K alignment is required, use the __4kmalc() function. (This function is available to C applications in stand-alone System Productivity Facility (SPF) applications.) The library functions specific to the System Programming C (SPC) environment are described in *z/OS XL C/C++ Programming Guide*.

Special behavior for C++

The C++ keywords new and delete are not interoperable with calloc(), free(), malloc(), or realloc().

Returned value

If successful, malloc() returns a pointer to the reserved space. The storage space to which the returned value points is always suitably aligned for storage of any type of object.

If not enough storage is available, or if *size* was specified as 0, malloc() returns NULL. If malloc() returns NULL because there is not enough storage, it sets errno to one of the following values:

Error Code
Description

ENOMEM
Insufficient memory is available

Example

CELEBM01

```
/* CELEBM01

   This example prompts you for the number of array entries you
   want and then reserves enough space in storage for the entries.
   If &malloc. was successful, the example assigns values
   to the entries and prints out each entry; otherwise, it prints
   out an error.

 */
#include <stdio.h>
#include <stdlib.h>

int main(void)
{
  long * array;    /* start of the array */
  long * index;    /* index variable     */
  int    i;        /* index variable     */
  int  num;        /* number of entries of the array */

  printf( "Enter the size of the array\n" );
  scanf( "%i", &num );

  /* allocate num entries */
  if ( (index = array = (long * )malloc( num * sizeof( long ))) != NULL )
  {

    for ( i = 0; i < num; ++i )           /* put values in array    */
       *index++ = i;                      /* using pointer notation */

    for ( i = 0; i < num; ++i )           /* print the array out    */
      printf( "array[ %i ] = %i\n", i, array[i] );
  }
  else { /* malloc error */
    printf( "Out of storage\n" );
    abort();
  }
}
```

Output

```
Enter the size of the array
array[ 0 ] = 0
array[ 1 ] = 1
array[ 2 ] = 2
array[ 3 ] = 3
array[ 4 ] = 4
```

Related information

- See the topic about using the system programming C facilities in *z/OS XL C/C++ Programming Guide*
- “stdlib.h” on page 70

- “calloc() — Reserve and initialize storage” on page 231
- “free() — Free a block of storage” on page 618
- “__malloc24() — Allocate 24-bit storage”
- “__malloc31() — Allocate 31–bit storage”
- “realloc() — Change reserved storage block size” on page 1391

__malloc24() — Allocate 24-bit storage

Standards

Standards / Extensions	C or C++	Dependencies
	both	AMODE 64

Format

```
#include <stdlib.h>

void  *__malloc24(size_t size);
```

General description

Reserves a block of storage of *size* bytes from “below-the-line” storage (that is, below 16 MB).

Returned value

If successful, __malloc24() returns a pointer to the reserved space. The storage space to which the returned value points is always suitably aligned for storage of any type of object.

If not enough storage is available, or if *size* was specified as 0, __malloc24() returns NULL. If __malloc24() returns NULL because there is not enough storage, it sets errno to one of the following values:

Error Code
Description

ENOMEM
Insufficient memory is available

Related information

- See the topic about using the system programming C facilities in *z/OS XL C/C++ Programming Guide*
- “stdlib.h” on page 70
- “calloc() — Reserve and initialize storage” on page 231
- “free() — Free a block of storage” on page 618
- “malloc() — Reserve storage block” on page 1031
- “__malloc31() — Allocate 31–bit storage”
- “realloc() — Change reserved storage block size” on page 1391

__malloc31() — Allocate 31–bit storage

Standards

Standards / Extensions	C or C++	Dependencies
	both	AMODE 64

Format

```
#include <stdlib.h>

void  *__malloc31(size_t size);
```

General description

Reserves a block of storage of *size* bytes from "below-the-bar" storage (that is, below 2 GB).

Returned value

If successful, __malloc31() returns a pointer to the reserved space. The storage space to which the returned value points is always suitably aligned for storage of any type of object.

If not enough storage is available, or if *size* was specified as 0, __malloc31() returns NULL. If __malloc31() returns NULL because there is not enough storage, it sets errno to one of the following values:

Error Code
    Description

ENOMEM
    Insufficient memory is available

Related information

- "Using the System Programming C Facilities" in *z/OS XL C/C++ Programming Guide*
- "stdlib.h" on page 70
- "calloc() — Reserve and initialize storage" on page 231
- "free() — Free a block of storage" on page 618
- "malloc() — Reserve storage block" on page 1031
- "__malloc24() — Allocate 24-bit storage" on page 1033
- "realloc() — Change reserved storage block size" on page 1391

__map_init() — Designate a storage area for mapping blocks

Standards

Standards / Extensions	C or C++	Dependencies
	both	POSIX(ON) OS/390 V2R9

Format

```
#define _OPEN_SYS_MAP_EXTENTION
#include <sys/mman.h>

int __map_init(struct _Mmg_init *parmlist);
```

General description

The __map_init() function allocates a map area in the private area of the calling address space. This map area is propagated to child address spaces on fork, which is the only way that multiple processes can share a map area. The application can connect and disconnect blocks of storage in the map area, providing a very fast way to connect up to persistent memory. The __map_init() function is meant to be used by applications which need more shared memory or mmap storage than will fit in the address space.

The application should set the following values in the _Mmg_init structure:

Element
: **Description**

_Mmg_numblks
: Set to the number of blocks to be contained in the map area.

_Mmg_megsperblk
: Set to the size in megabytes of each block in the map area.

_Mmg_token
: Set to an 8 character map token when successful. This map token should be saved and must be used as a parameter on calls to the __map_service() function calls.

_Mmg_res01a
: Reserved, set to 0.

_Mmg_res01b
: Reserved, set to 0.

_Mmg_areaaddr
: As input, set to 0 if you want the address assigned or set to the address of storage where you want the map to begin. As output, this field contains the actual address of the map area.

Usage notes

- It is intended that the application call the __map_init() service once to create the map area.
- The application then issues fork to create child processes which will inherit a map area initialized to the hidden state.
- The initial process or the child (and grandchildren) process can then use the __map_service to connect and disconnect blocks of storage which are persistent until explicitly deleted.
- When the process which created the initial map area terminates, all further activity against the map blocks is terminated. The map blocks are then deleted when the last child process with an active map area terminates.
- There is no explicit call to delete the map area. This is unlike shared memory or other IPC constructs.

Returned value

If successful, __map_init() returns NULL.

If unsuccessful, __map_init() returns -1 and sets errno to one of the following values:

Error Code
: **Description**

EEXIST
: An attempt was made to create more than one map for the process.

EFAULT
: The *parmlist* (_Mmg_init structure) has an argument that is not accessible to the caller.

EINVAL
: One of the following occurred:

The number of blocks (_Mmg_numblks) was zero or negative.

The number of megabytes per block (_Mmg_Megsperblk) was zero or negative.

A reserved field contained nonzero data.

The specified address (_Mmg_areaaddr) is not on a megabyte multiple.

EMVSSAF2ERR
An error occurred in the security product. Use the __errno2() function to retrieve the reason code to determine the exact reason the error occurred.

ENOMEM
The requested storage at location _Mmg_areaaddr or the size requested could not be obtained. The storage is either not available or your Region size is too small to contain the map area. Or, there is insufficient free virtual storage in the address space to satisfy the request.

EPERM
The user is not authorized to use the __map_init() function. Callers must be permitted to the BPX.MAP FACILITY class profile to use this service.

Related information

- "sys/mman.h" on page 73
- "__map_service() — Set memory mapping service"

__map_service() — Set memory mapping service

Standards

Standards / Extensions	C or C++	Dependencies
	both	POSIX(ON) OS/390 V2R9

Format

```
#define _OPEN_SYS_MAP_EXTENTION
#include <sys/mman.h>

int __map_service(struct _Mmg_service *parmlist, int count, *_Map_token_t);
```

General description

The __map_service() function is used to manipulate the map area created by the __map_init() function. The supported functions are defined under _Mmg_servicetype below.

Before calling the __map_service() service, the application should set values in the _Mmg_service structure as follows:

Element
Description

_Mmg_servicetype
Set the type of service being requested for each memory block defined in the array.

Request
Description

_Mmg_newblock
Set for an allocation of a new data block in the mapped area.

_Mmg_conn
Set to request that a data block be connected at the requested location in the map area.

_Mmg_disconn
Set to disconnect a data block from the map area.

_Mmg_free
Set to free the storage backing a data block.

_Mmg_cntl
Set to change the read or write permission settings for a data block.

_Mmg_serviceIflag
Used for _Mmg_cntl to indicate read or write and all other bits set to zero. For _Mmg_disconn to indicate if the backing storage is to be freed after disconnect. For _Mmg_newblock the option of _Mmg_NoConn can be set on to bypass the connect to the map area block. The token returned will have to be saved and used for connect services on a later call to make the block accessible. For all other _Mmg_serviceItype requests, set all the bits to zero.

_Mmg_serviceOflag
Used for status of the request. When the request has been successfully processed all the bits are set to zero. When processing an list of requests and a failure occurs in _Mmg_Reqfail is set on and further processing on the list is aborted. _Mmg_servicetype requests, set all the bits to zero.

_Mmg_token
This is returned as output for a _Mmg_newblock request and is used as input for _Mmg_conn, and _Mmg_free. It is ignored for _Mmg_disconn and _Mng_cntl.

_Mmg_res0b
Reserved, set to 0.

_Mmg_blkaddr
For _Mmg_newblock and _Mmg_conn this is input. It should be set to an address within the map area (on a block multiple) where you want to allocate a block or 0. If 0 is specified, the first available block in the map area is used. On output, this field contains the address within the map area that was assigned to the data block. For _Mmg_disconn it is input only and contains the address of the map block to be disconnected. For _Mng_cntl this field is required and specifies the block to be use for the _Mmg_cntl option. For _Mmg_free, _Mmg_cntl and _Mmg_newblock, when the option of _Mmg_NoConn is set on, this field is ignored.

With *count* reflecting the number of _Mmg_service structures included in the array structure supplied on *parmlist* parameter. With *count* a positive integer in the range of 1-1000.

With *_Map_token_t* the 8 character token retrieved from the _Mmg_token field in a _Mmg_service structure that returned successful from a __map_init() function call.

Usage notes

- The __map_init and the __map_service functions are intended to be used in the following manner:
  - The initial process calls __map_init to create a map area large enough for the biggest expected usage.
  - The initial process forks worker processes which inherit the map area at the same virtual address. By having the map area at the same virtual address, storage blocks can be connected to the same block in map areas of different

worker processes and pointers can be used to point to data in this or other blocks. This assumes they are always connected at the same location in the map area.

- As worker processes perform their tasks, they can request new blocks of storage to be created in the map area. Each block has a token associated with it. This token allows other worker processes to connect to the same block. In this respect, the map area acts like shared memory.
- The worker processes can connect as many blocks to their map area as will fit.
- When the worker process has no further need for a data block, it can disconnect it from the map area. After a delete request for a block, this block is actually freed when the last worker process disconnects for this block.
- When a worker process is completely done with a data block, the storage can be freed. This data is actually freed when the last worker process disconnects from that block.
- Using these services, the application could create multiple gigabytes of storage, of which only certain blocks are mapped into the worker processes at a given time.
- This service is designed to perform the storage connects and disconnects very fast. No data movement occurs.
- Storage blocks are initially connected in write mode. When a block is in write mode, all worker processes which have the block connected, have the block in write mode. If the block access is changed to read-only, then all worker processes which have the block connected, have the block in read-only mode.
- If the initial process or a worker process forks, then the child process inherits a map area initialized to the hidden state.
- Any areas within the map area which do not have a block connected are in the hidden state. Any reference to storage in the hidden state will trigger a SIGSEGV signal.

Returned value

If successful, __map_service() returns 0.

If unsuccessful, __map_service() returns -1 and sets errno to one of the following values:

Error Code
: **Description**

EEXIST
: A request was made to perform a service on a block but a map is not currently active in the process.

EFAULT
: The *parmlist* (_Mmg_service structure) argument addresses either could not be accessed or was in read-only storage and could not be updated.

EINVAL
: For one of the following reasons:

  The block address provided is either not in the map area or it is not on a map block boundary.

  A request was made to connect to a block or free the backing storage for a block but the token provided does not match that of any allocated block in the backing store.

  A request was made to disconnect from a block but the block is not currently in the map area for this process.

A newblock or connect request was specified for a map area block that is already in use.

A request was made to connect to a block in the backing store that is currently marked to be freed. The connect is not permitted.

The *count* value was not a positive integer in the range of 1-1000.

ENOMEM
A request to create a new block or connect to an existing block was made but there are no unused blocks in the map area to satisfy the request.

Related information

- "sys/mman.h" on page 73
- "__map_init() — Designate a storage area for mapping blocks" on page 1034

maxcoll() — Return maximum collating element

Standards

Standards / Extensions	C or C++	Dependencies
z/OS UNIX	both	

Format

```
#include <collate.h>

collel_t maxcoll(void);
```

General description

Returns the largest possible value of a collating element in the current locale.

Related information

- "collate.h" on page 19
- "cclass() — Return characters in a character class" on page 243
- "collequiv() — Return a list of equivalent collating elements" on page 295
- "collorder() — Return list of collating elements" on page 297
- "collrange() — Calculate the range list of collating elements" on page 298
- "colltostr() — Return a string for a collating element" on page 299
- "getmccoll() — Get next collating element from string" on page 734
- "getwmccoll() — Get next collating element from wide string" on page 803
- "ismccollel() — Identify a multicharacter collating element" on page 914
- "strtocoll() — Return collating element for string" on page 1763

maxdesc() — Get socket numbers to extend beyond the default range

Standards

Standards / Extensions	C or C++	Dependencies
z/OS UNIX	both	

Format

```
#define _OPEN_SYS_SOCK_EXT
#include <sys/types.h>
#include <sys/socket.h>

int maxdesc(int *totdesc, int *inetdesc);
```

General description

Bulk mode sockets are not supported. Do not use this function.

Returned value

If successful, maxdesc() returns 0.

If unsuccessful, maxdesc() returns -1 and sets errno to one of the following values:

Error Code
  Description

EALREADY
  Your program called maxdesc() after creating a socket, after a call to setibmsockopt(), or after a previous call to maxdesc().

EFAULT
  Using the totdesc parameter as specified results in an attempt to access storage outside of the caller's address space, or storage not modifiable by the caller.

ENOMEM
  Your address space has insufficient storage.

Related information

- "sys/socket.h" on page 74
- "sys/types.h" on page 75
- "getstablesize() — Get the socket table size" on page 786
- "getrlimit() — Get current or maximum resource consumption" on page 765

mblen() — Calculate length of multibyte character

Standards

Standards / Extensions	C or C++	Dependencies
ISO C XPG4 XPG4.2 C99 Single UNIX Specification, Version 3	both	

Format

```
#include <stdlib.h>

int mblen(const char *string, size_t n);
```

General description

Determines the length in bytes of the multibyte character pointed to by *string*. A maximum of *n* bytes is examined.

The behavior of this wide-character function is affected by the LC_CTYPE category of the current locale. Changing the LC_CTYPE category invalidates the internal shift state: undefined results can occur.

If the current locale supports EBCDIC DBCS characters, then the shift state is updated where applicable. (See "Conforming to ANSI Standards" in *z/OS XL C/C++ Language Reference*.) The length returned may be up to 4 (for the shift-out character, 2-byte code, and the shift-in character). If *string* is a NULL pointer, this function resets itself to the initial state.

The function maintains the internal shift state that is altered by subsequent calls.

Returned value

If *string* is NULL, mblen() returns:
- Nonzero when DBCS-host code (EBCDIC systems) is used
- Nonzero if multibyte encodings are state-dependent
- Zero otherwise

If *string* is not NULL, mblen() returns:
- Zero if *string* points to the NULL character
- The number of bytes comprising the multibyte character
- The value -1 if *string* does not point to a valid multibyte character

Example

```
#include <locale.h>
#include <stdlib.h>
#include <stdio.h>

int main(void)
{
    char *mbs = "a"
                "\x0E"      /* shift out */
                "\x44\x66"  /* <j0158> */
                "\x44\x76"  /* <j0159> */
                "\x42\x4e"  /* <j0160> */
                "\x0F"      /* shift in */
                "b";
    char *loc = setlocale(LC_ALL, "JA_JP.IBM-939");
    int n;

    if (!loc)   /* setlocale() failure */
    {
        exit(8);
    }

    printf("We're in the %s locale.\n", loc);

    n = mblen(NULL, MB_CUR_MAX);
    /*****************************************************************/
    /* n is nonzero, indicating state-dependent encoding; mblen() has */
    /* forced the internal shift state to "initial".                 */
    /*****************************************************************/
    printf("n = mblen(NULL, MB_CUR_MAX);     ===> n = %s\n",
        n ? "NONZERO" : "ZERO");

    n = mblen(mbs, MB_CUR_MAX);
    /*****************************************************************/
    /* n is 1, 'a' is a multibyte character of length 1, internal    */
    /* shift state remains at "initial".                             */
    /*****************************************************************/
    printf("n = mblen(mbs, MB_CUR_MAX);      ===> n = %d\n", n);
    n = mblen(mbs + 1, MB_CUR_MAX);
    /******************************************************************/
    /* n is 3, 'shift out' plus two byte character '<j0158>'.  The    */
    /* internal state changes to "shift out".                         */
    /******************************************************************/
    printf("n = mblen(mbs + 1, MB_CUR_MAX);  ===> n = %d\n", n);

    n = mblen(mbs + 4, MB_CUR_MAX);
    /******************************************************************/
    /* n is 2, two byte character '<j0159>'.  The internal shift      */
    /* state remains "shift out"                                      */
    /******************************************************************/
    printf("n = mblen(mbs + 4, MB_CUR_MAX);  ===> n = %d\n", n);

    n = mblen(mbs + 6, MB_CUR_MAX);
    /******************************************************************/
    /* n is 3, two byte character '<j0160>' plus 'shift in'.  The     */
    /* internal shift state returns to "initial".                     */
    /******************************************************************/
    printf("n = mblen(mbs + 6, MB_CUR_MAX);  ===> n = %d\n", n);

    n = mblen(mbs + 9, MB_CUR_MAX);
    /******************************************************************/
    /* n is 1, 'b' is a multibyte character of length 1, internal     */
    /* shift state remains at "initial".                              */
    /******************************************************************/
    printf("n = mblen(mbs + 9, MB_CUR_MAX);  ===> n = %d\n", n);

    n = mblen(mbs + 10, MB_CUR_MAX);
    /******************************************************************/
    /* n is 0 (end of multibyte character string).                    */
    /******************************************************************/
    printf("n = mblen(mbs + 10, MB_CUR_MAX); ===> n = %d\n", n);

    return 0;
}
```

Output

```
We're in the JA_JP.IBM-939 locale.
n = mblen(NULL, MB_CUR_MAX);     ===> n = NONZERO
n = mblen(mbs, MB_CUR_MAX);      ===> n = 1
n = mblen(mbs + 1, MB_CUR_MAX);  ===> n = 3
n = mblen(mbs + 4, MB_CUR_MAX);  ===> n = 2
n = mblen(mbs + 6, MB_CUR_MAX);  ===> n = 3
n = mblen(mbs + 9, MB_CUR_MAX);  ===> n = 1
n = mblen(mbs + 10, MB_CUR_MAX); ===> n = 0
```

Related information

- The topic describing internationalization of locales and character sets in *z/OS XL C/C++ Programming Guide*
- "locale.h" on page 40
- "stdlib.h" on page 70
- "mbrlen() — Calculate length of multibyte character" on page 1043
- "mbrtowc() — Convert a multibyte character to a wide character" on page 1049
- "mbsrtowcs() — Convert a multibyte string to a wide-character string" on page 1052
- "mbstowcs() — Convert multibyte characters to wide characters" on page 1055
- "mbtowc() — Convert multibyte character to wide character" on page 1056
- "setlocale() — Set locale" on page 1552
- "strlen() — Determine string length" on page 1748
- "wcrtomb() — Convert a wide character to a multibyte character" on page 2001
- "wcslen() — Calculate length of wide-character string" on page 2013

- “wcsrtombs() — Convert wide-character string to multibyte string” on page 2021
- “wctomb() — Convert wide character to multibyte character” on page 2053

mbrlen() — Calculate length of multibyte character

Standards

Standards / Extensions	C or C++	Dependencies
ISO C Amendment C99 Single UNIX Specification, Version 3	both	

Format

Non-XPG4

```
#include <wchar.h>

size_t mbrlen(const char * __restrict__s, size_t n, mbstate_t * __restrict__ps);
```

XPG4

```
#define _XOPEN_SOURCE
#define _MSE_PROTOS
#include <wchar.h>

size_t mbrlen(const char *s, size_t n, mbstate_t *ps);
```

General description

Calculates the number of bytes required to return to the initial shift state. This is equivalent to

```
   mbrtowc((wchar_t *)0, s, n, ps != NULL ? ps : &internal);
```

where `&internal` is the address of the internal `mbstate_t` object for the mbrlen() function.

mbrlen() is a restartable version of mblen(). That is, shift state information is passed as one of the arguments, and is updated on exit. With mbrlen(), you can switch from one multibyte string to another, provided that you have kept the shift-state information.

The behavior of this wide-character function is affected by the LC_CTYPE category of the current locale. If you change the category, undefined results can occur.

Special behavior for XPG4

If you define any feature test macro specifying XPG4 behavior before the statement in your program source file to include the wchar header, then you must also define the _MSE_PROTOS feature test macro to make the declaration of the mbrlen() function in the wchar header available when you compile your program. Please see Table 2 on page 4 for a list of XPG4 and other feature test macros.

Returned value

If *s* is a NULL pointer, mbrlen() resets the shift state to the initial shift state and returns 0.

If *s* is not a NULL pointer, mbrlen() returns the first of the following that applies.

0 If the next *n* or fewer bytes complete the valid multibyte character that corresponds to the NULL wide character.

positive
If the next *n* or fewer bytes complete the valid multibyte character; the value returned is the number of bytes that complete the multibyte character.

-2 If the next *n* bytes form an incomplete (but potentially valid) multibyte character, and all *n* bytes have been processed; it is unspecified whether this can occur when the value of *n* is less than that of the MB_CUR_MAX macro.

Note: When a -2 value is returned, and *n* is at least MB_CUR_MAX, the string would contain redundant shift-out and shift-in characters. To continue processing the multibyte string, increment the pointer by the value *n*, and call the mbrtowc() function.

-1 If an encoding error occurs (when the next *n* or fewer bytes do not contribute to the complete and valid multibyte character), the value of the macro EILSEQ is stored in errno, but the conversion state remains unchanged.

Example

CELEBM03

```
/* CELEBM03 */
#include <stdio.h>
#include <stdlib.h>
#include <wchar.h>

int main(void)
{
   char      mbs[5] = "a";    /* string containing the multibyte char */
   mbstate_t ss     = 0;      /* set shift state to the initial state */
   int       length;

   /* Determine the length in bytes of a multibyte character pointed  */
   /* to by mbs.                                                      */

   length = mbrlen(mbs, MB_CUR_MAX, &ss);

   printf("    length: %d \n", length);
   printf("       mbs:\"%s\"\n", mbs);
   printf("MB_CUR_MAX: %d \n", MB_CUR_MAX);
   printf("        ss: %d \n", ss);
}
```

Output

```
    length: 1
       mbs:"a"
MB_CUR_MAX: 4
        ss: 0
```

Related information

- “Internationalization: Locales and Character Sets” in *z/OS XL C/C++ Programming Guide*
- “locale.h” on page 40
- “wchar.h” on page 85

- "mblen() — Calculate length of multibyte character" on page 1040
- "mbrtowc() — Convert a multibyte character to a wide character" on page 1049
- "mbsrtowcs() — Convert a multibyte string to a wide-character string" on page 1052
- "mbtowc() — Convert multibyte character to wide character" on page 1056
- "setlocale() — Set locale" on page 1552
- "wcrtomb() — Convert a wide character to a multibyte character" on page 2001
- "wcsrtombs() — Convert wide-character string to multibyte string" on page 2021

mbrtoc16() — Convert a multibyte character to a char16_t character

Standards

Standards / Extensions	C or C++	Dependencies
ISO C Amendment C11	both	z/OS V2R1

Format

```
#include <uchar.h>

size_t mbrtoc16(char16_t * restrict pc16,
                const char * restrict s,
                size_t n,
                mbstate_t * restrict ps);
```

General description

The mbrtoc16() function converts a multibyte character to a wide character of type char16_t, and returns the number of bytes of the multibyte character.

If *s* is not a null pointer, the mbrtoc16() function inspects at most *n* bytes beginning with the byte pointed to by *s* to determine the number of bytes needed to complete the next multibyte character (including any shift sequences). If the function determines that the next multibyte character is complete and valid, it determines the values of the corresponding wide characters and then, if *pc16* is not a null pointer, stores the value of the first (or only) such character in the object pointed to by *pc16*. Subsequent calls will store successive wide characters without consuming any additional input until all the characters have been stored. If the corresponding wide character is the null wide character, the resulting state described is the initial conversion state.

If *s* is a null pointer, the mbrtoc16() function is equivalent to the call `mbrtoc16(NULL,"",1,ps)`. In this case, the values of the parameters *pc16* and *n* are ignored.

If *ps* is a null pointer, mbrtoc16() uses its own internal object to track the shift state. Otherwise **ps* must be a valid mbstate_t object. An mbstate_t object **ps* can be initialized to the initial state by assigning 0 to it, or by calling `mbrtoc16(NULL,NULL,0,ps)`.

Usage notes

1. To use the mbrtoc16() function, compile the source code with the LANGLVL(EXTC1X) option.
2. The mbrtoc16() function only supports the CCSIDs that are provided by Unicode Services.

3. The result of converting multiple string alternately in one thread by using multiple mbstate_t objects (including the internal one) is undefined.

Returned value

The mbrtoc16() function returns the first of the following that applies (given the current conversion state):

0
: If the next *n* or fewer bytes complete the multibyte character that corresponds to the null wide character (which is the value stored).

between 1 and *n* inclusive
: If the next *n* or fewer bytes complete a valid multibyte character (which is the value stored); the value returned is the number of bytes that complete the multibyte character.

-3
: If the next character resulting from a previous call has been stored (no byte from the input has been consumed by this call).

-2
: If the next *n* bytes contribute to an incomplete (but potentially valid) multibyte character, and all *n* bytes have been processed (no value is stored). When *n* has at least the value of the MB_CUR_MAX macro, this case can only occur if *s* points to a sequence of redundant shift sequence (for implementations with state-dependent encodings).

-1
: If an encoding error occurs (when the next *n* or fewer bytes do not contribute to a complete and valid multibyte character). The value of the macro EILSEQ is stored in errno, and the conversion state is unspecified.

Example

```
#include <stdio.h>
#include <stdlib.h>
#include <uchar.h>

int main(void)
{
   char16_t c16;
   char mbs[] = "a" ; /* string containing the multibyte character */
   mbstate_t ss = 0 ;    /* set shift state to the initial state */
   int length = 0 ;

   /* Determine the length of the multibyte character pointed to by */
   /* mbs. Store the multibyte character in the char16_t object */
   /* called c16. */

   length = mbrtoc16(&c16, mbs, MB_CUR_MAX, &ss);
   if (length < 0) {
      /* -2 and -3 return value could not happen during converting the 'a' */
      perror("mbrtoc16() fails to convert");
      exit(-1);
   }

   printf(" mbs:\"%s\"\n", mbs);
   printf(" length: %d \n", length);
   printf(" c16: 0x%04hx \n", c16);
}
```

Output:

```
mbs:"a"
length: 1
c16: 0x0061
```

Related information

- “Internationalization: Locales and Character Sets” in *z/OS XL C/C++ Programming Guide*
- “uchar.h” on page 82
- “setlocale() — Set locale” on page 1552
- “c16rtomb() — Convert a char16_t character to a multibyte character” on page 223
- “mbrtoc32() — Convert a multibyte character to a char32_t character”

mbrtoc32() — Convert a multibyte character to a char32_t character

Standards

Standards / Extensions	C or C++	Dependencies
ISO C Amendment C11	both	z/OS V2R1

Format

```
#include <uchar.h>

size_t mbrtoc32(char32_t * restrict pc32,
               const char * restrict s,
               size_t n,
               mbstate_t * restrict ps);
```

General description

The mbrtoc32() function converts a multibyte character to a wide character of type char32_t, and returns the number of bytes of the multibyte character.

If *s* is not a null pointer, the mbrtoc32() function inspects at most *n* bytes beginning with the byte pointed to by *s* to determine the number of bytes needed to complete the next multibyte character (including any shift sequences). If the function determines that the next multibyte character is complete and valid, it determines the values of the corresponding wide characters and then, if *pc32* is not a null pointer, stores the value of the first (or only) such character in the object pointed to by *pc32*. Subsequent calls will store successive wide characters without consuming any additional input until all the characters have been stored. If the corresponding wide character is the null wide character, the resulting state described is the initial conversion state.

If *s* is a null pointer, the mbrtoc32() function is equivalent to the call `mbrtoc32(NULL,"",1,ps)`. In this case, the values of the parameters *pc32* and *n* are ignored.

If *ps* is a null pointer, mbrtoc32() uses its own internal object to track the shift state. Otherwise **ps* must be a valid mbstate_t object. An mbstate_t object **ps* can be initialized to the initial state by assigning 0 to it, or by calling `mbrtoc32(NULL,NULL,0,ps)`.

Usage notes

1. To use the mbrtoc32() function, compile the source code with the LANGLVL(EXTC1X) option.

2. The mbrtoc32() function only supports the CCSIDs that are provided by Unicode Services.
3. The result of converting multiple string alternately in one thread by using multiple mbstate_t objects (including the internal one) is undefined.

Returned value

The mbrtoc32() function returns the first of the following that applies (given the current conversion state):

0
If the next *n* or fewer bytes complete the multibyte character that corresponds to the null wide character (which is the value stored).

between 1 and *n* inclusive
If the next *n* or fewer bytes complete a valid multibyte character (which is the value stored); the value returned is the number of bytes that complete the multibyte character.

-3
If the next character resulting from a previous call has been stored (no byte from the input has been consumed by this call).

-2
If the next *n* bytes contribute to an incomplete (but potentially valid) multibyte character, and all *n* bytes have been processed (no value is stored). When *n* has at least the value of the MB_CUR_MAX macro, this case can only occur if *s* points to a sequence of redundant shift sequence (for implementations with state-dependent encodings).

-1
If an encoding error occurs (when the next *n* or fewer bytes do not contribute to a complete and valid multibyte character). The value of the macro EILSEQ is stored in errno, and the conversion state is unspecified.

Example

```
#include <stdio.h>
#include <stdlib.h>
#include <uchar.h>

int main(void)
{
   char32_t c32;
   char mbs[] = "a" ; /* string containing the multibyte character */
   mbstate_t ss = 0 ;    /* set shift state to the initial state */
   int length = 0 ;

   /* Determine the length of the multibyte character pointed to by */
   /* mbs. Store the multibyte character in the char32_t object */
   /* called c32. */

   length = mbrtoc32(&c32, mbs, MB_CUR_MAX, &ss);
   if (length < 0) {
      /* -2 and -3 return value could not happen during converting the 'a' */
      perror("mbrtoc32() fails to convert");
      exit(-1);
   }

   printf(" mbs:\"%s\"\n", mbs);
   printf(" length: %d \n", length);
   printf(" c32: 0x%08x \n", c32);
}
```

Output:

```
mbs:"a"
length: 1
c32: 0x00000061
```

Related information

- "Internationalization: Locales and Character Sets" in *z/OS XL C/C++ Programming Guide*
- "uchar.h" on page 82
- "setlocale() — Set locale" on page 1552
- "c32rtomb() — Convert a char32_t character to a multibyte character" on page 225
- "mbrtoc16() — Convert a multibyte character to a char16_t character" on page 1045

mbrtowc() — Convert a multibyte character to a wide character

Standards

Standards / Extensions	C or C++	Dependencies
ISO C Amendment C99 Single UNIX Specification, Version 3	both	

Format

Non-XPG4

```
#include <wchar.h>

size_t mbrtowc(wchar_t * __restrict__pwc, const char * __restrict__s,
               size_t n, mbstate_t * __restrict__ps);
```

XPG4

```
#define _XOPEN_SOURCE
#define _MSE_PROTOS
#include <wchar.h>

size_t mbrtowc(wchar_t *pwc, const char *s, size_t n, mbstate_t *ps);
```

General description

The mbrtowc() function is equivalent to `mbrtowc(NULL,"",1,ps)`.

If *s* is a NULL pointer, the mbrtowc() function ignores the *n* and the *pwc*, and resets the shift state, pointed to by *ps*, to the initial shift state.

If *s* is not a NULL pointer, mbrtowc() inspects at most *n* bytes, beginning with the byte pointed to by *s*, and the shift state pointed to by *ps*, and determines the number of bytes that is needed to complete the valid multibyte character.

When the multibyte character is completed, mbrtowc() determines the value of the corresponding wide character and stores it in the object pointed to by *pwc*, so long as *pwc* is not a NULL pointer. Finally, mbrtowc() stores the actual shift state in the object pointed to by *ps*. If *ps* is a NULL pointer, mbrtowc() uses its own internal object to track the shift state.

mbrtowc() is a restartable version of mbtowc(). That is, shift-state information is passed as one of the arguments and is updated on exit. With mbrtowc(), you can switch from one multibyte string to another, provided that you have kept the shift-state information.

The behavior of this wide-character function is affected by the LC_CTYPE category of the current locale. If you change the category, undefined results may occur.

Special behavior for XPG4

If you define any feature test macro specifying XPG4 behavior before the statement in your program source file to include the wchar header, then you must also define the _MSE_PROTOS feature test macro to make the declaration of the mbrtowc() function in the wchar header available when you compile your program. Please see Table 2 on page 4 for a list of XPG4 and other feature test macros.

Returned value

If *s* is a NULL pointer, mbrtowc() resets the shift state to the initial shift state and returns 0.

If *s* is not a NULL pointer, mbrtowc() returns one of the following, in the order shown:

0
If the next *n* or fewer bytes complete the valid multibyte character that corresponds to the NULL wide character.

positive integer
If the next *n* or fewer bytes complete the valid multibyte character; the value returned is the number of bytes that complete the multibyte character.

-2
If the next *n* bytes form an incomplete (but potentially valid) multibyte character, and all *n* bytes have been processed. It is unspecified whether this can occur when the value of *n* is less than that of the MB_CUR_MAX macro.

Note: When a -2 value is returned, and *n* is at least MB_CUR_MAX, the string would contain redundant shift-out and shift-in characters. To continue processing the multibyte string, increment the pointer by the value *n*, and call the mbrtowc() function.

-1
If an encoding error occurs (when the next *n* or fewer bytes do not contribute to the complete and valid multibyte character). The value of the macro EILSEQ is stored in errno, but the conversion state is unchanged.

Example

CELEBM04

```
/* CELEBM04 */
#include <stdio.h>
#include <stdlib.h>
#include <wchar.h>

int main(void)
{
   wchar_t   wc;
   char      mbs[5] = "a";    /* string containing the multibyte char */
   mbstate_t ss     = 0;      /* set shift state to the initial state */
   int      length;

   /* Determine the length of the multibyte character pointed to by   */
   /* mbs.  Store the multibyte character in the wchar_t object       */
   /* called wc.                                                      */

   length = mbrtowc(&wc, mbs, MB_CUR_MAX, &ss);

   printf("    length: %d \n", length);
   printf("        wc:'%lc'\n", wc);
   printf("       mbs:\"%s\"\n", mbs);
   printf("MB_CUR_MAX: %d \n", MB_CUR_MAX);
   printf("        ss: %d \n", ss);
}
```

Related information

- "Internationalization: Locales and Character Sets" in *z/OS XL C/C++ Programming Guide*
- "locale.h" on page 40
- "wchar.h" on page 85
- "mblen() — Calculate length of multibyte character" on page 1040
- "mbrlen() — Calculate length of multibyte character" on page 1043
- "mbsrtowcs() — Convert a multibyte string to a wide-character string" on page 1052
- "setlocale() — Set locale" on page 1552
- "wcrtomb() — Convert a wide character to a multibyte character" on page 2001
- "wcsrtombs() — Convert wide-character string to multibyte string" on page 2021

mbsinit() — Test state object for initial state

Standards

Standards / Extensions	C or C++	Dependencies
ISO C Amendment C99 Single UNIX Specification, Version 3	both	

Format

Non-XPG4

```
#include <wchar.h>

int mbsinit(const mbstate_t *ps);
```

XPG4

```
#define _XOPEN_SOURCE
#define _MSE_PROTOS
#include <wchar.h>

int mbsinit(const mbstate_t *ps);
```

General description

If *ps* is not a NULL pointer the mbsinit() function determines whether the pointer to mbstate_t object describes an initial conversion state.

The behavior of this wide-character function is affected by the LC_CTYPE category of the current locale. If you change the category, undefined results can occur.

Special behavior for XPG4

If you define any feature test macro specifying XPG4 behavior before the statement in your program source file to include the wchar header, then you must also define the _MSE_PROTOS feature test macro to make the declaration of the mbsinit() function in the wchar header available when you compile your program. Please see Table 2 on page 4 for a list of XPG4 and other feature test macros.

Returned value

If *ps* is a NULL pointer or if the pointed-to object describes an initial conversion state, mbsinit() returns nonzero.

Otherwise, mbsinit() returns 0.

Example

CELEBM05

```
/* CELEBM05

   This example checks the conversion state to see if it is in the
   initial state.

 */
#include "stdio.h"
#include "wchar.h"
#include "stdlib.h"

main() {
   char     *string = "ABC";
   mbstate_t state = 0;
   wchar_t   wc;
   int   rc;

   rc = mbrtowc(&wc, string, MB_CUR_MAX, &state);
   if (mbsinit(&state))
     printf("In initial conversion state\n");
}
```

Related information

- “Internationalization: Locales and Character Sets” in *z/OS XL C/C++ Programming Guide*
- “locale.h” on page 40
- “wchar.h” on page 85
- “mbrlen() — Calculate length of multibyte character” on page 1043
- “mbrtowc() — Convert a multibyte character to a wide character” on page 1049
- “mbsrtowcs() — Convert a multibyte string to a wide-character string”
- “setlocale() — Set locale” on page 1552
- “wcrtomb() — Convert a wide character to a multibyte character” on page 2001
- “wcsrtombs() — Convert wide-character string to multibyte string” on page 2021

mbsrtowcs() — Convert a multibyte string to a wide-character string

Standards

Standards / Extensions	C or C++	Dependencies
ISO C Amendment C99 Single UNIX Specification, Version 3	both	

Format

Non-XPG4

```
#include <wchar.h>

size_t mbsrtowcs(wchar_t *dst, const char **src, size_t len, mbstate_t *ps);
```

XPG4

```
#define _XOPEN_SOURCE
#define _MSE_PROTOS
#include <wchar.h>

size_t mbsrtowcs(wchar_t *dst, const char **src, size_t len, mbstate_t *ps);
```

General description

Converts a sequence of multibyte characters that begins in the conversion state described by *ps* from the array indirectly pointed to by *src*. It converts this sequence into a sequence of corresponding wide characters, that, if *dst* is not a NULL pointer, are then stored into the array pointed to by *dst*. Conversion continues up to and including a terminating NULL character, and the terminating NULL wide character is also stored. Conversion stops earlier in two cases: (1) when a sequence of bytes is reached that does not form a valid multibyte character, or (2) if *dst* is not a NULL pointer, when *len* codes have been stored into the array pointed to by *dst*. Each conversion takes place as if by a call to the mbrtowc() function.

If *dst* is not a NULL pointer, the pointer object pointed to by *src* is assigned either a NULL pointer (if conversion stopped because a terminating NULL character was reached) or the address just past the last multibyte character converted. If conversion stopped because a terminating NULL character was reached, the resulting state is the initial state.

mbsrtowcs() is a restartable version of mbstowcs(). That is, shift-state information is passed as on of the arguments and is updated on exit. With mbsrtowcs(), you can switch from one multibyte string to another, provided that you have kept the shift-state information.

The behavior of this wide-character function is affected by the LC_CTYPE category of the current locale. If you change the category, undefined results can occur.

Special behavior for XPG4

If you define any feature test macro specifying XPG4 behavior before the statement in your program source file to include the wchar header, then you must also define the _MSE_PROTOS feature test macro to make the declaration of the mbsrtowcs() function in the wchar header available when you compile your program. Please see Table 2 on page 4 for a list of XPG4 and other feature test macros.

Returned value

If successful, mbsrtowcs() returns the number of multibyte characters converted, not including the terminating NULL character, if any.

If the *dst* argument is a null pointer, the *len* argument is ignored and mbsrtowcs() returns the required size in wide characters for the destination string.

If the input string contains an invalid multibyte character, mbsrtowcs() returns (size_t)-1 and sets errno to one of the following values:

Error Code
Description

EILSEQ
Encoding error (the conversion state is undefined).

Example

CELEBM06

```
/* CELEBM06 */
#include <stdio.h>
#include <stdlib.h>
#include <wchar.h>

#define SIZE 10

int main(void)
{
   wchar_t wcs[SIZE];
   char    mbs[SIZE]="abcd";  /* string containing the multibyte char */
   char   *ptr      = mbs;    /* pointer to the mbs string             */
   int     length;

   /* Determine the length of the multibyte string pointed to by       */
   /* mbs.  Store the multibyte characters in the wchar_t array        */
   /* pointed to by wcs.                                               */

   length = mbsrtowcs(wcs, (const char**)&ptr, SIZE, NULL);
   wcs[length] = L'\0';

   printf("    length: %d \n", length);
   printf("       wcs:\"%ls\"\n",wcs);
   printf("       mbs:\"%s\"\n",mbs);
   printf("      &mbs: %p \n", mbs);
   printf("      &ptr: %p \n", ptr);
}
```

Related information

- "Internationalization: Locales and Character Sets" in *z/OS XL C/C++ Programming Guide*
- "locale.h" on page 40
- "wchar.h" on page 85
- "mblen() — Calculate length of multibyte character" on page 1040
- "mbrlen() — Calculate length of multibyte character" on page 1043
- "mbrtowc() — Convert a multibyte character to a wide character" on page 1049
- "mbstowcs() — Convert multibyte characters to wide characters" on page 1055
- "setlocale() — Set locale" on page 1552
- "wcrtomb() — Convert a wide character to a multibyte character" on page 2001
- "wcsrtombs() — Convert wide-character string to multibyte string" on page 2021

mbstowcs() — Convert multibyte characters to wide characters

Standards

Standards / Extensions	C or C++	Dependencies
ISO C XPG4 XPG4.2 C99 Single UNIX Specification, Version 3	both	

Format

```
#include <stdlib.h>

size_t mbstowcs(wchar_t * __restrict__pwc, const char* __restrict__string, size_t n);
```

General description

Determines the length of the sequence of the multibyte characters that start in the initial shift state and that are pointed to by *string*. It then converts each of the multibyte characters to a `wchar_t`, and stores no more than *n* codes in the array pointed to by *pwc*. The conversion stops if either an invalid multibyte sequence is encountered or if *n* codes have been converted.

Processing continues up to and including the terminating NULL character, and characters that follow it are not processed. The terminating NULL character is converted into a code with the value 0.

The behavior of this wide-character function is affected by the LC_CTYPE category of the current locale. If you change the category, undefined results can occur.

If pwc is a NULL pointer, mbstowcs() will return the length required to convert the entire array regardless of the value of n, but no values are stored.

Returned value

If successful, mbstowcs() returns the number of *pwc* array elements modified (or required if *pwc* is NULL) , not counting the terminating 0 code (the `wchar_t` 0 code). Note that, if the return value is *n*, the resulting *wchar_t* array will *not* be NULL-terminated.

If an invalid multibyte character is encountered, mbstowcs() returns (size_t)-1.

Example

CELEBM07

```
/* CELEBM07

   This example uses &mbstowcs. to convert a multibyte character
   string to a wide character string.

 */
#include <stdio.h>
#include <stdlib.h>

int main()
{
    char      mbsin[8] = "\x50\x0e\x42\xf1\x0f\x50\x00";
    wchar_t   wcsout[5];
    size_t    wcssize;

    printf("mbsin is 0x%.2x 0x%.2x 0x%.2x 0x%.2x 0x%.2x 0x%.2x 0x%.2x\n",
            mbsin[0], mbsin[1], mbsin[2],
            mbsin[3], mbsin[4], mbsin[5],
            mbsin[6]);

    wcssize = mbstowcs(wcsout, mbsin, 5);

    printf("mbstowcs(wcsout, mbsin, 5); returned %d\n", wcssize);

    printf("wcsout is 0x%.4x 0x%.4x 0x%.4x 0x%.4x\n",
            wcsout[0], wcsout[1],
            wcsout[2], wcsout[3]);
}
```

Output

```
mbsin is 0x50 0x0e 0x42 0xf1 0x0f 0x50 0x00
mbstowcs(wcsout, mbsin, 5); returned 3
wcsout is 0x0050 0x42f1 0x0050 0x0000
```

Related information

- "Internationalization: Locales and Character Sets" in *z/OS XL C/C++ Programming Guide*
- "locale.h" on page 40
- "stdlib.h" on page 70
- "mblen() — Calculate length of multibyte character" on page 1040
- "mbsrtowcs() — Convert a multibyte string to a wide-character string" on page 1052
- "mbtowc() — Convert multibyte character to wide character"
- "setlocale() — Set locale" on page 1552
- "wcslen() — Calculate length of wide-character string" on page 2013
- "wcstombs() — Convert wide-character string to multibyte character string" on page 2041

mbtowc() — Convert multibyte character to wide character

Standards

Standards / Extensions	C or C++	Dependencies
ISO C XPG4 XPG4.2 C99 Single UNIX Specification, Version 3	both	

Format

```
#include <stdlib.h>

int mbtowc(wchar_t * __restrict__pwc, const char* __restrict__string, size_t n);
```

General description

Converts a multibyte character to a wide character and returns the number of bytes of the multibyte character. It first determines the length of the multibyte character pointed to by *string*. It then converts the multibyte character to the corresponding wide character and places the wide character in the location pointed to by *pwc*, if *pwc* is not a NULL pointer. A maximum of *n* bytes is examined.

The behavior of this wide-character function is affected by the LC_CTYPE category of the current locale. If you change the category, undefined results can occur.

Returned value

If *string* is NULL, mbtowc() returns:

- Nonzero if the multibyte encoding in the current locale (LC_CTYPE) is shift-dependent.
- 0 otherwise.
- The current shift state is set to the initial state.

Otherwise if *string* is not NULL, mbtowc() returns:

- The number of bytes comprising the converted multibyte character, if *n* or fewer bytes form a valid multibyte character.
- 0 if *string* points to the NULL character.
- -1 if *string* does not point to a valid multibyte character, and the next *n* bytes do not form a valid multibyte character.

If the current locale supports EBCDIC DBCS characters, the shift state is updated where applicable. The length returned may be up to 4 characters long (for the shift-out character, 2-byte code, and the shift-in character).

After the function is placed into its initial state, it interprets multibyte characters—pointed to by *string*—accordingly. During the processing of shift-dependent encoded characters, you cannot stop processing one string, then move temporarily to processing another string, and return to the first, because the state would be valid for the second string, not the place where you stopped in the first string.

Example

```
/* This example uses mbtowc() to convert a multibyte character into a wide
   character.
 */
#include <stdio.h>
#include <stdlib.h>

int temp;
char string [6];
wchar_t arr[6];

int main(void)
{ /* Set string to point to a multibyte character. */
⋮
   temp = mbtowc(arr, string, MB_CUR_MAX);
   printf("wide-character string: %ls",arr);
}
```

Related information

- "Internationalization: Locales and Character Sets" in *z/OS XL C/C++ Programming Guide*
- "locale.h" on page 40
- "stdlib.h" on page 70
- "mblen() — Calculate length of multibyte character" on page 1040
- "mbrtowc() — Convert a multibyte character to a wide character" on page 1049

- “mbstowcs() — Convert multibyte characters to wide characters” on page 1055
- “setlocale() — Set locale” on page 1552
- “wcslen() — Calculate length of wide-character string” on page 2013
- “wctomb() — Convert wide character to multibyte character” on page 2053

m_create_layout() — Create and initialize a layout object (bidi data)

Standards

Standards / Extensions	C or C++	Dependencies
ISO C XPG4 C99	both	z/OS V1R2

Format

```
#include <sys/layout.h>

LayoutObject m_create_layout(const AttrObject attrobj, const char *modifier);
```

General description

The m_create_layout() function is part of the support for handling of bidirectional (Bidi) conversion of data between Visual (MVS) and Implicit (z/OS UNIX) formats. Initial support is for Arabic and Hebrew data.

The m_create_layout() function creates a LayoutObject associated with the locale identified by *attrobj*. The LayoutObject is an opaque object containing all the data and methods necessary to perform the layout operations on context-dependent or directional characters of the locale identified by the *attrobj*.

The memory for the LayoutObject is allocated by m_create_layout(). The LayoutObject created has default layout values. If the *modifier* argument is not NULL, the layout values specified by the *modifier* will overwrite the default layout values associated with the locale. Also, internal states maintained by the layout transformation function across transformations are set to their initial values.

The *attrobj* argument is or may be an amalgam of many opaque objects. A locale object is just one example of the type of object that can be attached to an attribute object. The *attrobj* argument specifies a name that is usually associated with a locale category. If *attrobj* is NULL, the LayoutObject created is associated with the current locale as set by the setlocale() function.

The *modifier* argument can be used to announce a set of layout values when the LayoutObject is created.

A LayoutObject created by m_create_layout() is deleted by calling the m_destroy_layout() function.

For a detailed description of bidirectional layout transformation, see Bidirectional Language Support in *z/OS XL C/C++ Programming Guide*.

Returned value

If successful, m_create_layout() returns a LayoutObject for use in subsequent calls to m_*_layout() functions.

If unsuccessful, m_create_layout() returns (LayoutObject)0 and sets errno to one of the following values:

Error Code
Description

EBADF
The attribute object is invalid or the locale associated with the attribute object is not available.

EINVAL
The modifier string has a syntax error or it contains unknown layout values.

ENOMEM
Insufficient storage space is available.

Related information

- "sys/layout.h" on page 73
- "m_destroy_layout() — Destroy a layout object (bidi data)"
- "m_getvalues_layout() — Query layout values of a layout object (bidi data)" on page 1067
- "m_setvalues_layout() — Set layout values of a layout object (bidi data)" on page 1103
- "m_transform_layout() — Layout transformation for character strings (bidi data)" on page 1118
- "m_wtransform_layout() — Layout transformation for wide-character strings (bidi data)" on page 1125
- "setlocale() — Set locale" on page 1552

m_destroy_layout() — Destroy a layout object (bidi data)

Standards

Standards / Extensions	C or C++	Dependencies
ISO C XPG4 C99	both	z/OS V1R2

Format

```
#include <sys/layout.h>

int m_destroy_layout(const LayoutObject layoutobject);
```

General description

The m_destroy_layout() function is part of the support for handling of bidirectional (Bidi) conversion of data between Visual (MVS) and Implicit (z/OS UNIX) formats. Initial support is for Arabic and Hebrew data.

The m_destroy_layout() function destroys a LayoutObject by deallocating the layout object and all the associated resources previously allocated by the m_create_layout( ) function.

Returned value

If successful, m_destroy_layout() returns 0.

If unsuccessful, m_destroy_layout() returns -1 and sets errno to one of the following values:

Error Code
Description

EFAULT
Errors occurred while processing the request.

Related information

- “sys/layout.h” on page 73
- “m_destroy_layout() — Destroy a layout object (bidi data)” on page 1059
- “m_getvalues_layout() — Query layout values of a layout object (bidi data)” on page 1067
- “m_setvalues_layout() — Set layout values of a layout object (bidi data)” on page 1103
- “m_transform_layout() — Layout transformation for character strings (bidi data)” on page 1118
- “m_wtransform_layout() — Layout transformation for wide-character strings (bidi data)” on page 1125

memccpy() — Copy bytes in memory

Standards

Standards / Extensions	C or C++	Dependencies
XPG4 XPG4.2 Single UNIX Specification, Version 3	both	

Format

```
#define _XOPEN_SOURCE
#include <string.h>

void *memccpy(void *__restrict__ s1, const void *__restrict__ s2, int c, size_t n);
```

General description

The memccpy() function copies bytes from memory area s2 into memory area s1, stopping after the first occurrence of byte c (converted to an unsigned char) is copied, or after n bytes are copied, whichever comes first.

Returned value

If successful, memccpy() returns a pointer to the byte after the copy of *c* in *s1*.

If *c* was not found in the first *n* bytes of *s2*, memccpy() returns a NULL pointer.

Related information

- “string.h” on page 72
- “memchr() — Search buffer” on page 1061
- “memcmp() — Compare bytes” on page 1062
- “memcpy() — Copy buffer” on page 1063
- “memmove() — Move buffer” on page 1065
- “memset() — Set buffer to value” on page 1066
- “strchr() — Search for character” on page 1729

memchr() — Search buffer

Standards

Standards / Extensions	C or C++	Dependencies
ISO C XPG4 XPG4.2 C99 Single UNIX Specification, Version 3	both	

Format

```
#include <string.h>

void *memchr(const void *buf, int c, size_t count);
```

General description

The memchr() built-in function searches the first *count* bytes pointed to by *buf* for the first occurrence of *c* converted to an unsigned character. The search continues until it finds *c* or examines *count* bytes.

Returned value

If successful, memchr() returns a pointer to the location of *c* in *buf*.

If *c* is not within the first *count* bytes of *buf*, memchr() returns NULL.

Example

CELEBM11

```
/* CELEBM11

   This example finds the first occurrence of "x" in
   the string that you provide.
   If it is found, the string that starts with that character is
   printed.
   If you compile this code as MYPROG, then it could be invoked
   like this, with exactly one parameter:
        MYPROG skixing

 */
#include <stdio.h>
#include <string.h>

int main(int argc, char ** argv)
{
  char * result;

  if ( argc != 2 )
    printf( "Usage: %s string\n", argv[0] );
  else
  {
    if ((result = (char *)memchr( argv[1], 'x', strlen(argv[1])) ) != NULL)
      printf( "The string starting with x is %s\n", result );
    else
      printf( "The letter x cannot be found in the string\n" );
  }
}
```

Output

```
The string starting with x is xing
```

Related information

- "string.h" on page 72
- "memccpy() — Copy bytes in memory" on page 1060
- "memcmp() — Compare bytes"
- "memcpy() — Copy buffer" on page 1063
- "memmove() — Move buffer" on page 1065
- "memset() — Set buffer to value" on page 1066
- "strchr() — Search for character" on page 1729

memcmp() — Compare bytes

Standards

Standards / Extensions	C or C++	Dependencies
ISO C XPG4 XPG4.2 C99 Single UNIX Specification, Version 3	both	

Format

```
#include <string.h>

int memcmp(const void *buf1, const void *buf2, size_t count);
```

General description

The memcmp() built-in function compares the first *count* bytes of *buf1* and *buf2*.

The relation is determined by the sign of the difference between the values of the leftmost first pair of bytes that differ. The values depend on EBCDIC encoding. This function is *not* locale sensitive.

Returned value

Indicates the relationship between *buf1* and *buf2* as follows:

Value	Meaning
< 0	The contents of the buffer pointed to by *buf1* less than the contents of the buffer pointed to by *buf2*
= 0	The contents of the buffer pointed to by *buf1* identical to the contents of the buffer pointed to by *buf2*
> 0	The contents of the buffer pointed to by *buf1* greater than the contents of the buffer pointed to by *buf2*

Example

CELEBM12

```
/* CELEBM12

   This example compares first and second arguments passed to
   main to determine which, if either, is greater.
```

```
 */
#include <stdio.h>
#include <string.h>

int main(int argc, char ** argv)
{
  int  len;
  int  result;

  if ( argc != 3 )
  {
     printf( "Usage: %s string1 string2\n", argv[0] );
  }
  else
  {
     /* Determine the length to be used for comparison */
     if (strlen( argv[1] ) < strlen( argv[2] ))
       len = strlen( argv[1] );
     else
       len = strlen( argv[2] );

     result = memcmp( argv[1], argv[2], len );

     printf( "When the first %i characters are compared,\n", len );
     if ( result == 0 )
       printf( "\"%s\" is identical to \"%s\"\n", argv[1], argv[2] );
     else if ( result < 0 )
       printf( "\"%s\" is less than \"%s\"\n", argv[1], argv[2] );
     else
       printf( "\"%s\" is greater than \"%s\"\n", argv[1], argv[2] );
  }
}
```

Output

If the program is passed the arguments *firststring* and *secondstring*, you would obtain following:

```
When the first 11 characters are compared,
"firststring" is less than "secondstring"
```

Related information

- "string.h" on page 72
- "memccpy() — Copy bytes in memory" on page 1060
- "memchr() — Search buffer" on page 1061
- "memcpy() — Copy buffer"
- "memmove() — Move buffer" on page 1065
- "memset() — Set buffer to value" on page 1066
- "strcmp() — Compare strings" on page 1731

memcpy() — Copy buffer

Standards

Standards / Extensions	C or C++	Dependencies
ISO C XPG4 XPG4.2 C99 Single UNIX Specification, Version 3	both	

Format

```
#include <string.h>

void *memcpy(void * __restrict__dest, const void * __restrict__src, size_t count);
```

General description

The memcpy() built-in function copies *count* bytes from the object pointed to by *src* to the object pointed to by *dest*. See "Built-in functions" on page 93 for information about the use of built-in functions. For memcpy(), the source characters may be overlaid if copying takes place between objects that overlap. Use the memmove() function to allow copying between objects that overlap.

Returned value

memcpy() returns the value of *dest*.

Example

CELEBM13

```
/* CELEBM13

   This example copies the contents of source to target.

 */
#include <string.h>
#include <stdio.h>

#define MAX_LEN 80

char source[ MAX_LEN ] = "This is the source string";
char target[ MAX_LEN ] = "This is the target string";

int main(void)
{
  printf( "Before memcpy, target is \"%s\"\n", target );
  memcpy( target, source, sizeof(source));
  printf( "After memcpy, target becomes \"%s\"\n", target );
}
```

Output

```
Before memcpy, target is "This is the target string"
After memcpy, target becomes "This is the source string"
```

Related information

- "string.h" on page 72
- "memccpy() — Copy bytes in memory" on page 1060
- "memchr() — Search buffer" on page 1061
- "memcmp() — Compare bytes" on page 1062
- "memmove() — Move buffer" on page 1065
- "memset() — Set buffer to value" on page 1066
- "strcpy() — Copy string" on page 1734

memmove() — Move buffer

Standards

Standards / Extensions	C or C++	Dependencies
ISO C XPG4 XPG4.2 C99 Single UNIX Specification, Version 3	both	

Format

```
#include <string.h>

void *memmove(void *dest, const void *src, size_t count);
```

General description

Copies *count* bytes from the object pointed to by *src* to the object pointed to by *dest*. The memmove() function allows copying between possibly overlapping objects as if the *count* bytes of the object pointed to by *src* must first copied into a temporary array before being copied to the object pointed to by *dest*.

Returned value

memmove() returns the value of *dest*.

Example

CELEBM14

```
/* CELEBM14

   This example copies the word shiny from position target + 2
   to position target + 8.

 */
#include <string.h>
#include <stdio.h>
#define SIZE    21

char target[SIZE] = "a shiny white sphere";

int main( void )
{
  char * p = target + 8;  /* p points at the starting character
                          of the word we want to replace */
  char * source = target + 2; /* start of "shiny" */

  printf( "Before memmove, target is \"%s\"\n", target );
  memmove( p, source, 5 );
  printf( "After memmove, target becomes \"%s\"\n", target );
}
```

Output

```
Before memmove, target is "a shiny white sphere"
After memmove, target becomes "a shiny shiny sphere"
```

Related information

- "string.h" on page 72
- "memccpy() — Copy bytes in memory" on page 1060
- "memchr() — Search buffer" on page 1061
- "memcmp() — Compare bytes" on page 1062
- "memcpy() — Copy buffer" on page 1063
- "memset() — Set buffer to value"
- "strcpy() — Copy string" on page 1734

memset() — Set buffer to value

Standards

Standards / Extensions	C or C++	Dependencies
ISO C XPG4 XPG4.2 C99 Single UNIX Specification, Version 3	both	

Format

```
#include <string.h>

void *memset(void *dest, int c, size_t count);
```

General description

The memset() built-in function sets the first *count* bytes of *dest* to the value *c* converted to an unsigned int.

Returned value

memset() returns the value of *dest*.

Example

CELEBM15

```
/* CELEBM15

   This example sets 10 bytes of the buffer to "A" and
   the next 10 bytes to "B".

 */
#include <string.h>
#include <stdio.h>
#define  BUF_SIZE     20
#define  HALF_BUF_SIZE   BUF_SIZE/2

int main(void)
{
   char buffer[BUF_SIZE + 1];
   char *string;

   memset(buffer, 0, sizeof(buffer));
   string = (char *)memset(buffer,'A', HALF_BUF_SIZE);
   printf("\nBuffer contents: %s\n", string);
   memset(buffer+HALF_BUF_SIZE, 'B', HALF_BUF_SIZE);
   printf("\nBuffer contents: %s\n", buffer);
}
```

Output

```
Buffer contents: AAAAAAAAAA

Buffer contents: AAAAAAAAAABBBBBBBBBB
```

Related information

- “string.h” on page 72
- “memccpy() — Copy bytes in memory” on page 1060
- “memchr() — Search buffer” on page 1061
- “memcmp() — Compare bytes” on page 1062
- “memcpy() — Copy buffer” on page 1063
- “memmove() — Move buffer” on page 1065

m_getvalues_layout() — Query layout values of a layout object (bidi data)

Standards

Standards / Extensions	C or C++	Dependencies
ISO C XPG4 C99	both	z/OS V1R2

Format

```
#include <sys/layout.h>

int m_getvalues_layout(const LayoutObject layout_object, LayoutValues values,
                       int *index_returned);
```

General description

The m_getvalues_layout() function is part of the support for handling of bidirectional (Bidi) conversion of data between Visual (MVS) and Implicit (z/OS UNIX) formats. Initial support is for Arabic and Hebrew data.

The m_getvalues_layout() function is used to query the current setting of layout values within a LayoutObject. The *layout_object* argument specifies a LayoutObject returned by the m_create_layout() function. The *values* argument specifies the list of layout values which are to be queried.

Each value element of a LayoutValueRec must point to a location where the layout value is stored. For example, if a layout value is of type T, the argument must be of type T*. The values are queried from the LayoutObject and represent its current state.

If the layout value name has **QueryValueSize** OR-ed to it, instead of the value of the layout value, only its size is returned. This option can be used by the caller to determine the amount of memory needed to be allocated for the layout values queried.

It is the user's responsibility to manage the space allocation for the layout values queried.

For a detailed description of bidirectional layout transformation, see Bidirectional Language Support in *z/OS XL C/C++ Programming Guide*.

Returned value

If successful, m_getvalues_layout() returns 0.

If any value cannot be queried, m_getvalues_layout() stores into *index_returned* the (zero-based) index of the value causing the error. It returns -1 and sets errno to one of the following values:

Error Code
Description

EINVAL
The layout value specified by index_returned is unknown or its value is invalid or the argument layout_object is invalid.

Related information

- "sys/layout.h" on page 73
- "m_create_layout() — Create and initialize a layout object (bidi data)" on page 1058
- "m_destroy_layout() — Destroy a layout object (bidi data)" on page 1059
- "m_setvalues_layout() — Set layout values of a layout object (bidi data)" on page 1103
- "m_transform_layout() — Layout transformation for character strings (bidi data)" on page 1118
- "m_wtransform_layout() — Layout transformation for wide-character strings (bidi data)" on page 1125

mkdir() — Make a directory

Standards

Standards / Extensions	C or C++	Dependencies
POSIX.1 XPG4 XPG4.2 Single UNIX Specification, Version 3	both	

Format

```
#define _POSIX_SOURCE
#include <sys/stat.h>

int mkdir(const char *pathname, mode_t mode);
```

General description

Creates a new, empty directory, *pathname*. The file permission bits in *mode* are modified by the file creation mask of the process, and then used to set the file permission bits of the directory being created. For more information on the file creation mask, see "umask() — Set and retrieve file creation mask" on page 1947.

The *mode* argument is created with one of the flags defined in the sys/stat.h header file. Any mode flags that are not defined will be turned off, and the function will be allowed to proceed.

Flag **Description**

S_IRGRP
Read permission for the file's group.

S_IROTH
Read permission for users other than the file owner.

S_IRUSR
Read permission for the file owner.

S_IRWXG
Read, write, and search or execute permission for the file's group. S_IRWXG is the bitwise inclusive OR of S_IRGRP, S_IWGRP, and S_IXGRP.

S_IRWXO
Read, write, and search or execute permission for users other than the file owner. S_IRWXO is the bitwise inclusive OR of S_IROTH, S_IWOTH, and S_IXOTH.

S_IRWXU
Read, write, and search, or execute, for the file owner; S_IRWXU is the bitwise inclusive OR of S_IRUSR, S_IWUSR, and S_IXUSR.

S_ISGID
Privilege to set group ID (GID) for execution. When this file is run through an exec function, the effective group ID of the process is set to the group ID of the file. The process then has the same authority as the file owner, rather than the authority of the actual invoker.

S_ISUID
Privilege to set the user ID (UID) for execution. When this file is run through an exec function, the effective user ID of the process is set to the owner of the file. The process then has the same authority as the file owner, rather than the authority of the actual invoker.

S_ISVTX
Indicates shared text. Keep loaded as an executable file in storage.

S_IWGRP
Write permission for the file's group.

S_IWOTH
Write permission for users other than the file owner.

S_IWUSR
Write permission for the file owner.

S_IXGRP
Search permission (for a directory) or execute permission (for a file) for the file's group.

S_IXOTH
Search permission for a directory, or execute permission for a file, for users other than the file owner.

S_IXUSR
Search permission (for a directory) or execute permission (for a file) for the file owner.

The owner ID of the new directory is set to the effective user ID of the process. The group ID of the new directory is set to the group ID of the owning directory.

mkdir() sets the access, change, and modification times for the new directory. It also sets the change and modification times for the directory that contains the new directory.

If *pathname* names a symbolic link, mkdir() fails.

Returned value

If successful, mkdir() returns 0.

If unsuccessful, mkdir() does not create a directory, returns -1, and sets errno to one of the following values:

Error Code
: **Description**

EACCES
: The process did not have search permission on some component of *pathname*, or did not have write permission on the parent directory of the directory to be created.

EEXIST
: Either the named file refers to a symbolic link, or there is already a file or directory with the given *pathname*.

ELOOP
: A loop exists in symbolic links. This error is issued if more than POSIX_SYMLOOP (defined in the limits.h header file) symbolic links are detected in the resolution of *pathname*.

EMLINK
: The link count of the parent directory has already reached LINK_MAX (defined in the limits.h header file).

ENAMETOOLONG
: *pathname* is longer than **PATH_MAX** characters or some component of *pathname* is longer than **NAME_MAX** characters while _POSIX_NO_TRUNC is in effect. For symbolic links, the length of the pathname string substituted for a symbolic link exceeds **PATH_MAX**. The **PATH_MAX** and **NAME_MAX** values can be determined using pathconf().

ENOENT
: Some component of *pathname* does not exist, or *pathname* is an empty string.

ENOSPC
: The file system does not have enough space to contain a new directory, or the parent directory cannot be extended.

ENOTDIR
: A component of the *pathname* prefix is not a directory.

EROFS
: The parent directory of the directory to be created is on a read-only file system.

Example

CELEBM16

```
/* CELEBM16

   The following example creates a new directory.

 */
#define _POSIX_SOURCE
#include <sys/stat.h>
#include <unistd.h>
#undef _POSIX_SOURCE
#include <stdio.h>

main() {
  char new_dir[]="new_dir";

  if (mkdir(new_dir, S_IRWXU|S_IRGRP|S_IXGRP) != 0)
    perror("mkdir() error");
  else if (chdir(new_dir) != 0)
    perror("first chdir() error");
  else if (chdir("..") != 0)
    perror("second chdir() error");
  else if (rmdir(new_dir) != 0)
    perror("rmdir() error");
  else
    puts("success!");
}
```

Related information

- “limits.h” on page 39
- “sys/stat.h” on page 75
- “chdir() — Change the working directory” on page 267
- “chmod() — Change the mode of a file or directory” on page 272
- “stat() — Get file information” on page 1720
- “umask() — Set and retrieve file creation mask” on page 1947

mkfifo() — Make a FIFO special file

Standards

Standards / Extensions	C or C++	Dependencies
POSIX.1 XPG4 XPG4.2 Single UNIX Specification, Version 3	both	

Format

```
#define _POSIX_SOURCE
#include <sys/stat.h>

int mkfifo(const char *pathname, mode_t mode);
```

General description

Sets the access, change, and modification times for the new file. It also sets the change and modification times for the directory that contains the new file.

mkfifo() creates a new FIFO special file, *pathname*. The file permission bits in *mode* are changed by the file creation mask of the process, and then used to set the file permission bits of the FIFO file being created. If *pathname* contains a symbolic link, mkfifo() fails. For more information on the file creation mask, see “umask() — Set and retrieve file creation mask" on page 1947; for information about the file permission bits, see "chmod() — Change the mode of a file or directory" on page 272.

The owner ID of the FIFO file is set to the effective user ID of the process. The group ID of the FIFO file is set to the group ID of the owning directory. *pathname* cannot end in a symbolic link.

Returned value

If successful, mkfifo() returns 0.

If unsuccessful, mkfifo() does not create a FIFO file, returns -1, and sets errno to one of the following values:

Error Code
Description

EACCES
The process does not have search permission on some component of *pathname*, or does not have write permission on the parent directory of the file to be created.

EEXIST
Either the named file refers to a symbolic link, or there is already a file or directory with the given *pathname*.

EINTR
A signal is received while this open is blocked waiting for an open() for read (if O_WRONLY was specified) or for an open() for write (if O_RDONLY was specified).

ELOOP
A loop exists in symbolic links. This error is issued if more than POSIX_SYMLOOP (defined in the limits.h header file) symbolic links are detected in the resolution of *pathname*.

EMLINK
The link count of the parent directory has already reached the maximum defined for the system.

ENAMETOOLONG
pathname is longer than **PATH_MAX** characters or some component of *pathname* is longer than **NAME_MAX** characters while _POSIX_NO_TRUNC is in effect. For symbolic links, the length of the pathname string substituted for a symbolic link exceeds **PATH_MAX**. The **PATH_MAX** and **NAME_MAX** values can be determined using pathconf().

ENOENT
Some component of *pathname* does not exist, or *pathname* is an empty string.

ENOSPC
The file system does not have enough space to contain a new file, or the parent directory cannot be extended.

ENOTDIR
A component of the *pathname* prefix is not a directory.

EROFS
The parent directory of the FIFO file is on a read-only file system.

Example

CELEBM17

```
/* CELEBM17

   This example uses mkfifo() to create a FIFO specail file named
   temp.fifo and then writes and reads from the file before closing it.

 */
#define _POSIX_SOURCE
#include <sys/stat.h>
#include <unistd.h>
#include <fcntl.h>
#include <stdio.h>

main() {
  char fn[]="temp.fifo";
  char out[20]="FIFO's are fun!", in[20];
  int rfd, wfd;

  if (mkfifo(fn, S_IRWXU) != 0)
    perror("mkfifo() error");
  else {
    if ((rfd = open(fn, O_RDONLY|O_NONBLOCK)) < 0)
      perror("open() error for read end");
    else {
      if ((wfd = open(fn, O_WRONLY)) < 0)
        perror("open() error for write end");
      else {
        if (write(wfd, out, strlen(out)+1) == -1)
          perror("write() error");
        else if (read(rfd, in, sizeof(in)) == -1)
          perror("read() error");
        else printf("read '%s' from the FIFO\n", in);
        close(wfd);
      }
      close(rfd);
    }
    unlink(fn);
  }
}
```

Output

```
read 'FIFO's are fun!' from the FIFO
```

Related information

- “limits.h” on page 39
- “sys/stat.h” on page 75
- “chmod() — Change the mode of a file or directory” on page 272
- “open() — Open a file” on page 1152
- “pipe() — Create an unnamed pipe” on page 1179
- “stat() — Get file information” on page 1720
- “umask() — Set and retrieve file creation mask” on page 1947

mknod() — Make a directory or file

Standards

Standards / Extensions	C or C++	Dependencies
XPG4.2 Single UNIX Specification, Version 3 z/OS UNIX	both	

Format

_OPEN_SYS

```
#define _OPEN_SYS
#include <sys/stat.h>

int mknod(const char *path, mode_t mode, rdev_t dev_identifier);
```

_XOPEN_SOURCE_EXTENDED 1

```
#define _XOPEN_SOURCE_EXTENDED 1
#include <sys/stat.h>

int mknod(const char *path, mode_t mode, dev_t dev_identifier);
```

General description

Creates a new directory, regular file, character special file, or FIFO special file (named pipe), with the pathname specified in the *path* argument.

The first byte of the *mode* argument determines the file type of the file:

S_IFCHR
Character special file

S_IFFIFO
FIFO special file

S_IFREG
Regular file

S_IFDIR
Directory file

The file permission bits of the new file are initialized with the remaining bits in *mode* and changed by the file creation mask of the process. For more information on these symbols, refer to "chmod() — Change the mode of a file or directory" on page 272.

dev_identifier applies only to a character special file. It is ignored for the other file types. *dev_identifier* contains a value representing the device major and device minor numbers. The major number is contained in the high-order 16 bits; it identifies a device driver supporting a class of devices, such as interactive terminals. The minor number is contained in the low-order 16 bits of *dev_identifier*; it identifies a specific device within the class referred to by the device major number. With z/OS UNIX services, the device major numbers are:

1 Master pseudoterminal

2 Slave pseudoterminal

3 /dev/tty

4 /dev/null

5 /dev/fdn

6 Sockets

7 OCSRTY

8 OCSADMIN

9 "/dev/console"

Device major numbers 1,2 and 7: The device minor numbers range between 0 and one less than the maximum number of pseudoterminal pairs defined by the installation.

Device major numbers 3,4,6,8 and 9: The device minor number is ignored.

Device major number 5: The device minor number value represents the file descriptor to be referred to. For example, device minor 0 refers to file descriptor 0.

When it completes successfully, mknod() marks for update the following fields of the file: `st_atime`, `st_ctime`, and `st_mtime`. It also marks for update the `st_ctime` and `st_mtime` fields of the directory that contains the new file.

Returned value

If successful, mknod() returns 0.

If unsuccessful, mknod() returns -1 and sets errno to one of the following values:

Error Code
Description

EACCES
Search permission is denied on a component of *path*, or write permission is denied on the parent directory of the file to be created.

EEXIST
A file by that name already exists.

ELOOP
A loop exists in symbolic links. This error is issued if more than POSIX_SYMLOOP (defined in the limits.h header file) symbolic links are detected in the resolution of *path*.

EMLINK
The link count of the parent directory has already reached the maximum defined for the system.

ENAMETOOLONG
pathname is longer than **PATH_MAX** characters, or some component of *pathname* is longer than **NAME_MAX** characters while _POSIX_NO_TRUNC is in effect. For symbolic links, the length of the pathname string substituted for a symbolic link exceeds **PATH_MAX**. The **PATH_MAX** and **NAME_MAX** values can be determined through pathconf().

ENOENT
A component of *path* was not found, or no *path* was specified.

ENOSPC
The file system does not have enough space to contain a new directory, or the parent directory cannot be extended.

ENOTDIR
A component of *path* is not a directory

EPERM
Added for XPG4.2: The invoking process does not have appropriate privileges and the file type is not FIFO-special.

EROFS
The file named in *path* cannot be created, because it would reside on a read-only file system.

Example

CELEBM18

```
/* CELEBM18 */
#include <sys/stat.h>
#include <unistd.h>
#include <stdio.h>

#define master 0x00010000

main() {
  char fn[]="char   ec";

  if (mknod(fn, S_IFCHR|S_IRUSR|S_IWUSR, master|0x0001) != 0)
    perror("mknod() error");
  else if (unlink(fn) != 0)
    perror("unlink() error");
}
```

Related information

- "limits.h" on page 39
- "sys/stat.h" on page 75
- "mkdir() — Make a directory" on page 1068
- "mkfifo() — Make a FIFO special file" on page 1071
- "open() — Open a file" on page 1152

mkstemp() — Make a unique filename

Standards

Standards / Extensions	C or C++	Dependencies
XPG4.2 Single UNIX Specification, Version 3	both	POSIX(ON)

Format

```
#define _XOPEN_SOURCE_EXTENDED 1
#include <stdlib.h>

int mkstemp(char *template);
```

General description

The mkstemp() function replaces the contents of the string pointed to by *template* with a unique file name, and returns a file descriptor for the file open for reading and writing. The function thus prevents any possible race condition between testing whether the file exists and opening it for use. The string in *template* should look like a file name with six trailing 'X's; mkstemp() replaces each 'X' with a character from the portable file name character set. The characters are chosen such that the resulting name does not duplicate the name of an existing file. This function is supported only in a POSIX program.

Returned value

If successful, mkstemp() returns an open file descriptor.

If no suitable file could be created, mkstemp() returns -1.

There are no errno values defined.

Related information

- “stdlib.h” on page 70
- “mktemp() — Make a unique file name”

mktemp() — Make a unique file name

Standards

Standards / Extensions	C or C++	Dependencies
XPG4.2 Single UNIX Specification, Version 3	both	POSIX(ON)

Format

```
#define _XOPEN_SOURCE_EXTENDED 1
#include <stdlib.h>

char *mktemp(char *template);
```

General description

The mktemp() function replaces the contents of the string pointed by *template* by a unique file name and returns *template*. The application must initialize *template* to be a file name with six trailing 'X's; mktemp() replaces each 'X' with a single-byte character from the portable file name character set.

This function is supported only in a POSIX program.

Note: The mktemp() function has been moved to the Legacy Option group in Single UNIX Specification, Version 3 and may be withdrawn in a future version. The mkstemp() function is preferred for portability and greater reliability.

Returned value

If successful, mktemp() returns a pointer to *template*.

If a unique name cannot be created, mktemp() sets *template* to a NULL string.

There are no errno values defined.

Related information

- "stdlib.h" on page 70
- "mkstemp() — Make a unique filename" on page 1076

mktime(), mktime64() — Convert local time

Standards

Standards / Extensions	C or C++	Dependencies
ISO C POSIX.1 XPG4 XPG4.2 C99 Single UNIX Specification, Version 3 Language Environment	both	

Format

```
#include <time.h>

time_t mktime(struct tm *tmptr);
```

```
#define _LARGE_TIME_API
#include <time.h>

time64_t mktime64 (struct tm *tmptr);
```

Compile requirement: Use of mktime64() function requires the long long data type. For more information on how to make long long data type available, see *z/OS XL C/C++ Language Reference*.

General description

The mktime() function converts broken-down time, expressed as a local time, in the tm structure (described in Table 16 on page 80) pointed to by *tmptr*, to calendar time. Calendar time is the number of seconds since epoch, which was at 00:00:00 Coordinated Universal Time (UTC), January 1, 1970.

The values of the structure members passed to the mktime() function are not restricted to the ranges described in "time.h" on page 79. For values that are outside the specified range, the mktime() function will use date arithmetic to adjust the values to produce a valid date and time (see "Example" on page 1080). The values of *tm_wday* and *tm_yday* are ignored and are assigned their correct values on return.

On the successful completion of the function, all the members of the structure pointed to by *time* are set to represent the specified time since the Epoch with their values forced into the ranges described in "time.h" on page 79. The value of *tm_wday* is set after the values of *tm_mon* and *tm_year* are determined. The output value of *tm_isdst* is determined by the mktime() function, independent of the value presented as input to the mktime() function.

If an application uses the localtime() function to determine local time, the localtime() function will determine if daylight savings applies (assuming DST information is available) and will correctly set the tm_isdst flag. If the application wants to determine seconds from Epoch corresponding to a `tm` structure returned by the localtime() function, it should not modify the tm_isdst flag set by the localtime() function.

If an application sets tm_isdst to 0 before calling the mktime() function, it is asserting that daylight savings does not apply to the input values of the `tm` structure, regardless of the system DST start and end dates. Likewise, if the application has set a value for tm_isdst to be greater than 0, it is asserting that the time represented by the `tm` structure has been shifted for daylight savings. Therefore, the mktime() function unshifts the time in determining seconds since Epoch.

Setting tm_isdst to -1 indicates the mktime() function will determine whether daylight savings time applies. If yes, the mktime() function returns tm_isdst greater than 0. If not, it returns tm_isdst of 0 unless DST information is not available on the system, in which case tm_isdst of -1 is returned.

Your time zone might not be using DST, perhaps because the TZ environment variable does not specify a daylight savings time name or perhaps because DSTNAME is unspecified in the current LC_TOD locale category. In such a case, if you code `tm_isdst=1` and call the mktime() function, `(time-t)-1` is returned to indicate an error.

The mktime64() function behaves exactly like the mktime() function except it will support a structured date beyond 00:00:00 UTC January 1, 2038 with a limit of 23:59:59 UTC on December 31, 9999.

Returned value

If successful, the mktime() function returns a *time_t* representing seconds since the Epoch, The number of seconds corresponds to the broken-down time, expressed as local time, based on the user-supplied *tm* structure pointed to by *tmptr*.

If the mktime() function cannot convert the broken-down time to a calendar time, it returns `(time_t)-1` to indicate an error, such as time before January 1, 1970 (UTC).

Error Code
Description

EOVERFLOW
The result cannot be represented.

Usage notes

1. The ctime(), localtime(), and mktime() functions now return Coordinated Universal Time (UTC) unless customized locale information is made available, which includes setting the `timezone_name` variable.
2. Applications working with local time may define time zone information by using the TZ (POSIX) or _TZ (non-POSIX) environment variable or by customizing the LC_TOD category of the locale in use.
3. When neither TZ nor _TZ is defined, the current locale is interrogated for time zone information. If neither TZ nor _TZ is defined and LC_TOD time zone information is not present in the current locale, a default value is applied to local time. POSIX programs simply default to Coordinated Universal Time (UTC), while non-POSIX programs establish an offset from UTC based on the

setting of the system clock. For more information about customizing a time zone to work with local time, see “Customizing a time zone” in *z/OS XL C/C++ Programming Guide*.

4. The mktime() functions fails when a result overflows the time_t object used to return the number of seconds elapsed from the time in *tmptr* back to the start of the standard epoch. In 31-bit, the last year that mktime() supports is 2037. In 64-bit, the time_t grows from 4 bytes to 8 bytes in length, so that mktime() can accommodate dates further into the future. The upper bound in 64-bit is set to the year 9999.

Example

CELEBM19

```
/* CELEBM19

   This example prints the day of the week that is 40 days and
   16 hours from the current date.

 */
#include <stdio.h>
#include <time.h>

char *wday[] = { "Sunday", "Monday", "Tuesday", "Wednesday",
                 "Thursday", "Friday", "Saturday" };

int main(void)
{
  time_t t1, t3;
  struct tm *t2;

  t1 = time(NULL);
  t2 = localtime(&t1);
  t2 -> tm_mday += 40;
  t2 -> tm_hour += 16;
  t3 = mktime(t2);

  printf("40 days and 16 hours from now, it will be a %s \n",
         wday[t2 -> tm_wday]);
}
```

Output

```
40 days and 16 hours from now, it will be a Sunday
```

Related information

- “time.h” on page 79
- “asctime(), asctime64() — Convert time to character string” on page 182
- “asctime_r(), asctime64_r() — Convert date and time to a character string” on page 184
- “ctime(), ctime64() — Convert time to character string” on page 359
- “ctime_r(), ctime64_r() — Convert time value to date and time character string” on page 362
- “gmtime(), gmtime64() — Convert time to broken-down UTC time” on page 810
- “gmtime_r(), gmtime64_r() — Convert a time value to broken-down UTC time” on page 812
- “localdtconv() — Date and time formatting convention inquiry” on page 981
- “localtime(), localtime64() — Convert time and correct for local time” on page 984
- “localtime_r(), localtime64_r() — Convert time value to broken-down local time” on page 986
- “strftime() — Convert to formatted time” on page 1743

- “time(),time64() — Determine current UTC time” on page 1878
- “tzset() — Set the time zone” on page 1936

__mlockall() — Lock the address space of a process

Standards

Standards / Extensions	C or C++	Dependencies
z/OS UNIX	both	

Format

```
#define _OPEN_SOURCE_EXTENDED 2
#include <sys/mman.h>

int __mlockall(int flags);
```

General description

The function__mlockall() causes all of the pages mapped by the address space of a process to be memory resident until unlocked or until the process exits or execs another process image. The *flags* argument determines whether the pages are to be locked or unlocked.

Flags Meaning

_BPX_NONSWAP
Lock the current pages mapped for this address space.

_BPX_SWAP
Unlock the current pages previously locked.

Returned value

If successful, __mlockall() returns 0.

If unsuccessful, __mlockall() returns -1 and no change is made to the memory state of the address space.

Note: This function will return a EINVAL with an errno2() of 09300be if the kernel is not available.

Related information

- “sys/mman.h” on page 73

mmap() — Map pages of memory

Standards

Standards / Extensions	C or C++	Dependencies
XPG4.2 Single UNIX Specification, Version 3 z/OS UNIX	both	

Format

```
#define _XOPEN_SOURCE_EXTENDED 1
#include <sys/mman.h>

void *mmap(void *addr, size_t len, int prot, int flags, int fildes, off_t off);
```

General description

The mmap() function establishes a mapping between an address space of a process (for *len* bytes) and a file associated with the file descriptor *fildes* at offset *off* for *len* bytes. The format of the call is as follows:

```
pa=mmap(addr, len, prot, flags, fildes, off);
```

The value of *pa* is an unspecified function of the argument *addr* and values of *flags*, further described below. A successful mmap() call returns *pa* as its results. The address ranges covered by [*pa, pa* + *len*] and [*off, off* + *len*] must be legitimate for the possible (not necessarily current) address space of a process and the file, respectively.

If the size of the mapped file changes after the call to mmap(), the effect of references to portions of the mapped region that correspond to added or removed portions of the file is unspecified.

The *prot* argument determines whether read, write, execute, or some combination of accesses are permitted to the pages being mapped. The protection options are defined in <sys/mman.h>:

PROT_READ
Page can be read

PROT_WRITE
Page can be written

PROT_EXEC
Page can be executed

PROT_NONE
Page can be accessed

Implementations need not enforce all combinations of access permissions. However, write shall only be permitted when PROT_WRITE has been set.

The *flags* argument provides other information about the handling of the mapped pages. The options are defined in <sys/mman.h>:

MAP_SHARED
Share changes

MAP_PRIVATE
Changes are private

MAP_FIXED
Interpret addr exactly

__MAP_MEGA
Map in megabyte increments

The MAP_PRIVATE and MAP_SHARED flags control the visibility of write references to the memory region. Exactly one of these flags must be specified. The mapping type is retained across a *fork*.

If MAP_SHARED is set in *flags*, write references to the memory region by the calling process may change the file and are visible in all MAP_SHARED mappings of the same portion of the file by any process.

If MAP_PRIVATE is set in *flags*, write references to the memory region by the calling process do not change the file and are not visible to any process in other mappings of the same portion of the file.

All changes to the mapped data made by processes that have mapped the memory region using MAP_SHARED are shared and are visible to all other processes that have mapped the same file-offset range.

When MAP_FIXED is set in the *flags* argument, the implementation is informed that the value of *pa* must be *addr*, exactly. If MAP_FIXED is set, mmap() may return (void*)-1 and set errno to EINVAL. If a MAP_FIXED request is successful, the mapping established by mmap() replaces any previous mappings for the process's pages in the range [*pa, pa* + *len*].

When MAP_FIXED is not set, the implementation uses *addr* in an unspecified manner to arrive to *pa.* The *pa* so chosen will be an area of the address space which the implementation deems suitable for a mapping of *len* bytes to the file. All implementation interpret an *addr* value of 0 as granting the implementation complete freedom in selecting *pa*, subject to constraints described below. A nonzero value of *addr* is taken to be a suggestion of a process address near which the mapping should be placed. When the implementation selects a value for *pa*, it never places a mapping at address 0, nor does it replace any extant mapping, nor map into dynamic memory allocation areas.

The *off* argument is constrained to be aligned and sized according to the value returned by sysconf() when passed_SC_PAGESIZE or_SC_PAGE_SIZE. When MAP_FIXED is specified, the argument *addr* must also meet these constraints. The implementation performs mapping operations over whole pages. Thus, while the argument *len* need not meet a size or alignment constraint, the implementation will include, in any mapping operation, any partial page specified by the range [*pa*, *pa* + *len*].

The implementation always zero-fills any partial page at the end of a memory region. Further, the implementation never writes out any modified portions of the last page of a file that are beyond the end of the mapped portion of the file. If the mapping established by mmap() extends into pages beyond the page containing the last byte of the file, an application references to any of the pages in the mapping that are beyond the last page results in the delivery of a SIGBUS or SIGSEGV signal.

The mmap() function adds an extra reference to the file associated with the file descriptor *fildes* which is not removed by a subsequent close() on that file descriptor. This reference is removed when there are not more mappings to the file.

The st_atime field of the mapped file may be marked for update at any time between the mmap() call and the corresponding munmap() call. The initial read or write reference to a mapped region will cause the file's set_atime field to be marked for update if it has not already been marked for update.

The st_ctime and st_mtime fields of a file that is mapped with MAP_SHARED and PROT_WRITE, will be marked for update at stime point in the interval between a write reference to the mapped region and the next call to msync() with

MS_ASYNC or MS_SYNC for that portion of the file by any process. If there is no such call, these fields may be marked for update at any time after a write reference if the underlying file is modified as a result.

If a memory mapped region is not unmapped before the process terminates, process termination will not automatically write out to disk any modified data in the mapped region. Modified private data in a MAP_PRIVATE region will be discarded. If the map region is MAP_SHARED, the modified data will continue to reside in the cache (if the same file-offset range is being shared) and may ultimately be written out to disk by another process using the msync() service. However, if no other processes map the same file-offset range as MAP_SHARED, the modified data is discarded.

There may be implementation-dependent limits on the number of memory regions that can be mapped (per process or per system). If such a limit is imposed, whether the number of memory regions that can be mapped by a process is decreased by the use of shmat() is implementation-dependent.

Specification of the __MAP_MEGA option results in the system allocating storage to map the file in megabyte increments. This option should only be used for large files. Any file over half a megabyte in size will likely achieve better performance by using this option. When using this option, mmaps and munmaps are in megabyte ranges on megabyte boundaries.

When __MAP_MEGA is specified, all changes to the mapped data are shared. Modifications to the mapped data are visible to all other processes that map the same file-offset range. That is, __MAP_MEGA behaves much like MAP_SHARED. __MAP_MEGA is mutually exclusive with MAP_PRIVATE and MAP_SHARED. Specification of __MAP_MEGA with either MAP_PRIVATE or MAP_SHARED will result in the request failing with errno set to EINVAL.

The __MAP_MEGA option may be specified with MAP_FIXED.

Map_address parameter: If the map address is not zero and __MAP_MEGA has been specified, then for non MAP_FIXED requests, the kernel will attempt to create the mapping at the map_address, truncated to the nearest megabyte boundary. If unsuccessful, it will proceed as if a map_address of zero were specified. For MAP_FIXED requests, the value of map_address must be multiples of the segment size (megabyte multiples). If not, the mmap request fails with errno set to EINVAL.

Map_length parameter: When __MAP_MEGA is specified, mapping operations are performed over whole segments (megabyte chunks). If the length is not a multiple of the segment size, the entire trailing portion of the last segment will also be mapped into the user storage.

File_descriptor: The file descriptor identifies the file being mapped. If an attempt is made to map a file that is already mapped but was mapped with a different specification of __MAP_MEGA, then the current request fails with errno set to EINVAL-MMapTypeMismatch. That is, at any point in time a file may be mapped with the __MAP_MEGA option or without the __MAP_MEGA option but not both ways at the same time.

For a __MAP_MEGA mapping, if this is the first map to the file represented by the specified file descriptor then whether the file was opened for read or for write will determine what protection options may be specified for the file by this mmap and any future mmaps and mprotects, by this or any other process mapping to the same file. If the file was opened for read but not write then only PROT_READ, PROT_EXEC or PROT_NONE will be allowed. If the file was opened for write, then any of the protection options will be allowed. Only regular files may be mapped. Note also that remote files accessed through NFS or DFS may not be mapped.

Protect_options: The specification made for Protect_options has a global effect when the file is mapped with the __MAP_MEGA option. The Protect_option specified immediately effects all processes currently mapped to the same file-offset range.

Large file support for z/OS UNIX files: Large z/OS UNIX files are supported automatically for AMODE 64 C/C++ applications. AMODE 31 C/C++ applications must be compiled with the option LANGLVL(LONGLONG) and define the _LARGE_FILES feature test macro before any headers are included to enable this function to operate on z/OS UNIX files that are larger than 2 GB in size. File size and offset fields are enlarged to 63 bits in width. Therefore, any other function operating on the file is required to define the _LARGE_FILES feature test macro as well.

Usage notes

1. The mmap() function does not support shared memory objects or typed memory objects.

Returned value

If successful, mmap() returns the address at which the mapping was placed.

If unsuccessful, mmap() returns MAP_FAILED and sets errno to one of the following values:

Error Code
: **Description**

EACCES
: The *fildes* argument is not open for read, regardless of the protection specified, or *fildes* is not open for write and PROT_WRITE was specified for a MAP_SHARED type mapping.

EBADF
: The *fildes* argument is not a valid open file descriptor.

EINVAL
: The *addr* argument (if MAP_FIXED was specified) or *off* is not a multiple of the page size as returned by sysconf(), or are considered invalid by the implementation.

The value of *flags* is invalid (neither MAP_PRIVATE nor MAP_SHARED is set).

EMFILE
: The number of mapped regions would exceed an implementation-dependent limit (per process or per system).

ENODEV
: The *fildes* argument refers to a file whose type is not supported by mmap().

ENOMEM
: MAP_FIXED was specified, and the range [*addr, addr* + range [*addr, addr* +

len], rounding *len*] exceeds that allowed for the address space for a process; or if MAP_FIXED was not specified and there is insufficient room in the address space to effect the mapping.

ENXIO
Address in the range [*off*, *off* + *len*] are invalid for *fildes*

EOVERFLOW
The file is a regular file and the value of off plus len exceeds the offset maximum established in the open file.

Note: Starting with z/OS V1.9, environment variable _EDC_EOVERFLOW can be used to control behavior of mmap() with respect to detecting an EOVERFLOW condition for z/OS UNIX files. By default, mmap() will not set EOVERFLOW when the offset maximum is exceeded associated with fildes. When _EDC_EOVERFLOW is set to YES, mmap() will check for an overflow condition.

Related information

- "sys/mman.h" on page 73
- "exec functions" on page 436
- "fcntl() — Control open file descriptors" on page 474
- "fork() — Create a new process" on page 571
- "lockf() — Record locking on files" on page 987
- "mprotect() — Set protection of memory mapping" on page 1100
- "msync() — Synchronize memory with physical storage" on page 1117
- "munmap() — Unmap pages of memory" on page 1122
- "shmat() — Shared memory attach operation" on page 1598
- "sysconf() — Determine system configuration options" on page 1798

modf(), modff(), modfl() — Extract fractional and integral parts of floating-point value

Standards

Standards / Extensions	C or C++	Dependencies
ISO C POSIX.1 XPG4 XPG4.2 ISO/ANSI C++ C99 Single UNIX Specification, Version 3 C++ TR1 C99	both	

Format

```
#include <math.h>

double modf(double x, double *intptr);
float modf(float x, int *intptr);               /* C++ only */
long double modf(long double x, int *intptr);   /* C++ only */
float modff(float x, int *intptr);
long double modfl(long double x, int *intptr);
```

General description

Breaks down the floating-point value *x* into fractional and integral parts. The integral part is stored as double, in the object pointed to by *intptr*. Both the fractional and integral parts are given the same sign as *x*.

Restriction

The modff() function does not support the _FP_MODE_VARIABLE feature test macro.

Returned value

Returns the signed fractional portion of *x*.

Example

CELEBM20

```
/* CELEBM20

   This example breaks the floating-point number -14.876 into
   its fractional and integral components.

 */
#include <math.h>
#include <stdio.h>

int main(void)
{
   double x, y, d;

   x = -14.876;
   y = modf(x, &d);

   printf("x = %lf\n", x);
   printf("Integral part = %lf\n", d);
   printf("Fractional part = %lf\n", y);
}
```

Output

```
x = -14.876000
Integral part = -14.000000
Fractional part = -0.876000
```

Related information

- “math.h” on page 44
- “fmod(), fmodf(), fmodl() — Calculate floating-point remainder” on page 559
- “frexp(), frexpf(), frexpl() — Extract mantissa and exponent of the floating-point value” on page 623
- “ldexp(), ldexpf(), ldexpl() — Multiply by a power of two” on page 941

modfd32(), modfd64(), modfd128() — Extract fractional and integral parts of decimal floating-point value

Standards

Standards / Extensions	C or C++	Dependencies
C/C++ DFP	both	z/OS V1.8

Format

```
#define __STDC_WANT_DEC_FP__
#include <math.h>

_Decimal32  modfd32(_Decimal32 x, _Decimal32 *p);
_Decimal64  modfd64(_Decimal64 x, _Decimal64 *x);
_Decimal128 modfd128(_Decimal128 x, _Decimal128 *x);
_Decimal32  modf(_Decimal32 x, _Decimal32 *x);   /* C++ only */
_Decimal64  modf(_Decimal64 x, _Decimal64 *x);   /* C++ only */
_Decimal128 modf(_Decimal128 x, _Decimal128 *x); /* C++ only */
```

General description

Breaks down the decimal floating-point value x into fractional and integral parts. The integral part is stored in the object point to by p. Both the fractional and integral parts are given the same sign as x.

Note:

- To use IEEE decimal floating-point, the hardware must have the Decimal Floating-Point Facility installed.
- These functions work in IEEE decimal floating-point format. See "IEEE Decimal Floating-Point" for more information.

Returned value

The lround functions return the signed fractional portion of x.

If the rounded value is outside the range of the return type, the numeric result is unspecified. A range error may occur if the magnitude of x is too large.

Example

```
/* CELEBM24

   This example illustrates the modfd128() function.

   This example breaks the floating-point number -14.876 into
   its fractional and integral components.

*/

#define  __STDC_WANT_DEC_FP__
#include <math.h>
#include <stdio.h>

int main(void)
{
  _Decimal128 x, y, d;

  x = -14.876DL;
  y = modfd128(x, &d);

  printf("Number          = %DDf\n", x);
  printf("Integral part   = %DDf\n", d);
  printf("Fractional part = %DDf\n", y);
}
```

Related information

- “math.h” on page 44
- “frexpd32(), frexpd64(), frexpd128() — Extract mantissa and exponent of the decimal floating-point value” on page 624

- “ldexpd32(), ldexpd64(), ldexpd128() — Multiply by a power of ten” on page 942
- “modf(), modff(), modfl() — Extract fractional and integral parts of floating-point value” on page 1086

__moservices() - Memory object services

Standards

Standards / Extensions	C or C++	Dependencies
AMODE 64 Language Environment	both	z/OS V1.10

Format

```
#include <stdlib.h>

int __moservices(int reqtype, size_t mopllen,
                 struct __mopl_s * mopl, void ** moorigin);
```

General description

The __moservices() function allows the caller to perform the following memory object services:

- Create a memory object that will be associated with the current Language Environment enclave;
- Free a memory object that was created using a previous __moservices() call;
- Specify a shared memory dump priority to be used when allocating shared memory.

Argument
Description

reqtype

The request type. Valid values are

__MO_GETSTOR
Use IARV64 REQUEST(GETSTOR) to create a memory object that is associated with the current Language Environment enclave.

__MO_DETACH
Use IARV64 REQUEST(DETACH) to free a memory object that was created using __moservices().

__MO_SHMDUMPPRIORITY
Set a shared memory dump priority, to be associated with shared memory segments on subsequent shmget() calls.

mopllen
The length of the passed mopl structure. If a mopl is not passed, this length should be set to zero.

mopl

Points to a structure describing additional characteristics for the request.

For __MO_GETSTOR and __MO_SHMDUMPPRIORITY requests, this structure is required.

For __MO_DETACH requests, this structure is optional.

moorigin

On __MO_GETSTOR calls, moorigin contains the origin address of the memory object upon return.

On __MO_DETACH calls, moorigin contains the origin address of the memory object to be detached.

The __mopl_s structure is defined in stdlib.h and has the following format:

```
struct __mopl_s {
     unsigned long   __moplrequestsize;
     int     __mopldumppriority;
     unsigned int      __moplgetstorflags;
     long     __moplreserved;
     int     __mopl_iarv64_rc;
     int     __mopl_iarv64_rsn;
  };
```

The entire __mopl_s structure must be set to binary zeroes prior to use. The fields are used as follows:

__mopldumppriority	For __MO_GETSTOR, the desired memory object dump priority (1-99). For __MO_SHMDUMPPRIORITY, the desired shared memory dump priority (1-99).
__moplsize	For __MO_GETSTOR, the size of the memory object to be created. The size is in 1MB units when 4K or 1MB page frames are requested. The size is in 2GB units when 2GB page frames are requested.
__moplstorflags	For __MO_GETSTOR, flags identifying additional characteristics for the memory object to be created. The following flags are defined: • __MOPL_PAGEFRAMESIZE1MEG - the memory object will be backed by fixed 1MB page frames. __moservices() returns -1 with errno EMVSERR if the current hardware does not have Large Page support or if large page frames are not available on the system. • __MOPL_PAGEFRAMESIZEMAX - the memory object should be backed by fixed 1MB page frames. If the request cannot be backed by fixed 1MB page frames due to unavailability of large page frames, then the request will be backed by 4K page frames. When 4K page frames are used, the request will be successful, but __mopl_iarv64_rc and __mopl_iarv64_rsn will contain values to indicate that the memory object is not backed by fixed 1MB page frames. • __MOPL_PAGEFRAMESIZE_PAGEABLE1MEG - the memory object should be preferentially backed by pageable 1MB page frames. Pageable 1MB page frames will be backed at first reference. If pageable 1MB page frames are not available at first reference, pageable 4K page frames will be used. • __MOPL_PAGEFRAMESIZE_2G - the memory object should be backed by 2GB page frames. __moservices() returns -1 with errno EMVSERR if the current hardware does not have Large Page support or if large page frames are not available on the system. **Note:** If none of the above flags are set, then the memory object will be backed by 4K page frames. Also, the above flags are mutually exclusive.
__mopliarv64rc	The return code from the call to IARV64
__mopliarv64rsn	The reason code from the call to IARV64

Constants are available which identify the dump priorities of memory objects obtained by Language Environment:

__MO_DUMP_PRIORITY_STACK	Dump priority of memory objects to be used as Language Environment stacks. Value is 5.
__MO_DUMP_PRIORITY_HEAP	Dump priority of memory objects to be used as Language Environment heaps. Value is 15.

For the dump priority of memory objects obtained by other programs (such as Java), refer to the corresponding documentation for those programs.

All memory objects obtained by __moservices() are unguarded and count towards the address space memlimit. For more information on memory objects and the IARV64 service, See *z/OS MVS Programming: Assembler Services Guide* and the *z/OS MVS Programming: Assembler Services Reference ABE-HSP*, *z/OS MVS Programming: Assembler Services Reference IAR-XCT*.

Returned value

If successful, __moservices() returns 0.

If unsuccessful, __moservices() returns -1 and sets errno to one of the following values:

Error Code
  Description

EINVAL
  An argument to this function contained an incorrect value. Use __errno2() to obtain more detailed information on the error.

EMVSERR
  A z/OS environmental or internal error occurred. Use __errno2() to obtain more detailed information on the error.

  If the underlying IARV64 call is unsuccessful, and the mopl argument has been specified, then __mopl_iarv64_rc and __mopl_iarv64_rsn will contain the return and reason codes returned by IARV64.

Example

```
/* CELEBM25

   This example illustrates the __moservices() function.

   This example sets a shared memory dump priority, then
   gets and detaches a memory object.

*/

#include <stdlib.h>
#include <stdio.h>
#include <errno.h>
#include <string.h>

int main(void)  {

    __mopl_t mymopl;
    int  mos_rv;
    void * mymoptr;

    memset(&mymopl, 0, sizeof(__mopl_t));

    /* Set a shared memory dump priority for subsequent shmget()     */
    mymopl.__mopldumppriority = 8;

    mos_rv = __moservices(__MO_SHMDUMPPRIORITY, sizeof(mymopl),
                          &mymopl , &mymoptr);

    if (mos_rv != 0) {
      perror("moservices(SHMDUMPPRIORITY) call failed");
    }

    /* Obtain a 100MB memory object whose dump priority falls        */
    /* between the dump priorities of the Language Environment       */
    /* stacks and heaps.                                             */
    mymopl.__mopldumppriority = __MO_DUMP_PRIORITY_STACK + 5;
    mymopl.__moplrequestsize = 100;

    mos_rv = __moservices(__MO_GETSTOR, sizeof(mymopl),
                          &mymopl , &mymoptr);
if (mos_rv == 0) {
      printf("moservices(GETSTOR) successful, MO addr: %p\n",
             mymoptr);

      /* Free the 100MB memory object.                               */
      mos_rv = __moservices(__MO_DETACH, 0, NULL, &mymoptr);

      if (mos_rv != 0) {
        perror("moservices(DETACH) call failed");
      }
    }
    else {
      perror("moservices(GETSTOR) call failed");
    }

    return 0;
}
```

Related information

- "stdlib.h" on page 70

mount() — Make a file system available

Standards

Standards / Extensions	C or C++	Dependencies
z/OS UNIX	both	

Format

```
#define _OPEN_SYS

#include <sys/stat.h>

int mount(const char *path, char *filesystem,
          char *filesystype, mtm_t mtm,
          int parmlen, char *parm);
```

General description

Adds a file system to the hierarchical file system (HFS). The same file system cannot be mounted at more than one place in the hierarchical file system.

In order to mount a file system, the caller must be an authorized program, or must be running for a user with appropriate privileges.

path The mount point directory that the file system is to be mounted to.

filesystem
The name of the file system to be mounted; it must be unique within the system. For a hierarchical file system (HFS) data set, this is a 1-to-44-character MVS data set name specified as all uppercase letters.

This name is terminated with NULL characters.

filesystype
The name for the file system that will perform the mount. This 8-character name must match the TYPE operand on a FILESYSTYPE statement in the BPXPRMxx parmlib member for the file system.

mtm A flag field that specifies the mount mode and additional mount options:

MTM_RDONLY
Mount the file system as a read-only file system.

MTM_RDWR
Mount the file system as a read/write file system.

MTM_NOSUID
The SETUID and SETGID mode flags will be ignored for programs that reside in this file system.

MTM_SYNCHONLY
The mount must be completed synchronously or fail if it cannot.

parmlen
Length of the *parm* argument. The maximum length is 500 characters. For a hierarchical file system (HFS) data set, this is not specified.

parm A parameter passed to the physical file system that performs the mount. This parameter may not be required. The form and content of the *parm* are determined by the physical file system. A hierarchical file system (HFS) data set does *not* require a *parm*.

Returned value

If successful, mount() returns 0.

If the mount() is proceeding asynchronously, it returns 1.

If unsuccessful, mount() returns -1 and sets errno to one of the following values:

Error Code
Description

EBUSY
The specified file system is unavailable.

EINVAL
A parameter was incorrectly specified. Verify *filesystype* and *mtm*. Another possible reason for this error is that the mount point is the root of a file system or that the file system is already mounted.

EIO An I/O error occurred.

ELOOP
A loop exists in symbolic links. This error is issued if more than POSIX_SYMLOOP (defined in the limits.h header file) symbolic links are detected in the resolution of *pathname*.

ENOENT
The mount point does not exist.

ENOMEM
There is not enough storage available to save the information required for this file system.

ENOTDIR
The mount point is not a directory.

EPERM
Superuser authority is required to issue a mount.

Example

CELEBM21

```
/* CELEBM21

   This example adds a file system to the hierarchical
   file system.

 */
#define _OPEN_SYS
#include <sys/stat.h>
#include <stdio.h>
#include <unistd.h>

main() {
  char mount_point[]="/new_fs";
  char HFS[]="POSIX.NEW.HFS";
  char filesystype[9]="HFS     ";

  setvbuf(stdout, NULL, _IOLBF, 0);
  puts("before mount()");
  system("df -Pk");
  if (mount(mount_point, HFS, filesystype, MTM_RDWR, 0, NULL) != 0)
    perror("mount() error");
  else {
    puts("After mount()");
    system("df -Pk");
    if (umount(HFS, MTM_UMOUNT) != 0)
      perror("umount() error");
  }
}
```

Output

```
before mount()
Filesystem   1024-blocks        Used  Available  Capacity     Mounted on
POSIX.ROOT.FS       9600        8660        940       90%     /

After mount()
Filesystem   1024-blocks        Used  Available  Capacity     Mounted on
POSIX.NEW.HFS        200          20        180       10%     /new_fs
POSIX.ROOT.FS       9600        8660        940       90%     /
```

Related information

- “limits.h” on page 39
- “sys/stat.h” on page 75
- “umount() — Remove a virtual file system” on page 1949
- “w_getmntent() — Get information on mounted file systems” on page 2058
- “w_statfs() — Get the file system status” on page 2095

__mount() — Make a file system available

Standards

Standards / Extensions	C or C++	Dependencies
z/OS UNIX	both	OS/390 V2R9

Format

```
#define _OPEN_SYS
#include <sys/stat.h>
#include <sys/mntent.h>

int __mount(struct mnte2 *mnte, char *sysname);
```

General description

Adds a file system to the hierarchical file system. The same file system cannot be mounted at more than one place in the hierarchical file system.

In order to mount a file system, the caller must be an authorized program or must be running for a user with appropriate privileges.

Element descriptions

To mount or change the mount of an HFS file in a sysplex, the application should set values in *mnte* as follows:

mnt2h_cbid
: The mnte2 control block ID. Initialize it to "MNT2".

mnt2h_cblen
: The mnte2 size. Initialize it to the size of struct mnte2.

mh2_cursor
: Contains the internal cursor. This should be set to 0 initially, and must be left unchanged for subsequent calls.

mnth_devno
: This element contains the device number, if needed.

mh_bodylen
: The mnte2 size of w_mntent2. Must be initialized to the sizeof the w_mntent2 body.

rsvd This field must be set to all zeros.

mnt2_fstype
: The file system type.

mnt2_mode
: File system mount mode. A flag field that specifies the mount mode and additional mount options:

mntentfsaunmount
: If it is 1 after the file system is mounted, the file system will be unmounted when a system leaves the sysplex. If it is 0, then the setting of mntentfsnoautomove will be used. See mntentfsnoautomove below. This option can be changed after the file is mounted by changing this bit and setting the request bit,

mntentnewauto, to 1 before calling __mount(). If changed to 0, also set mntentfsnoautomove to indicate automove or no move.

mntentfsclient
If it is 0, then the file system is a sysplex client. If it is 1, then the file system is not a sysplex client.

Note: Note that mntentfsclient is not an input parameter.

mntentfsnoautomove
If it is 0 after the file system is mounted, it can be moved automatically. If it is 1 after the file system is mounted, it will not be moved automatically. The mode can be reversed after the file is mounted (when mntentfsaunmount is 0) by changing this bit and setting the request bit, mnte2ntnewauto, to 1 before calling __mount().

Note: The setting of this bit only applies if mntentfsaunmount is 0.

mntentfsmodenosec
If it is 1, then the file system will not enforce security checks. If it is 0, then the file system will enforce security checks.

mntentfsmodeexport
If it is 0, then the file system has not been exported by DFS. If it is 1, then the file system has been exported by DFS.

mntentfsmoderdonly
If it is 0,then the file system is mounted as read/write. If it is 1, then the file system is mounted as read-only.

mntentfsmodenosuid
If it is 1, then the SETUID and SETGID mode flags will be ignored for programs that reside in this file system. If it is 0 then the SETUID and SETGID mode flags will be enforced for programs that reside in this file system. This information is returned by the function but should not be changed.

mnt2_dev
Device # which stat will return for all files in this file system. Not set on input to __mount().

mnt2_parentdev
st_dev of parent file system. Not set on input to __mount().

mnt2_rootino
The ino of the mount point. Not set on input to __mount().

mnt2_status
status of the file system. The field is not an input parameter. It can be tested on output after a successful request.

mnt2_ddname
The ddname specified on mount. 1 to 8 characters are allowed.

mnt2_fstname
The name of the file system type specified by the FILESYS statement. The name for the file system that will perform the mount. This 8-character name must match the TYPE operand on a FILESYSTYPE statement in the BPXPRMxx parmlib member for the file system. 1 to 8 characters are allowed.

mnt2_fsname
The name of the file system to be mounted; it must be unique within the system. For a hierarchical file system (HFS) data set, this is a 1-to-44-character MVS data set name specified as all uppercase letters. This name is terminated with NULL characters.

mnt2_pathlen
The length of mount point path.

mnt2_mountpoint
The name of the directory where the file system is mounted. 1 to 1023 characters are allowed. Also refers to the mount point directory where the file system will be mounted.

mnt2_parmoffset
Offset of mount parameter *mnt2_parmreturn* from *mnt2_fstype*. Also refers to a parameter passed to the physical file system that performs the mount. This parameter may not be required. The form and content of the parameter are determined by the physical file system. A hierarchical file system (HFS) data set does not require a parameter.

mnt2_parmlen
The length of the mount parameter with size *mnt2_parmreturn*. Also refers to the length of the parameter argument. The maximum length is 1024 characters. A hierarchical file system data set does not require a parameter.

mnt2_sysname
The name of the target system. 1 to 8 characters are allowed. Changing the target system is always supplied as *sysname*. For all other calls, *sysname* must be supplied as NULL or the target name will be changed. When *sysname* is supplied, the mnte2ntchange flag must be set off for a mount function call, or the mnte2ntchange flag must be set on for a change mount function call. When you specify system on a mount it means mount this file on this system or when you specify system on a change mount it means move the file system from where it is currently mounted to this system.

mnt2_qsystem
The name of the quiesce system. 1 to 8 characters are allowed but the character(s) are padded with blanks and do not contain a NULL terminator. This field is an output only field from getmntent().

mnt2_fromsys
The name of the system from which the file system has moved.

mnt2_flags
The field containing the request flags. A flag field that specifies the change for existing mounted file system:

mnte2ntnewauto
This flag instigates a change of mode which will effect the automove state depending on the value that is set for mnte2ntfsnoautomove. See the explanation under mnte2ntfsnoautomove.

mnte2ntchange
The request in this w_mntent is a change to existing status or mode. This flag must be set on for all change mount requests. The w_mntent structure needs to modify either the mnte2ntfsname field or the mnte2ntmountpoint and mnte2ndpathlen fields. When the request is to mount a directory, then this flag must be set to off.

mnt2_status2
The file system status extensions.

mnt2_success
This field is used to return the number of successfully moved file systems when moving a collection of file systems. It is not used in other cases.

parm_point
This field contains the mountpoint parameters to be used when mounting a file system. It is a separate field in the *mnte2* structure but contiguiously allocated following the *w_mnte2* body. The *mnt2_parmoffset* field contains the offset to the start of *parm_point*.

mnt2_syslistlength
Length of system list.

mnt2_syslistoffset
Offset of system list.

mnt2_aggnamelength
Length of the aggregate name in mnt2_aggname. The length does not include the NULL terminating character, and is only valid if mnt2_aggnameoffset has a non-zero value.

mnt2_aggnameoffset
The offset of mnt2_aggname from w_mntent. If the value is zero, then no aggregate name is returned.

The *mnte3* structure can also be used in place of the *mnte2* structure. The following *mnte3* fields are equivalent to their *mnte2* counterparts, with appropriate changes to the header fields to specify the usage of the *mnte3* structure:

mnt3h_cbid
The mnte3 control block ID. Initialize it to "MNT3".

mnt3h_cblen
The mnte3 size. Initialize to the size of struct mnte3.

mh3_cursor

mh3_devno

mh_bodylen
The mnte3 size of w_mntent3. Must be initialized to the sizeof the w_mntent3 body.

rsvd

mnt3_fstype

mnt3_mode

mnt3_dev

mnt3_parentdev

mnt3_rootino

mnt3_status

mnt3_ddname

mnt3_fstname

mnt3_fsname

mnt3_pathlen

mnt3_mountpoint

mnt3_parmoffset

mnt3_parmlen

mnt3_sysname

mnt3_qsysname

mnt3_fromsys

mnt3_flags

mnt3_status2

mnt3_success

Returned value

If successful, __mount() returns 0.

If the __mount() is proceeding asynchronously, it returns 1.

If unsuccessful, __mount() returns -1 and sets errno to one of the following values:

Error Code
Description

EBUSY
The specified file system is unavailable.

EINVAL
A parameter was incorrectly specified. Verify *filesystype* and *mtm*. Another possible reason for this error is that the mount point is the root of a file system or that the file system is already mounted.

EIO
An I/O error occurred.

ELOOP
A loop exists in symbolic links. This error is issued if more than POSIX_SYMLOOP (defined in the limits.h header file) symbolic links are detected in the resolution of *pathname*.

ENOENT
The mount point does not exist.

ENOMEM
There is not enough storage available to save the information required for this file system.

ENOTDIR
The mount point is not a directory.

EPERM
Appropriate authority is required to issue a mount.

Related information

- “limits.h” on page 39
- “sys/mntent.h” on page 73
- “sys/stat.h” on page 75
- “umount() — Remove a virtual file system” on page 1949
- “w_getmntent() — Get information on mounted file systems” on page 2058
- “w_statfs() — Get the file system status” on page 2095

mprotect() — Set protection of memory mapping

Standards

Standards / Extensions	C or C++	Dependencies
XPG4.2 Single UNIX Specification, Version 3 z/OS UNIX	both	

Format

```
#define _XOPEN_SOURCE_EXTENDED 1
#include <sys/mman.h>

int mprotect(void *addr, size_t len, int prot);
```

General description

The mprotect() function changes the access protections on the mappings specified by the *len* up to the next multiple of the page size as returned by sysconf(), to be that specified by *prot*. Legitimate values for *prot* are the same as those permitted for mprotect() and are defined in <sys/mman.h.>:

PROT_READ
page can be read

PROT_WRITE
page can be written

PROT_EXEC
page can be executed

PROT_NONE
page cannot be accessed

The range provided by the Map_address and Map_length may span regular maps as well as __MAP_MEGA maps. Mprotect affects __MAP_MEGA maps very differently than regular maps. The difference is in the scope of the change. When a change is made to a __MAP_MEGA map, the change affects all processes which are currently mapped to the same file-offset range represented by the pages within the provided range. For example, changing a file-offset range (storage pages) that is currently in use with a protection of write to a protection of read, makes the file-offset range read for all processes, not just the current one. In other words, the changes are global. On the other hand, changes to regular maps affect only the process that issues mprotect.

When mprotect() fails for reasons other than EINVAL, the protection on some of the pages in the range [*addr, addr + len*) may have been changed.

Returned value

If successful, mprotect() returns 0.

If unsuccessful, mprotect() returns -1 and sets errno to one of the following values:

Error Code
Description

EACCES
The *prot* argument specifies a protection that violates the access permission the process has to the underlying memory object.

EAGAIN
The *prot* argument specifies PROT_WRITE over a MAP_PRIVATE mapping and there are insufficient memory resources to reserve for locking the private page.

EINVAL
The *addr* argument is not a multiple of the page size as returned by sysconf().

ENOMEM
Addresses in the range [*addr, addr + len*) are invalid for the address space of a process, or specify one or more pages

Related information

- "sys/mman.h" on page 73
- "mmap() — Map pages of memory" on page 1081
- "sysconf() — Determine system configuration options" on page 1798

mrand48() — Pseudo-random number generator

Standards

Standards / Extensions	C or C++	Dependencies
XPG4 XPG4.2 Single UNIX Specification, Version 3	both	

Format

```
#define _XOPEN_SOURCE
#include <stdlib.h>

long int mrand48(void);
```

General description

The drand48(), erand48(), jrand48(), lrand48(), mrand48() and nrand48() functions generate uniformly distributed pseudo-random numbers using a linear congruential algorithm and 48-bit integer arithmetic.

The functions drand48() and erand48() return nonnegative, double-precision, floating-point values, uniformly distributed over the interval [0.0,1.0).

The functions lrand48() and nrand48() return nonnegative, long integers, uniformly distributed over the interval [0,2**31).

The functions mrand48() and jrand48() return signed long integers, uniformly distributed over the interval [-2**31,2**31).

The mrand48() function generates the next 48-bit integer value in a sequence of 48-bit integer values, X(i), according to the linear congruential formula:

```
X(n+1) = (aX(n) + c)mod(2**48)     n>=0
```

The initial values of X, a, and c are:

```
X(0) = 1
a   = 5deece66d (base 16)
c   = b         (base 16)
```

C/370 provides storage to save the most recent 48-bit integer value of the sequence, X(i). This storage is shared by the drand48(), lrand48() and mrand48() functions. The value, X(n), in this storage may be reinitialized by calling the lcong48(), seed48() or srand48() function. Likewise, the values of a and c, may be changed by calling the lcong48() function. Thereafter, whenever the seed48() or srand48() function is called to change X(n), the initial values of a and c are also reestablished.

Special behavior for z/OS UNIX services

You can make the mrand48() function and other functions in the drand48 family thread-specific by setting the environment variable _RAND48 to the value THREAD before calling any function in the drand48 family.

If you do not request thread-specific behavior for the drand48 family, C/370 serializes access to the storage for X(n), a and c by functions in the drand48 family when they are called by a multithreaded application.

If thread-specific behavior is requested, and the mrand48() function is called from thread t, the mrand48() function generates the next 48-bit integer value in a sequence of 48-bit integer values, X(t,i), for the thread t. The sequence of values for a thread is generated according to the linear congruential formula:

```
X(t,n+1) = (a(t)X(t,n) + c(t))mod(2**48)     n>=0
```

The initial values of X(t), a(t) and c(t) for the thread t are:

```
X(t,0) = 1
a(t)   = 5deece66d (base 16)
c(t)   = b         (base 16)
```

C/370 provides storage which is specific to the thread t to save the most recent 48-bit integer value of the sequence, X(t,i), generated by the drand48(), lrand48() or mrand48() function. The value, X(t,n), in this storage may be reinitialized by calling the lcong48(), seed48() or srand48() function from the thread t. Likewise, the values of a(t) and c(t) for thread t may be changed by calling the lcong48() function from the thread. Thereafter, whenever the seed48() or srand48() function is called from the thread t to change X(t,n), the initial values of a(t) and c(t) are also reestablished.

Returned value

mrand48() transforms the generated 48-bit value, X(n+1), to a signed long integer value on the interval [-2**31,2**31) and returns this transformed value.

Special behavior for z/OS UNIX Services

If thread-specific behavior is requested for the drand48 family and the mrand48() function is called on thread t, the mrand48() function transforms the generated 48-bit value, X(t,n+1), to a signed long integer value on the interval [-2**31,2**31) and returns this transformed value.

Related information

- "stdlib.h" on page 70
- "drand48() — Pseudo-random number generator" on page 402
- "erand48() — Pseudo-random number generator" on page 426
- "jrand48() — Pseudo-random number generator" on page 928
- "lcong48() — Pseudo-random number initializer" on page 939
- "lrand48() — Pseudo-random number generator" on page 1010
- "nrand48() — Pseudo-random number generator" on page 1148
- "seed48() — Pseudo-random number initializer" on page 1467
- "srand48() — Pseudo-random number initializer" on page 1718

m_setvalues_layout() — Set layout values of a layout object (bidi data)

Standards

Standards / Extensions	C or C++	Dependencies
ISO C XPG4 C99	both	z/OS V1R2

Format

```
#include <sys/layout.h>

int m_setvalues_layout(LayoutObject layout_object, const LayoutValues values,
                       int *index_returned);
```

General description

The m_setvalues_layout() function is part of the support for handling of bidirectional (Bidi) conversion of data between Visual (MVS) and Implicit (z/OS UNIX) formats. Initial support is for Arabic and Hebrew data.

The m_setvalues_layout() function is used to change the layout values of a LayoutObject. The *layout_object* argument specifies a LayoutObject returned by the m_create_layout() function. The *values* argument specifies the list of layout values which are to be changed. The values are written into the LayoutObject and may affect the behavior of subsequent layout functions.

Note: Some layout values do alter internal states maintained by a LayoutObject. The m_setvalues_layout() function can be implemented as a macro that evaluates the first argument twice.

Returned value

If successful, m_setvalues_layout() sets the requested layout values and returns 0.

If any value cannot be set, m_setvalues_layout() does not change any of the layout values. It stores into *index_returned* the (zero-based) index of the value causing the error. It returns -1 and sets errno to one of the following values:

Error Code
: **Description**

EINVAL
: The layout value specified by index_returned is unknown or its value is invalid or the argument layout_object is invalid.

Related information

- "sys/layout.h" on page 73
- "m_create_layout() — Create and initialize a layout object (bidi data)" on page 1058
- "m_destroy_layout() — Destroy a layout object (bidi data)" on page 1059
- "m_getvalues_layout() — Query layout values of a layout object (bidi data)" on page 1067
- "m_transform_layout() — Layout transformation for character strings (bidi data)" on page 1118
- "m_wtransform_layout() — Layout transformation for wide-character strings (bidi data)" on page 1125

msgctl() — Message control operations

Standards

Standards / Extensions	C or C++	Dependencies
XPG4 XPG4.2 Single UNIX Specification, Version 3	both	

Format

```
#define _XOPEN_SOURCE
#include <sys/msg.h>

int  msgctl(int msgid, int cmd, struct msqid_ds *buf);
```

General description

The msgctl() function provides message control operations as specified by *cmd*. The following values for *cmd*, and the message control operations they specify, are (These symbolic constants are defined by the <sys/ipc.h> header):

IPC_STAT
Place the current value of each member of the `msqid_ds` data structure associated with *msgid* into the structure pointed to by *buf*. The contents of this structure are defined in <sys/msg.h>. This command requires read permission.

IPC_SET
Set the value of the following members of the `msqid_ds` data structure associated with *msgid* to the corresponding value found in the structure pointed to by *buf*:

- msg_perm.uid
- msg_perm.gid
- msg_perm.mode
- msg_qbytes

IPC_SET can only be executed by a process with the appropriate privileges or that has an effective user ID equal to the value of `msg_perm.cuid` or `msg_perm.uid` in the `msqid_ds` data structure associated with *msgid*. Only a process with appropriate privileges can raise the value of `msg_gbytes`.

IPC_RMID
Remove the message queue identifier specified by *msgid* from the system and destroy the message queue and `msqid_ds` data structure associated with it. **IPC_RMID** can only be executed by a process with appropriate

privileges or one that has an effective user ID equal to the value of `msg_perm.cuid` or `msg_perm.uid` in the `msqid_ds` data structure associated with *msgid*.

Returned value

If successful, msgctl() returns 0.

If unsuccessful, msgctl() returns -1 and sets errno to one of the following values:

Error Code
: **Description**

EACCES
: The argument *cmd* is **IPC_STAT** and the calling process does not have read permission.

EINVAL
: The value of *msgid* is not a valid message queue identifier, or the value of *cmd* is not a valid command.

EPERM
: The argument *cmd* is **IPC_RMID** or **IPC_SET** and the effective user ID of the calling process is not equal to that of a process with appropriate privileges and it is not equal to the value of `msg_perm.cuid` or `msg_perm.uid` in the data structure associated with *msgid*.

  Or the argument *cmd* is **IPC_SET**, an attempt is being made to increase the value of `msg_qbytes`, and the effective user ID of the calling process does not have appropriate privileges.

Related information

- “sys/ipc.h” on page 73
- “sys/msg.h” on page 74
- “msgget() — Get message queue”
- “msgrcv() — Message receive operation” on page 1108
- “msgsnd() — Message send operations” on page 1113
- “msgxrcv() — Extended message receive operation” on page 1115

msgget() — Get message queue

Standards

Standards / Extensions	C or C++	Dependencies
XPG4 XPG4.2 Single UNIX Specification, Version 3	both	

Format

```
#define _XOPEN_SOURCE
#include <sys/msg.h>

int msgget(key_t key, int msgflg);
```

General description

The msgget() function returns the message queue identifier associated with the argument *key*.

A message queue identifier, associated message queue and data structure (see <sys/msg.h>) are created for the argument *key* if one of the following is true:

- The argument *key* is equal to **IPC_PRIVATE**
- The argument *key* does not already have a message queue identifier associated with it, and the flag **IPC_CREAT** is on in *msgflg*.

Valid values for the argument *msgflg* include any combination of the following constants defined in <sys/ipc.h> and <sys/modes.h>:

IPC_CREAT
Create a message queue if the *key* specified does not already have an associated ID. **IPC_CREAT** is ignored when **IPC_PRIVATE** is specified

IPC_EXCL
Causes the msgget() function to fail if the *key* specified has an associated ID. **IPC_EXCL** is ignored when **IPC_CREAT** is not specified or **IPC_PRIVATE** is specified

IPC_RCVTYPEPID
Creates a message queue that can only be read from msgrcv() when Message_Type is the process ID of the invoker. This restriction does not apply if the msgrcv() invoker has the same effective UID as the message queue creator.

IPC_SNDTYPEPID
Creates a message queue that can only be written to msgsnd() when MSG_TYPE is the process ID of the invoker. This restriction does not apply if the msgsnd() invoker has the same effective UID as the message queue creator.

S_IRUSR
Permits read access when the effective user ID of the caller matches either `msg_perm.cuid` or `msg_perm.uid`

S_IWUSR
Permits write access when the effective user ID of the caller matches either `msg_perm.cuid` or `msg_perm.uid`

S_IRGRP
Permits read access when the effective group ID of the caller matches either `msg_perm.cgid` or `msg_perm.gid`

S_IWGRP
Permits write access when the effective group ID of the caller matches either `msg_perm.cgid` or `msg_perm.gid`

S_IROTH
Permits other read access

S_IWOTH
Permits other write access

When a message set associated with argument *key* already exists, setting **IPC_EXCL** and **IPC_CREAT** in argument *msgflg* will force msgget() to fail.

Upon creation, the `msg_ds` data structure associated with the new message queue identifier is initialized as follows:

- The fields `msg_perm.cuid`, `msg_perm.uid`, `msg_perm.cgid`, and `msg_perm.gid` are set equal to the effective user ID and effective group ID, respectively, of the calling process.

- The low-order 9 bits of `msg_perm.mode` are set equal to the low-order 9 bits of *msgflg*.
- The fields `msg_qnum`, `msg_lspid`, `msg_lrpid`, msg_stime, and `msg_rtime` are set to zero.
- The field `msg_ctime` is set equal to the current time.
- The field `msg_qbytes` is set equal to the system limit.

Usage notes

1. In a client/server environment, two message queues can be used. One inbound to the server created with IPC_SNDTYPEPID and the other outbound from the server created with IPC_RCVTYPEPID. This arrangement guarantees that the server knows the process ID of the client and the client is the only process that receives the server's returned message. The server may invoke msgrcv() with PID=0 to see if any messages belong to process IDs that have gone away.
2. Important terms and their descriptions are explained:

   Term **Descriptions**

   PLO Perform Lock Operation.

   IPC_PLO1
   Use PLO serialization (if available) until a select() involving this message queue is detected.

   IPC_PLO2
   Allow the kernel to use its best judgment with serialization (IPC_PLO1 ignored).

   - Message_Flags IPC_PLO1 and IPC_PLO2 are ignored if the PLO instruction is not present on the hardware.
   - Performance of the PLO instruction for serialization will vary with the msgrcv type, number of messages on the queue and the number of tasks doing msgsnd() and msgrcv(). Msgrcv() with type<0 and long message queues is expected to be a worse performer. Msgrcv() with type>0 is expected to be an equivalent or good performer. Msgrcv() with type=0 is expected to be a very good performer.
   - Message queues created with *Ipc_RcvTypePID*, *Ipc_SndTypePID*, *IPC_PLO1* and *IPC_PLO2* will show these bits and may show the *IPC_PLOINUSE* bit in the S_MODE byte returned with *w_getipc*.
   - Message queue PLO serialization is not compatible with select() using message queues. When msgrcv() detects a select() for a message queue, serialization will be changed to use traditional latches.
   - Performance runs should be made with *IPC_PLO1* since *IPC_PLO2* may switch to latch serialization and the user would not know when.

Returned value

If successful, msgget() returns a nonnegative integer, namely a message queue identifier.

If unsuccessful, msgget() returns -1 and sets errno to one of the following values:

Error Code
Description

EACCES
A message queue identifier exists for the argument *key*, but access permission as specified by the low-order 9 bits of *msgflg* could not be granted

EEXIST
A message queue identifier exists for the argument *key*, but both **IPC_CREAT** and **IPC_EXCL** are specified in *msgflg*

EINVAL
The value of argument *msgflg* is not currently supported

ENOENT
A message queue identifier does not exist for the argument *key* and **IPC_CREAT** is not specified.

ENOSPC
A message queue identifier is to be created but the system-imposed limit on the maximum number of allowed message queue identifiers system-wide would be exceeded.

When *msgflg* equals 0, the following applies:

- If a message queue identifier has already been created with *key* earlier, and the calling process of this msgget() has read and/or write permissions to it, then msgget() returns the associated message queue identifier.
- If a message queue identifier has already been created with *key* earlier, and the calling process of this msgget() does not have read and/or write permissions to it, then msgget() returns-1 and sets errno to EACCES.
- If a message queue identifier has not been created with *key* earlier, then msgget() returns -1 and sets errno to ENOENT.

Related information

- "sys/ipc.h" on page 73
- "sys/msg.h" on page 74
- "sys/types.h" on page 75
- "ftok() — Generate an interprocess communication (IPC) key" on page 662
- "msgctl() — Message control operations" on page 1104
- "msgrcv() — Message receive operation"
- "msgsnd() — Message send operations" on page 1113
- "msgxrcv() — Extended message receive operation" on page 1115

msgrcv() — Message receive operation

Standards

Standards / Extensions	C or C++	Dependencies
XPG4 XPG4.2 Single UNIX Specification, Version 2 Single UNIX Specification, Version 3	both	

Format

Non-Single UNIX Specification, Version 2

```
#define _XOPEN_SOURCE
#include <sys/msg.h>

int msgrcv(int msgid, void *msgp, size_t msgsz, long int msgtyp, int msgflg);
```

Single UNIX Specification, Version 2

```
#define _XOPEN_SOURCE 500
#include <sys/msg.h>

ssize_t msgrcv(int msgid, void *msgp, size_t msgsz, long int msgtyp, int msgflg);
```

General description

The msgrcv() function reads a message from the queue associated with the message queue identifier specified by *msgid* and places it in the user-defined buffer pointed to by *msgp*.

The argument *msgp* points to a user-defined buffer that must contain first a field of type long int that will specify the type of the message, and then a data portion that will hold the data bytes of the message. The structure below is an example of what this user-defined buffer should look like:

```
struct message
{
     long int   mtype;    Message type
     int        mtext[n]; Message text
}
```

The structure member, `mtype`, is the received message's type as specified by the sending process. The structure member, `mtext`, is the text of the message.

The argument *msgsz* specifies the size in bytes of `mtext`, The received message is truncated to *msgsz* bytes if it is larger than *msgsz* and the **MSG_NOERROR** flag was specified in the argument *msgflg*. The truncated portion of the message is lost and no indication of the truncation is given to the calling process.

The argument *msgtyp* specifies the type of message requested, as follows:
- If *msgtyp* is equal to zero, the first message on the queue is received.
- If *msgtyp* is greater than 0, the first message of type, *msgtyp*, is received.
- If *msgtyp* is less than 0, the first message of the lowest type that is less than or equal to the absolute value of *msgtyp* is received.

The argument *msgflg* specifies the action to be taken if a message of the desired type is not on the queue. These are as follows:
- If the **IPC_NOWAIT** flag is on in *msgflg*, the calling process will return immediately with a return value of -1 and errno set to **ENOMSG**.
- If the **IPC_NOWAIT** flag is off in *msgflg* the calling process will suspend execution until one of the following occurs:
  - A message of the desired type is placed on the queue.
  - The message queue identifier, *msgid*, is removed from the system; when this occurs, errno is set to EIDRM and a value of -1 is returned.
  - The calling process receives a signal that is to be caught; in this case a message is not received and the calling process resumes execution. A value of -1 is returned and errno is set to EINTR.

If successful, the following actions are taken with respect to the data structure, `msgiq_ds`, associated with *msgid*:

1. `msg_qnum` is decremented by 1.
2. `msg_lrpid` is set equal to the process ID of the calling process.
3. `msg_rtime` is set equal to the current time.

Returned value

If successful, msgrcv() returns a value equal to the number of bytes actually placed into the `mtext` field of the user-defined buffer pointed to by *msgp*. A value of zero indicates that only the `mtype` field was received from the message queue.

If unsuccessful, msgrcv() returns -1 and sets errno to one of the following values:

Error Code
Description

E2BIG The value of `mtext` is greater than *msgsz* and the flag **MSG_NOERROR** was not specified.

EACCES
The calling process does not have read permission to the message queue associated with the message queue identifier *msgid* or the message queue was built with IPC_RCVTYPEPID and the Message_Type was other than the invoker's process ID (JRTypeNotPID).

EIDRM
The message queue identifier, *msgid*, has been removed from the system while the caller of msgrcv() was waiting.

EINTR
The function msgrcv() was interrupted by a signal before a message could be received.

EINVAL
The value of argument *msgid* is not a valid message queue identifier or the value of *msgsz* is less than zero.

ENOMSG
The flag **IPC_NOWAIT** was specified and the message queue does not contain a message of the desired type.

Related information

- “sys/ipc.h” on page 73
- “sys/msg.h” on page 74
- “msgctl() — Message control operations” on page 1104
- “msgget() — Get message queue” on page 1105
- “msgsnd() — Message send operations” on page 1113
- “msgxrcv() — Extended message receive operation” on page 1115

__msgrcv_timed() — Message receive operation with timeout

Standards

Standards / Extensions	C or C++	Dependencies
z/OS UNIX	both	OS/390 V2R10

Format

```
#define _OPEN_SYS_TIMED_EXT 1
#include <time.h>
#include <sys/msg.h>

int __msgrcv_timed(int msgid, void *msgp, size_t msgsz,
                   long int msgtyp, int msgflg, struct timespec *set);
```

General description

Reads a message from the queue associated with the message queue identifier specified by *msgid* and places it in the user-defined buffer pointed to by *msgp*.

The argument *msgp* points to a user-defined buffer that must contain first a field of type long int that will specify the type of the message, and then a data portion that will hold the data bytes of the message. The structure below is an example of what this user-defined buffer should look like:

```
struct message {
   long int   mtype;    Message type
   int        mtext[n]; Message text
 }
```

The structure member, `mtype`, is the received message's type as specified by the sending process. The structure member, `mtext`, is the text of the message.

The argument *msgsz* specifies the size in bytes of `mtext`. The received message is truncated to *msgsz* bytes if it is larger than *msgsz* and the **MSG_NOERROR** flag was specified in the argument *msgflg*. The truncated portion of the message is lost and no indication of the truncation is given to the calling process.

The argument *msgtyp* specifies the type of message requested, as follows:

- If *msgtyp* is equal to zero, the first message on the queue is received.
- If *msgtyp* is greater than 0, the first message of type, *msgtyp*, is received.
- If *msgtyp* is less than 0, the first message of the lowest type that is less than or equal to the absolute value of *msgtyp* is received.

The argument *msgflg* specifies the action to be taken if a message of the desired type is not on the queue. These are as follows:

- If the **IPC_NOWAIT** flag is on in *msgflg*, the calling process will return immediately with a return value of -1 and errno set to ENOMSG.
- If the **IPC_NOWAIT** flag is off in *msgflg* the calling process will suspend execution until one of the following occurs:
  - A message of the desired type is placed on the queue.
  - The message queue identifier, *msgid*, is removed from the system; when this occurs, errno is set to EIDRM and a value of -1 is returned.
  - The calling process receives a signal that is to be caught; in this case a message is not received and the calling process resumes execution. A value of -1 is returned and errno is set to EINTR.

The argument *set* is the timespec structure which contains the timeout value.

If successful, the following actions are taken with respect to the data structure, `msqid_ds`, associated with *msgid*:

1. `msg_qnum` is decremented by 1.

2. `msg_lrpid` is set equal to the process ID of the calling process.
3. `msg_rtime` is set equal to the current time.

The variable *set* gives the timeout specification.

- If the __msgrcv_timed() function finds that none of the messages specified by *msgid* are received, it waits for the time interval specified in the **timespec** structure referenced by *set*. If the **timespec** structure pointed to by *set* is zero-valued and if none of the messages specified by *msgid* are received, then __msgrcv_timed() returns immediately with EAGAIN. A **timespec** with the tv_sec field set with *INT_MAX*, as defined in <limits.h>, will cause the __msgrcv_timed() service to wait until a message is received. If *set* is the NULL pointer, it will be treated the same as when **timespec** structure was supplied with the tv_sec field set with *INT_MAX*.

Returned value

If successful, __msgrcv_timed() returns a value equal to the number of bytes actually placed into the `mtext` field of the user-defined buffer pointed to by *msgp*. A value of zero indicates that only the `mtype` field was received from the message queue.

If unsuccessful, __msgrcv_timed() returns -1 and sets errno to one of the following values:

Error Code
Description

E2BIG The value of `mtext` is greater than *msgsz* and the flag **MSG_NOERROR** was not specified.

EACCES
The calling process does not have read permission to the message queue associated with the message queue identifier *msgid*.

EAGAIN
The operation would result in time requested expired before any messages were received. This would result if the timeout specified expires before a message is posted.

EIDRM
The message queue identifier, *msgid*, has been removed from the system while the caller of __msgrcv_timed() was waiting.

EINTR
The function __msgrcv_timed() was interrupted by a signal before a message could be received.

EINVAL
The value of argument *msgid* is not a valid message queue identifier or the value of *msgsz* is less than zero.

ENOMSG
The flag **IPC_NOWAIT** was specified and the message queue does not contain a message of the desired type.

Related information

- “time.h” on page 79
- “sys/msg.h” on page 74

msgsnd() — Message send operations

Standards

Standards / Extensions	C or C++	Dependencies
XPG4 XPG4.2 Single UNIX Specification, Version 3	both	

Format

```
#define _XOPEN_SOURCE
#include <sys/msg.h>

int msgsnd(int msgid, const void *msgp, size_t msgsz, int msgflg);
```

General description

The msgsnd() function is used to send a message to the queue associated with the message queue identifier specified by *msgid*.

The argument *msgp* points to a user-defined buffer that must contain first a field of type long int that will specify the type of the message, and then a data portion that will hold the data bytes of the message. The structure below is an example of what this user-defined buffer should look like:

```
struct message
{
    long int   mtype;    Message type
    int        mtext[n];  Message text
}
```

The structure member, mtype, must be a nonzero positive value that can be used by the receiving process for message selection. The structure member, mtext, is any text of length, *msgsz*, bytes.

The argument *msgsz* specifies the size in bytes of mtext. When only mtype is to be sent with no mtext, msgsz is set to zero. The argument can range from zero to a system-imposed maximum or the maximum number of bytes allowed in the message queue.

The argument *msgflg* specifies the action to be taken if one or more of the following are true:

- Placing the message on the message queue would cause the current number of bytes on the message queue (msg_cbytes) to exceed the maximum number of bytes allowed on this queue, as specified in msg_qbytes.
- The total number of messages on the queue is equal to the system-imposed limit.

These actions are as follows:

- If the **IPC_NOWAIT** flag is on in msgflg, the message will not be sent and the calling process will return immediately. msgsnd() will return -1 and set errno to EAGAIN.
- If the **IPC_NOWAIT** flag is off in *msgflg*, the calling process will suspend execution until one of the following occurs:
  1. The condition responsible for the suspension no longer exists, in which case the message is sent.

2. The message queue identifier, *msgid*, is removed from the system; when this occurs, errno is set to EIDRM and a value of -1 is returned.
3. The calling process receives a signal that is to be caught; in this case a message is not sent and the calling process resumes execution. A value of -1 is returned and error is set to EINTR.

If successful, the following actions are taken with respect to the data structure, `msqid_ds`, associated with *msgid*:

1. msg_qnum is incremented by 1.
2. msg_lspid is set equal to the process ID of the calling process.
3. msg_stime is set equal to the current time.

Returned value

If successful, msgsnd() returns 0.

If unsuccessful, no message is sent, msgsnd() returns -1, and sets errno to one of the following values:

Error Code
Description

EACCES
The calling process does not have write permission to the message queue associated with the message queue identifier *msgid* or the message queue was built with IPC_SNDTYPEPID and the MSG_TYPE was other than the invoker's process ID (JRTypeNotPID).

EAGAIN
The message cannot be sent for one of the reasons cited above and IPC_NOWAIT was specified.

EIDRM
The message queue identifier, *msgid*, has been removed from the system while the caller of msgsnd() was waiting.

EINTR
The function msgsnd() was interrupted by a signal before a message could be sent.

EINVAL
The value of argument *msgid* is not a valid message queue identifier, or the value of mtype is less than 1; or the value of msgsz is less than zero or greater than the system-imposed limit.

ENOMEM
Not enough system storage exists to complete the msgsnd() function.

Related information

- “sys/ipc.h” on page 73
- “sys/msg.h” on page 74
- “msgctl() — Message control operations” on page 1104
- “msgget() — Get message queue” on page 1105
- “msgrcv() — Message receive operation” on page 1108
- “msgxrcv() — Extended message receive operation” on page 1115

msgxrcv() — Extended message receive operation

Standards

Standards / Extensions	C or C++	Dependencies
z/OS UNIX	both	

Format

```
#define _OPEN_SYS_IPC_EXTENSIONS
#include <sys/msg.h>

int msgxrcv(int msgid, void *msgp, size_t msgsz, long int msgtyp, int msgflg);
```

Note: To expose the msgxrcv() name, the feature test macro **_OPEN_SYS_IPC_EXTENSIONS** should be defined. Otherwise, the function's name is __msgxrcv().

General description

The msgxrcv() function reads an extended message from the queue associated with the message queue identifier specified by *msgid* and places it in the user-defined buffer pointed to by *msgp*.

The argument *msgp* points to a user-defined buffer where the extended message will be received. This buffer must be defined by a data structure of the following format: .

```
struct msgxbuf {
    time_t     mtime;    Time and date message was sent
    uid_t      muid;     Sender's effective user ID
    gid_t      mgid;     Sender's effective group ID
    pid_t      mpid;     Sender's process ID
    long int   mtype;    Message type
    int        mtext[n]; Message text
}
```

The structure member, `mtype`, is the received message's type as specified by the sending process. The structure member, `mtext`, is the text of the message.

The argument *msgsz* specifies the size in bytes of `mtext`, The received message is truncated to *msgsz* bytes if it is larger than *msgsz* and the **MSG_NOERROR** flag was specified in the argument *msgflg*. The truncated portion of the message is lost and no indication of the truncation is given to the calling process.

The argument *msgtyp* specifies the type of message requested, as follows:

- If *msgtyp* is equal to zero, the first message on the queue is received.
- If *msgtyp* is greater than 0, the first message of type, *msgtyp*, is received.
- If *msgtyp* is less than 0, the first message of the lowest type that is less than or equal to the absolute value of *msgtyp* is received.

The argument *msgflg* specifies the action to be taken if a message of the desired type is not on the queue. These are as follows:

- If the **IPC_NOWAIT** flag is on in *msgflg*, the calling process will return immediately with a return value of -1 and errno set to **ENOMSG**.
- If the **IPC_NOWAIT** flag is off in *msgflg* the calling process will suspend execution until one of the following occurs:

- A message of the desired type is placed on the queue.
- The message queue identifier, *msgid*, is removed from the system; when this occurs, errno is set to EIDRM and a value of -1 is returned.
- The calling process receives a signal that is to be caught; in this case a message is not received and the calling process resumes execution. A value of -1 is returned and errno is set to EINTR.

If successful, the following actions are taken with respect to the data structure, `msqid_ds`, associated with *msgid*:

1. `msg_qnum` is decremented by 1.
2. `msg_lrpid` is set equal to the process ID of the calling process.
3. `msg_rtime` is set equal to the current time.

Returned value

If successful, msgxrcv() returns a value equal to the number of bytes actually placed into the `mtext` field of the user-defined buffer pointed to by *msgp*. A value of zero indicates that only the `mtype` field was received from the message queue.

If unsuccessful, msgxrcv() returns -1 and sets errno to one of the following values:

Error Code
Description

E2BIG The value of `mtext` is greater than *msgsz* and the flag **MSG_NOERROR** was not specified.

EACCES
The calling process does not have read permission to the message queue associated with the message queue identifier *msgid*.

EIDRM
The message queue identifier, *msgid*, has been removed from the system while the caller of msgxrcv() was waiting.

EINTR
The function msgxrcv() was interrupted by a signal before a message could be received.

EINVAL
The value of argument *msgid* is not a valid message queue identifier or the value of *msgsz* is less than zero.

ENOMSG
The flag **IPC_NOWAIT** was specified and the message queue does not contain a message of the desired type.

Related information

- “sys/ipc.h” on page 73
- “sys/msg.h” on page 74
- “msgctl() — Message control operations” on page 1104
- “msgget() — Get message queue” on page 1105
- “msgrcv() — Message receive operation” on page 1108
- “msgsnd() — Message send operations” on page 1113

msync() — Synchronize memory with physical storage

Standards

Standards / Extensions	C or C++	Dependencies
XPG4.2 Single UNIX Specification, Version 3	both	

Format

```
#define _XOPEN_SOURCE_EXTENDED 1
#include <sys/mman.h>

int msync(void *addr, size_t len, int flags);
```

General description

The msync() function writes all modified copies of pages over the range [*addr, addr* + *len*) to the underlying hardware, or invalidates any copies so that further references to the pages will be obtained by the system from their permanent storage locations.

The *flags* argument is:

MS_ASYNC
Perform asynchronous writes

MS_INVALIDATE
Invalidate mappings

MS_SYNC
Perform synchronous writes

The function synchronizes the file contents to match the current contents to the memory region.
- All write references to the memory region made before the call are visible by subsequent read operations on the file.
- It is unspecified whether writes to the same portion of the file before the call are visible by read references to the memory region.
- It is unspecified whether unmodified pages in the specified range are also written to the underlying hardware.

If *flags* is MS_ASYNC, the function may return immediately once all write operations are schedule; if *flags* is MS_SYNC, the function does not return until all write operations are completed.

MS_INVALIDATE synchronizes the contents of the memory region to match the current file contents.
- All writes to the mapped portion of the file made before the call are visible by subsequent read references to the mapped memory region.
- It is unspecified whether write references before the call, by any process, to memory regions mapped to the same portion of the file using MAP_SHARED, are visible by read references to the region.

If msync() causes any write to the file, then the file's st_ctime and st_mtime fields are marked for update.

Returned value

If successful, msync() returns 0.

If unsuccessful, msync() returns -1 and sets errno to one of the following values:

Error Code
Description

EINVAL
The *addr* argument is not a multiple of the page size as returned by *sysnonf*.

EIO An I/O error occurred while reading from or writing to the file system.

ENOMEM
Some or all the addresses in the range [*addr, addr* + range [*addr, addr* + *len*) are *len*) are invalid for the address space of the process or pages not mapped are specified.

Related information

- "sys/mman.h" on page 73
- "mmap() — Map pages of memory" on page 1081
- "sysconf() — Determine system configuration options" on page 1798

m_transform_layout() — Layout transformation for character strings (bidi data)

Standards

Standards / Extensions	C or C++	Dependencies
ISO C XPG4 C99	both	z/OS V1R2

Format

```
#include <sys/layout.h>

int m_transform_layout(LayoutObject layout_object,
                       const char *InpBuf,
                       const size_t InpSize,
                       void *OutBuf,
                       size_t *Outsize,
                       size_t *InpToOut,
                       size_t *OutToInp,
                       unsigned char *Property,
                       size_t *InpBufIndex);
```

General description

The m_transform_layout() function is part of the support for handling of bidirectional (Bidi) conversion of data between Visual (MVS) and Implicit (z/OS UNIX) formats. Initial support is for Arabic and Hebrew data.

The m_transform_layout() function performs layout transformations (reordering, shaping, cell determination). Alternatively, it may provide additional information needed for layout transformation, such as:

- The expected size of the transformed layout

- The nesting level of different segments in the text
- Cross references between the locations of the corresponding elements before and after the layout transformation.

Both the input text and output text are character strings. The m_transform_layout() function transforms the input text in *InpBuf* according to the current layout values in *layout_object*.

Any layout value whose value type is **LayoutTextDescriptor** describes the attributes of the *InpBuf* and *OutBuf* arguments. If the attributes are the same for both *InpBuf* and *OutBuf*, a NULL transformation is performed with respect to that specific layout value.

The *InpBuf* argument specifies the source text to be processed. The *InpBuf* may not be NULL, except when there is a need to reset the internal state.

The *InpSize* argument is the number of bytes within *InpBuf* to be processed by the transformation. Its value will not have changed at the return from the transformation. *InpSize* set to -1 indicates that the text in *InpBuf* is delimited by a NULL code element. If *InpSize* is not set to -1, it is possible to have some NULL elements in the input buffer. This might be used, for example, for a *one shot* transformation of several strings, separated by NULLs.

Outputs of this function may be one or more of the following, depending on the setting of the arguments:

Output
: **Description**

OutBuf
: Any transformed data is stored in *OutBuf*, converted to **ShapeCharset**.

Outsize
: The number of bytes in *OutBuf*.

InpToOut
: A cross reference from each *InpBuf* code element to the transformed data. The cross reference relates to the data in *InpBuf*, starting with the first element that *InpBufIndex* points to (and not necessarily starting from the beginning of the *InpBuf*).

OutToInp
: A cross reference to each *InpBuf* code element from the transformed data. The cross reference relates to the data in *InpBuf*, starting with the first element that *InpBufIndex* points to (and not necessarily starting from the beginning of the *InpBuf*).

Property
: A weighted value that represents specific input string transformation properties with different connotations as explained below. If this argument is not a NULL pointer, it represents an array of values with the same number of elements as the source substring text before the transformation.

  Each byte will contain relevant *property* information of the corresponding element in *InpBuf*, starting from the element pointed by *InpBufIndex*.

  The four rightmost bits of each *property* byte will contain information for bidirectional environments (when **ActiveDirectional** is True) and they will mean **NestingLevels**. The possible value from 0 to 15 represents the nesting level of the corresponding element in the *InpBuf*, starting from the element pointed by *InpBufIndex*. If **ActiveDirectional** is False, the content of NestingLevels bits will be ignored.

The leftmost bit of each *property* byte will contain a *new cell indicator* for composed character environments. It will be a value of either 1, for an element in *InpBuf* that is transformed to the beginning of a new cell, or 0, for the *zero-length* composing character elements when these are grouped into the same presentation cell with a non-composing character. Here again, each element of *property* pertains to the elements in the *InpBuf*, starting from the element pointed by *InpBufIndex*. (Remember that this is not necessarily the beginning of InpBuf.)

If none of the transformation properties is required, the argument *Property* can be NULL.

The use of *property* can be enhanced in the future to pertain to other possible usage in other environments.

InpBufIndex

An offset value to the location of the transformed text. When m_transform_layout() is called, *InpBufIndex* contains the offset to the element in *InpBuf* that will be transformed first. (Note that this is not necessarily the first element in *InpBuf*.)

At the return from the transformation, *InpBufIndex* contains the offset to the first element in the *InpBuf* that has not been transformed. If the entire substring has been transformed successfully, *InpBufIndex* will be incremented by the amount defined by *InpSize*.

Each of these output arguments may be NULL to specify that no output is desired for the specific argument, but at least one of them should be set to non-NULL to perform any significant work.

The *layout_object* maintains a directional state that keeps track of directional changes, based on the last segment transformed. The directional state is maintained across calls to the layout transformation functions and allows stream data to be processed with the layout functions. The directional state is reset to its initial state whenever any of the layout values **TypeOfText**, **Orientation** or **ImplicitAlg** is modified by means of a call to m_setvalues_layout().

The *layout_object* argument specifies a LayoutObject returned by the m_create_layout() function.

The *OutBuf* argument contains the transformed data. This argument can be specified as a NULL pointer to indicate that no transformed data is required. The encoding of the *OutBuf* argument depends on the **ShapeCharset** layout value defined in *layout_object*. If the **ActiveShapeEditing** layout value is not set (False), the encoding of *OutBuf* is guaranteed to be the same as the codeset of the locale associated with the LayoutObject defined by *layout_object*.

On input, the *OutSize* argument specifies the size of the output buffer in number of bytes. The output buffer should be large enough to contain the transformed result; otherwise, only a partial transformation is performed. If the **ActiveShapeEditing** layout value is set (True), the *OutBuf* should be allocated to contain at least the *InpSize* multiplied by **ShapeCharsetSize**.

OutSize

Upon return, the *OutSize* argument is updated to be the actual number of bytes placed in *OutBuf*.

When the *OutSize* argument is specified as zero, the function calculates the size of an output buffer large enough to contain the transformed text, and the result is returned in this field. The content of the buffers specified by *InpBuf* and *OutBuf*, and the value of *InpBufIndex*, remain unchanged.

If *OutSize* = NULL, the EINVAL error condition is returned.

If the *InpToOut* argument is not a NULL pointer, it points to an array of values with the same number of bytes as *InpBuf*, starting with the one pointed by *InpBufIndex* and up to the end of the substring in the buffer.

On output, the nth value in *InpToOut* corresponds to the nth byte in *InpBuf*. This value is the index (in units of bytes) in *OutBuf* that identifies the transformed **ShapeCharset** element of the nth byte in *InpBuf*.

In the case of multibyte encoding, for each of the bytes of a code element in the *InpBuf*, the index points to the first byte of the transformed code element in the *OutBuf*. *InpToOut* may be specified as NULL if no index array from *InpBuf* to *OutBuf* is desired.

If the *OutToInp* argument is not a NULL pointer, it points to an array of values with the same number of bytes as contained in *OutBuf*. On output, the nth value in *OutToInp* corresponds to the nth byte in *OutBuf*. This value is the index in *InpBuf*, starting with the byte pointed to by *InpBufIndex*, that identifies the logical code element of the nth byte in *OutBuf*.

In the case of multibyte encoding, the index will point, for each of the bytes of a transformed code element in the *OutBuf*, to the first byte of the code element in the *InpBuf*.

OutToInp may be specified as NULL if no index array from *OutBuf* to *InpBuf* is desired.

To perform shaping of a text string without reordering of code elements, the *layout_object* should be set with input and output layout value **TypeOfText** set to TEXT_VISUAL, and both in and out of **Orientation** set to the same value.

Returned value

If successful, m_transform_layout() returns 0.

If unsuccessful, m_transform_layout() returns -1 and sets errno to one of the following values:

Error Code
Description

E2BIG The size of *OutBuf* is not large enough to contain the entire transformed text. The input text state at the end of the uncompleted transformation is saved internally.

EBADF
The layout values are set to a meaningless combination or the layout object is not valid.

EINVAL
Transformation stopped due to an incomplete composite sequence at the end of the input buffer, or *OutSize* contains NULL.

Related information

- "sys/layout.h" on page 73
- "m_create_layout() — Create and initialize a layout object (bidi data)" on page 1058
- "m_destroy_layout() — Destroy a layout object (bidi data)" on page 1059
- "m_getvalues_layout() — Query layout values of a layout object (bidi data)" on page 1067
- "m_setvalues_layout() — Set layout values of a layout object (bidi data)" on page 1103
- "m_wtransform_layout() — Layout transformation for wide-character strings (bidi data)" on page 1125

munmap() — Unmap pages of memory

Standards

Standards / Extensions	C or C++	Dependencies
XPG4.2 Single UNIX Specification, Version 3 z/OS UNIX	both	

Format

```
#define _XOPEN_SOURCE_EXTENDED 1
#include <sys/mman.h>

int munmap(void *addr, size_t len);
```

General description

The munmap() function removes the mappings for pages in the range [*addr, addr + len*) rounding the *len* argument up to the next multiple of the page size as returned by sysconf(). If *addr* is not the address of a mapping established by a prior call to mmap(), the behavior is undefined. After a successful call to munmap() and before any subsequent mapping of the unmapped pages, further references to these pages will result in the delivery of a SIGBUS or SIGSEGV signal to the process.

__MAP_MEGA mapping: The munmap service removes the mapping for pages in the requested range. The requested range may span multiple maps, and the maps may represent the same or different files. The pages in the range may be part of a regular mapping or may be part of a __MAP_MEGA mapping. When unmapping a regular mapping, entire pages are unmapped; when unmapping a __MAP_MEGA mapping, entire segments are unmapped.

Map_address: The value of map address must be a multiple of the page size. The specified value does not have to be the start of a mapping. However, if the value specified for Map_address falls within a __MAP_MEGA map, then the address is rounded down to a megabyte multiple so that an entire segment is included in the unmap operation. It is not possible to unmap a part of a segment when processing a __MAP_MEGA map.

Map_length: The length can be the size of the whole mapping, or a part of it. If the specified length is not in multiples of the page size, it will be rounded up to a page boundary. If the Map_address plus the Map_length falls within a __MAP_MEGA map, then the length is rounded up to a segment boundary, thus including the entire segment (not necessarily the entire __MAP_MEGA mapping).

Returned value

If successful, munmap() returns 0.

If unsuccessful, munmap() returns -1 and sets errno to one of the following values:

Error Code
Description

EINVAL
One of the following error conditions exists:
- The *addr* argument is not a multiple of the page size as returned by *sysconf*.
- Addresses outside the valid range for the address space of a process.
- The *len* argument is 0.

Related information

- “sys/mman.h” on page 73
- “mmap() — Map pages of memory” on page 1081
- “sysconf() — Determine system configuration options” on page 1798

__must_stay_clean() — Enable or query clean

Standards

Standards / Extensions	C or C++	Dependencies
z/OS UNIX	both	Z/OS V1R8

Format

```
#define _OPEN_SYS
#include <unistd.h>

int __must_stay_clean(int request);
```

General description

The __must_stay_clean() function queries or enables the "must stay clean" state for a process. A process that must stay clean is prohibited from doing an exec(), spawn(), or other load of a non-program controlled executable. Only a program controlled executable can enable the must stay clean state. The must stay clean state of a process is propagated to its children created using fork or spawn. Once the must stay clean state is enabled, it cannot be changed. All processes in the address space will be forced to stay clean until they have all terminated. The query support allows a process to determine if it was created in a trusted environment. The BPX.DAEMON class profile must be defined to use the enable function.

Argument
Description

request
Specify the value _MSC_QUERY to query the state. Specify the value _MSC_ENABLE to enable "must stay clean" for the process.

Returned value

If successful, __must_stay_clean() returns the current "must stay clean" state of the process. The following state values are possible:

_MSC_NOT_ENABLED
The "must stay clean" state is not enabled.

_MSC_ENABLED
The "must stay clean" state is enabled, meaning that it was set using this function, and that it will continue to be enabled even after an exec() that causes job step termination.

_MSC_ENABLED_COND
The "must stay clean" state is enabled conditionally, meaning that a prior call to a security service, such as __passwd(), implicitly enabled the must stay clean state, and that the state will be reset to "not enabled" at the next exec() that causes job step termination. This state value can only be returned using the query request.

If unsuccessful, __must_stay_clean() returns _MSC_FAILED(-1) and sets errno to one of the following values:

Error Code
Description

EINVAL
A parameter was not valid.

EMVSERR
An MVS environmental error occurred. One possible cause is that a 'dirty' process attempted to enable the must stay clean attribute. Another cause could be that the BPX.DAEMON class profile is not defined.

EMVSSAF2ERR
An error occurred in the security product.

Example

```
/* celebm22.c */
/* This example shows how to use __must_stay_clean() to request  */
/* the environment is to "stay clean" until all processes in the */
/* address space are terminated.                                 */
/* Requirements:                                                 */
/* 1. The environment must already be clean, noting that the     */
/*    program issuing the request must be program-controlled     */
/* 2. BPX.DAEMON must be defined                                 */

#define _OPEN_SYS
#include <unistd.h>
#include <errno.h>
#include <stdio.h>
#include <stdlib.h>

int main(void){
 int rc;
 rc = __must_stay_clean(_MSC_ENABLE); /* Stay Clean! */
 if (rc == __MSC_FAILED){
  perror("could not enable must stay clean");
  printf("errno=%d errno2=%08x\n",errno,__errno2());
  exit(1);
 }
 return 0;
}
```

Related information

m_wtransform_layout() — Layout transformation for wide-character strings (bidi data)

Standards

Standards / Extensions	C or C++	Dependencies
ISO C XPG4 C99	both	z/OS V1R2

Format

```
#include <sys/layout.h>

int m_wtransform_layout(LayoutObject layout_object,
                        const wchar_t *InpBuf,
                        const size_t InpSize,
                        void *OutBuf,
                        size_t *Outsize,
                        size_t *InpToOut,
                        size_t *OutToInp,
                        unsigned char *Property,
                        size_t *InpBufIndex);
```

General description

The m_wtransform_layout() function is part of the support for handling of bidirectional (Bidi) conversion of data between Visual (MVS) and Implicit (z/OS UNIX) formats. Initial support is for Arabic and Hebrew data.

The m_wtransform_layout() function performs layout transformations (reordering, shaping, cell determination). Alternatively, it may provide additional information needed for layout transformation, such as:

- The expected size of the transformed layout
- The nesting level of different segments in the text
- Cross references between the locations of the corresponding elements before and after the layout transformation.

Both the input text and output text are wide-character strings. The m_wtransform_layout() function transforms the input text in *InpBuf* according to the current layout values in *layout_object*.

Any layout value whose value type is **LayoutTextDescriptor** describes the attributes of the *InpBuf* and *OutBuf* arguments. If the attributes are the same for both *InpBuf* and *OutBuf*, a NULL transformation is performed with respect to that specific layout value.

The *InpBuf* argument specifies the source text to be processed. The *InpBuf* may not be NULL, except when there is a need to reset the internal state.

The *InpSize* argument is the number of characters within *InpBuf* to be processed by the transformation. Its value will not have changed at the return from the transformation. *InpSize* set to -1 indicates that the text in *InpBuf* is delimited by a NULL code element. If *InpSize* is not set to -1, it is possible to have some NULL elements in the input buffer. This might be used, for example, for a *one shot* transformation of several strings, separated by NULLs.

Outputs of this function may be one or more of the following, depending on the setting of the arguments:

Argument
: **Description**

OutBuf
: Any transformed data is stored in *OutBuf*, converted to **ShapeCharset**.

Outsize
: The number of wide characters in *OutBuf*.

InpToOut
: A cross reference from each *InpBuf* code element to the transformed data. The cross reference relates to the data in *InpBuf*, starting with the first element that *InpBufIndex* points to (and not necessarily starting from the beginning of the *InpBuf*.)

OutToInp
: A cross reference to each *InpBuf* code element from the transformed data. The cross reference relates to the data in *InpBuf*, starting with the first element that *InpBufIndex* points to (and not necessarily starting from the beginning of the *InpBuf*.)

Property
: A weighted value that represents specific input string transformation properties with different connotations as explained below. If this argument is not a NULL pointer, it represents an array of values with the same number of elements as the source substring text before the transformation.

  Each byte will contain relevant *property* information of the corresponding element in *InpBuf*, starting from the element pointed by *InpBufIndex*.

  The four rightmost bits of each *property* byte will contain information for bidirectional environments (when **ActiveDirectional** is True) and they will mean **NestingLevels**. The possible value from 0 to 15 represents the nesting level of the corresponding element in the *InpBuf*, starting from the element pointed by *InpBufIndex*. If **ActiveDirectional** is False, the content of NestingLevels bits will be ignored.

  The leftmost bit of each *property* byte will contain a *new cell indicator* for composed character environments. It will be a value of either 1, for an element in *InpBuf* that is transformed to the beginning of a new cell, or 0, for the *zero-length* composing character elements, when these are grouped into the same presentation cell with a non-composing character. Here again, each element of *property* pertains to the elements in the *InpBuf*, starting from the element pointed by *InpBufIndex*. (Remember that this is not necessarily the beginning of InpBuf.)

  If none of the transformation properties is required, the argument *Property* can be NULL.

  The use of *property* can be enhanced in the future to pertain to other possible usage in other environments.

InpBufIndex

An offset value to the location of the transformed text. When m_wtransform_layout() is called, *InpBufIndex* contains the offset to the element in *InpBuf* that will be transformed first. (Note that this is not necessarily the first element in *InpBuf*.)

At the return from the transformation, *InpBufIndex* contains the offset to the first element in the *InpBuf* that has not been transformed. If the entire substring has been transformed successfully, *InpBufIndex* will be incremented by the amount defined by *InpSize*.

Each of these output arguments may be NULL to specify that no output is desired for the specific argument, but at least one of them should be set to non-NULL to perform any significant work.

In addition to the possible outputs above, the *layout_object* maintains a directional state across calls to the transform functions. The directional state is reset to its initial state whenever any of the layout values **TypeOfText**, **Orientation** or **ImplicitAlg** is modified by means of a call to m_setvalues_layout().

The *layout_object* argument specifies a LayoutObject returned by the m_create_layout() function.

The *OutBuf* argument contains the transformed data. This argument can be specified as a NULL pointer to indicate that no transformed data is required. The encoding of the *OutBuf* argument depends on the **ShapeCharset** layout value defined in *layout_object*. If the **ActiveShapeEditing** layout value is not set (False), the encoding of *OutBuf* is guaranteed to be the same as the codeset of the locale associated with the LayoutObject defined by *layout_object*.

On input, the *OutSize* argument specifies the size of the output buffer in number of wide characters. The output buffer should be large enough to contain the transformed result; otherwise, only a partial transformation is performed. If the **ActiveShapeEditing** layout value is set (True), the *OutBuf* should be allocated to contain at least the *InpSize* multiplied by **ShapeCharsetSize**.

OutSize

Upon return, the *OutSize* argument is updated to be the actual number of code elements placed in *OutBuf*.

When the *OutSize* argument is specified as zero, the function calculates the size of an output buffer large enough to contain the transformed text, and the result is returned in this field. The content of the buffers specified by *InpBuf* and *OutBuf*, and the value of *InpBufIndex*, remain unchanged.

If *OutSize* = NULL, the EINVAL error condition is returned.

If the *InpToOut* argument is not a NULL pointer, it points to an array of values with the same number of wide characters as *InpBuf*, starting with the one pointed by *InpBufIndex* and up to the end of the substring in the buffer.

On output, the nth value in *InpToOut* corresponds to the nth wide character in *InpBuf*. This value is the index (in units of wide characters) in *OutBuf* that identifies the transformed **ShapeCharset** element of the nth wide character in *InpBuf*.

InpToOut may be specified as NULL if no index array from *InpBuf* to *OutBuf* is desired.

If the *OutToInp* argument is not a NULL pointer, it points to an array of values with the same number of wide characters as contained in *OutBuf*. On output, the nth value in *OutToInp* corresponds to the nth wide character in *OutBuf*. This value is the index in *InpBuf*, starting with the wide character pointed to by *InpBufIndex*, that identifies the logical code element of the nth byte in *OutBuf*.

OutToInp may be specified as NULL if no index array from *OutBuf* to *InpBuf* is desired.

To perform shaping of a text string without reordering of code elements, the *layout_object* should be set with input and output layout value **TypeOfText** set to TEXT_VISUAL, and both in and out of **Orientation** set to the same value.

Returned value

If successful, m_wtransform_layout() returns 0.

If unsuccessful, m_wtransform_layout() returns -1 and sets errno to one of the following values:

Error Code
Description

E2BIG The size of *OutBuf* is not large enough to contain the entire transformed text. The input text state at the end of the uncompleted transformation is saved internally.

EBADF
The layout values are set to a meaningless combination or the layout object is not valid.

EINVAL
Transformation stopped due to an incomplete composite sequence at the end of the input buffer, or *OutSize* contains NULL.

Related information

- “sys/layout.h” on page 73
- “m_create_layout() — Create and initialize a layout object (bidi data)” on page 1058
- “m_destroy_layout() — Destroy a layout object (bidi data)” on page 1059
- “m_getvalues_layout() — Query layout values of a layout object (bidi data)” on page 1067
- “m_setvalues_layout() — Set layout values of a layout object (bidi data)” on page 1103
- “m_transform_layout() — Layout transformation for character strings (bidi data)” on page 1118

nan(), nanf(), nanl() — Return quiet NaN

Standards

Standards / Extensions	C or C++	Dependencies
C99 Single UNIX Specification, Version 3 C++ TR1 C99	both	z/OS V1R7

Format

```
#define _ISOC99_SOURCE
#include <math.h>

double nan(const char *tagp);
float nanf(const char *tagp);
long double nanl(const char *tagp);
```

General description

In the nan() family of functions, the call nan("n-char-sequence") is equivalent to strtod("NAN(n-charsequence)", (char**) NULL) and the call nan("") is equivalent to strtod("NAN()", (char**) NULL). If *tagp* does not point to an n-char sequence or an empty string, the call is equivalent to strtod("NAN", (char**) NULL). Calls to nanf() and nanl() are equivalent to the corresponding calls strtof() and strtold().

Note: The following table shows the viable formats for these functions. See “IEEE binary floating-point” on page 94 for more information about IEEE Binary Floating-Point.

Function	Hex	IEEE
nan	X	X
nanf	X	X
nanl	X	X

Returned value

If successful, they return a quiet NaN with content indicated by *tagp*.

Special behavior in hex: The nan() family of functions always return 0.

Example

```
/*
 * This program illustrates the use of the nan() function
 *
 * It calls both nan and strtod with equivalent arguments
 * and displays output of both. Output should be identical.
 *
 */
#define _ISOC99_SOURCE
#include <stdio.h>
#include <stdlib.h>         /* needed of strtod */
#include <math.h>


#define TESTVALS 5
struct {
   const char * str;
} nan_vals[] = {
/*0*/  { "0"           },
/*1*/  { "1"           },
/*2*/  { "something"   },   /* invalid n-char seq. */
/*3*/  { "2147483647"  },   /* int max      */
/*4*/  { "2147483648"  }    /* int max  +1  */
},

strod_vals[] = {
/*0*/  { "NAN(0)"          },
/*1*/  { "NAN(1)"          },
/*2*/  { "NAN"             },
/*3*/  { "NAN(2147483647)" },
/*4*/  { "NAN(2147483648)" }
};


void main()
{
  double outnan,
         outstrtod;
  int i;
  char *tagp = (char *)NULL;

  printf("Illustrates the nan() function\n");
  printf("Output for both nan() and strtod() should be identical.\n\n");
  for (i=0; i<TESTVALS; i++) {
        outnan = nan(nan_vals[i].str);
        outstrtod = strtod(strod_vals[i].str, &tagp);

        printf("nan(%s)         returned = %g\n",nan_vals[i].str, outnan);
        printf("strtod(%s) returned = %g\n\n",strod_vals[i].str, outstrtod);
  }

}
```

```
Output

Illustrates the nan() function
Output for both nan() and strtod() should be identical.

nan(0)         returned = 0
strtod(NAN(0)) returned = 0

nan(1)         returned = NaNQ(1)
strtod(NAN(1)) returned = NaNQ(1)

nan(something)         returned = NaNQ(1)
strtod(NAN) returned = NaNQ(1)

nan(2147483647)         returned = NaNQ(2147483647)
strtod(NAN(2147483647)) returned = NaNQ(2147483647)

nan(2147483648)         returned = 0
strtod(NAN(2147483648)) returned = 0
```

Related information

- "math.h" on page 44
- "strtod() — Convert character string to double" on page 1765

nand32(), nand64(), nand128() — Return quiet NaN

Standards

Standards / Extensions	C or C++	Dependencies
C/C++ DFP	both	z/OS V1.8

Format

```
#define __STDC_WANT_DEC_FP__
#include <math.h>

_Decimal32  nand32(const char *tagp);
_Decimal64  nand64(const char *tagp);
_Decimal128 nand128(const char *tagp);
```

General description

In the nan() family of functions, the call nand32("n-char-sequence") is equivalent to strtod32("NAN(n-charsequence)", (char**) NULL) and the call nand32("") is equivalent to strtod32("NAN()", (char**) NULL). If tagp does not point to an n-char sequence or an empty string, the call is equivalent to strtod32("NAN", (char**) NULL). Calls to nand64() and nand128() are equivalent to the corresponding calls strtod64() and strtod128().

Notes:

1. To use IEEE decimal floating-point, the hardware must have the Decimal Floating-Point Facility installed.
2. These functions work in IEEE decimal floating-point format. See "IEEE Decimal Floating-Point" for more information.

Returned value

If successful, they return a quiet NaN with content indicated by tagp.

Example

```
/* CELEBN05

   This program illustrates the use of the nand32() function.

   It calls both nand32() and strtod32() with equivalent arguments
   and displays output of both. Output should be identical.

*/

#define  __STDC_WANT_DEC_FP__
#include <math.h>
#include <stdio.h>
#include <stdlib.h>          /* needed for strtod32()                */


#define TESTVALS 5

struct
{
  const char * str;
}
nan_vals[] =
{
 /*0*/  { "0"           },
 /*1*/  { "1"           },
 /*2*/  { "something"   },   /* invalid n-char seq.                  */
 /*3*/  { "999999"      },   /* max nancode                          */
 /*4*/  { "1000000"     }    /* max nancode + 1                      */
}
,
strod_vals[] =
{
 /*0*/  { "NAN(0)"      },
 /*1*/  { "NAN(1)"      },
 /*2*/  { "NAN"         },
 /*3*/  { "NAN(999999)"  },
 /*4*/  { "NAN(1000000)" }
};


int main(void)
{
  _Decimal32 outnan,
             outstrtod;
  int        i;

  printf("Illustrates the nand32() function\n");
  printf("Output for both nand32() and strtod32()"
         "should be identical.\n\n");

  for (i = 0; i < TESTVALS; i++)
  {
    outnan    =   nand32(  nan_vals[i].str      );
    outstrtod = strtod32(strod_vals[i].str, NULL);

    printf("nand32(%s)        returned = %Hg\n"
          ,   nan_vals[i].str, outnan   );
    printf("strtod32(%s) returned = %Hg\n\n"
          , strod_vals[i].str, outstrtod);
  }

  return 0;
}
```

Related information

- "math.h" on page 44
- "nan(), nanf(), nanl() — Return quiet NaN" on page 1128
- "strtod32(), strtod64(), strtod128() — Convert character string to decimal floating point" on page 1767

nearbyint(), nearbyintf(), nearbyintl() — Round the argument to the nearest integer

Standards

Standards / Extensions	C or C++	Dependencies
C99 Single UNIX Specification, Version 3 C++ TR1 C99	both	z/OS V1R7

Format

```
#define _ISOC99_SOURCE
#include <math.h>

double nearbyint(double x);
float nearbyintf(float x);
long double nearbyintl(long double x);
```

C++ TR1 C99:

```
#define _TR1_C99
#include <math.h>

float nearbyint(float x);
long double nearbyint(long double x);
```

General description

The nearbyint() family of functions round *x* to an integer value, in floating-point format, using the current rounding mode without raising the **inexact** floating-point exception.

Note: The following table shows the viable formats for these functions. See “IEEE binary floating-point” on page 94 for more information about IEEE Binary Floating-Point.

Function	Hex	IEEE
nearbyint	X	X
nearbyintf	X	X
nearbyinfl	X	X

Returned value

If successful, they return the rounded integer value. If the correct value causes an overflow, a range error occurs and the value, respectively, of the macro: +/-HUGE_VAL, +/-HUGE_VALF, or +/-HUGE_VALL (with the same sign as *x*) is returned.

Example

```
/*
 * This program illustrates the use of nearbyint() function
 *
 * Note: to get the results shown in this book , this program
 *       should be compiled using FLOAT(IEEE)
 *
 */
#define _ISOC99_SOURCE
#include <math.h>
#include <stdio.h>
#include <_Ieee754.h>   /* save/get fpc functions   */

char *RoundStr (_FP_rmode_t rm_type) {
  char *RndStr="undetermined";
   switch (rm_type) {
     case (_RMODE_RN):
        RndStr="round to nearest";
        break;
     case (_RMODE_RZ):
        RndStr="round toward zero";
        break;
     case (_RMODE_RP):
        RndStr="round toward +infinity ";
        break;
     case (_RMODE_RM):
        RndStr="round toward -infinity ";
        break;
   }
   return (RndStr);
}


void main() {

  _FP_fpcreg_t save_rmode, current_rmode;
  double       rnd2nearest;
  double       number1=1.5,
               number2=-3.92;

  printf("Illustrates the nearbyint() function\n");
  __fpc_rd(&current_rmode);      /* get current rounding mode */

  rnd2nearest = nearbyint(number1);
  printf ("When rounding direction is %s:\n nearbyint(%.2f) = %f\n",
           RoundStr(current_rmode.rmode),number1, rnd2nearest);
  save_rmode.rmode = _RMODE_RZ;
  __fpc_sm(save_rmode.rmode);    /* set rounding mode to round to zero */
  rnd2nearest = nearbyint(number2);
  printf ("When rounding direction is %s:\n nearbyint(%.2f) = %f\n",
           RoundStr(save_rmode.rmode),number2, rnd2nearest);
}
```

```
Output

Illustrates the nearbyint() function
When rounding direction is round to nearest:
 nearbyint(1.50) = 2.000000
When rounding direction is round toward zero:
 nearbyint(-3.91) = -3.000000
```

Related information

- “math.h” on page 44
- “ceil(), ceilf(), ceill() — Round up to integral value” on page 249
- “floor(), floorf(), floorl() — Round down to integral value” on page 548
- “llround(), llroundf(), llroundl() — Round to the nearest integer” on page 975
- “lrint(), lrintf(), lrintl() and llrint(), llrintf(), llrintl() — Round the argument to the nearest integer” on page 1012
- “lround(), lroundf(), lroundl() — Round a decimal floating-point number to its nearest integer” on page 1016
- “rint(), rintf(), rintl() — Round to nearest integral value” on page 1450
- “round(), roundf(), roundl() — Round to the nearest integer” on page 1455
- “trunc(), truncf(), truncl() — Truncate an integer value” on page 1915

nearbyintd32(), nearbyintd64(), nearbyintd128() — Round the argument to the nearest integer

Standards

Standards / Extensions	C or C++	Dependencies
C/C++ DFP	both	z/OS V1.8

Format

```
#define __STDC_WANT_DEC_FP__
#include <math.h>

_Decimal32  nearbyintd32(_Decimal32 x);
_Decimal64  nearbyintd64(_Decimal64 x);
_Decimal128 nearbyintd128(_Decimal128 x);
_Decimal32  nearbyint(_Decimal32 x);     /* C++ only */
_Decimal64  nearbyint(_Decimal64 x);     /* C++ only */
_Decimal128 nearbyint(_Decimal128 x);    /* C++ only */
```

General description

The nearbyint() family of functions round *x* to an integer value, in decimal floating-point format, using the current rounding mode without raising the inexact decimal floating-point exception.

Notes:

1. To use IEEE decimal floating-point, the hardware must have the Decimal Floating-Point Facility installed.
2. These functions work in IEEE decimal floating-point format. See "IEEE Decimal Floating-Point" for more information.

Returned value

If successful, they return the rounded integer value. If the correct value causes an overflow, a range error occurs and the value, respectively, of the macro: ±HUGE_VAL_D32, ±HUGE_VAL_D64, or ±HUGE_VAL_D128 (with the same sign as *x*) is returned.

Example

```
/* CELEBN06

   This example illustrates the nearbyintd64() function.

*/

#pragma strings(readonly)

#define __STDC_WANT_DEC_FP__
#include <fenv.h>
#include <math.h>
#include <stdio.h>

static void try_rm(int);
/* pass back printable rounding mode */

static
char *rm_str(int rm)
{
  char *s = "undetermined";

  switch (rm)
  {
    case    FE_DEC_TONEAREST            :
      s =  "FE_DEC_TONEAREST"           ;  break;
    case    FE_DEC_TOWARDZERO           :
      s =  "FE_DEC_TOWARDZERO"          ;  break;
    case    FE_DEC_UPWARD               :
      s =  "FE_DEC_UPWARD"              ;  break;
    case    FE_DEC_DOWNWARD             :
      s =  "FE_DEC_DOWNWARD"            ;  break;
    case    FE_DEC_TONEARESTFROMZERO    :
      s =  "FE_DEC_TONEARESTFROMZERO"   ;  break;
    case   _FE_DEC_TONEARESTTOWARDZERO  :
      s = "_FE_DEC_TONEARESTTOWARDZERO" ;  break;
    case   _FE_DEC_AWAYFROMZERO         :
      s = "_FE_DEC_AWAYFROMZERO"        ;  break;
    case   _FE_DEC_PREPAREFORSHORTER    :
      s = "_FE_DEC_PREPAREFORSHORTER"   ;  break;
   }

   return s;
}


/* Try out one passed-in number with rounding mode */

static void try_rm(int rm)
{
  _Decimal64 r64;
  _Decimal64 d64 = 500.99DD;

  (void)fe_dec_setround(rm);

  r64 = nearbyintd64(d64);

  printf("nearbyintd64(%.2DF) = %DG - rounding mode = %s\n",
         d64, r64, rm_str(rm)
        );

  return;
}


int main()
{
  try_rm( FE_DEC_TONEAREST          );
  try_rm( FE_DEC_TOWARDZERO         );
  try_rm( FE_DEC_UPWARD             );
  try_rm( FE_DEC_DOWNWARD           );
  try_rm( FE_DEC_TONEARESTFROMZERO  );
  try_rm(_FE_DEC_TONEARESTTOWARDZERO);
  try_rm(_FE_DEC_AWAYFROMZERO       );
  try_rm(_FE_DEC_PREPAREFORSHORTER  );

  return 0;
}
```

Related information

- "math.h" on page 44
- "ceild32(), ceild64(), ceild128() — Round up to integral value" on page 250
- "floord32(), floord64(), floord128() — Round down to integral value" on page 549
- "llroundd32(), llroundd64(), llroundd128() — Round to the nearest integer" on page 977
- "lrintd32(), lrintd64(), lrintd128() and llrintd32(), llrintd64(), llrintd128() — Round the argument to the nearest integer" on page 1014
- "lroundd32(), lroundd64(), lroundd128() — Round a floating-point number to its nearest integer" on page 1017
- "nearbyint(), nearbyintf(), nearbyintl() — Round the argument to the nearest integer" on page 1132
- "rintd32(), rintd64(), rintd128() — Round to nearest integral value" on page 1451
- "roundd32(), roundd64(), roundd128() — Round to the nearest integer" on page 1456
- "truncd32(), truncd64(), truncd128() — Truncate an integer value" on page 1916

nextafter(), nextafterf(), nextafterl() — Next representable double float

Standards

Standards / Extensions	C or C++	Dependencies
XPG4.2 C99 Single UNIX Specification, Version 3 C++ TR1 C99	both	

Format

```
#define _XOPEN_SOURCE_EXTENDED 1
#include <math.h>

double nextafter(double x, double y);
```

C99:

```
#define _ISOC99_SOURCE
#include <math.h>

float nextafterf(float x, float y);
long double nextafterl(long double x, long double y);
```

C++ TR1 C99:

```
#define _TR1_C99
#include <math.h>

float nextafter(float x, float y);
long double nextafter(long double x, long double y);
```

General description

The nextafter() function computes the next representable double-precision floating-point value following *x* in the direction of *y*. Thus, if *y* is less than *x*, nextafter() returns the largest representable floating-point number less than *x*.

Note: The following table shows the viable formats for these functions. See "IEEE binary floating-point" on page 94 for more information about IEEE Binary Floating-Point.

Function	Hex	IEEE
nextafter	X	X
nextafterf	X	X
nextafterl	X	X

Restriction: The nextafterf() function does not support the _FP_MODE_VARIABLE feature test macro.

Returned value

The nextafter() functions return the next representable value following x in the direction of y. They always succeed.

If x is finite and the correct function value overflows, a range error occurs and ±HUGE_VAL, ±HUGE_VALF, and ±HUGE_VALL (with the same sign as x) are returned as appropriate for the return type of the function.

Errno Descirption

ERANGE
The correct value overflows.

Related information

- "math.h" on page 44

nextafterd32(), nextafterd64(), nextafterd128() — Next representable decimal floating-point value

Standards

Standards / Extensions	C or C++	Dependencies
C/C++ DFP	both	z/OS V1.8

Format

```
#define __STDC_WANT_DEC_FP__
#include <math.h>

_Decimal32  nextafterd32(_Decimal32 x, _Decimal32 y);
_Decimal64  nextafterd64(_Decimal64 x, _Decimal64 y);
_Decimal128 nextafterd128(_Decimal128 x, _Decimal128 y);
_Decimal32  nextafter(_Decimal32 x, _Decimal32 y);  /* C++ only */
_Decimal64  nextafter(_Decimal64 x, _Decimal64 y);  /* C++ only */
_Decimal128 nextafter(_Decimal128 x, _Decimal128 y);/* C++ only */
```

General description

The nextafter() function computes the next representable decimal floating-point value following *x* in the direction of *y*. Thus, if *y* is less than *x*, nextafter() returns the largest representable decimal floating-point number less than *x*.

Notes:

1. To use IEEE decimal floating-point, the hardware must have the Decimal Floating-Point Facility installed.
2. These functions work in IEEE decimal floating-point format. See "IEEE Decimal Floating-Point" for more information.

Returned value

The nextafter() functions return the next representable value following *x* in the direction of *y*.

If...	**Then...**
x equals *y*	copysign(*x*,*y*) is returned.
x is less than *y*	the next representable value after *x* is returned.
x is greater than *y*	the largest representable decimal floating-point number less than *x* is returned.
x or *y* is a NaN	either *x* or *y* is returned.

Example

```
/* CELEBN07

   This example illustrates the nextafterd128() function.

*/

#define  __STDC_WANT_DEC_FP__
#include <math.h>
#include <stdio.h>

int main(void)
{
   _Decimal128 x = 123456789.70DL, dir = 123456790.00DL, z;

   z = nextafterd128(x, dir);

   printf("The next number after %DDf in the direction %DDf\n is %DDf\n",
          x, dir, z);
}
```

Related information

- "math.h" on page 44
- "nextafter(), nextafterf(), nextafterl() — Next representable double float" on page 1136

nexttoward(), nexttowardf(), nexttowardl() — Calculate the next representable value

Standards

Standards / Extensions	C or C++	Dependencies
C99 Single UNIX Specification, Version 3 C++ TR1 C99	both	z/OS V1R7

Format

```
#define _ISOC99_SOURCE
#include <math.h>

double nexttoward(double x, long double y);
float nexttowardf(float x, long double y);
long double nexttowardl(long double x, long double y);
```

C++ TR1 C99:

```
#define _TR1_C99
#include <math.h>

float nexttoward(float x, long double y);
long double nexttoward(long double x, long double y);
```

General description

The nexttoward() family of functions compute the next representable floating-point value following *x* in the direction of *y*.

Note: The following table shows the viable formats for these functions. See "IEEE binary floating-point" on page 94 for more information about IEEE Binary Floating-Point.

Function	Hex	IEEE
nexttoward	X	X
nexttowardf	X	X
nexttowardl	X	X

Restriction: The nexttowardf() function does not support the _FP_MODE_VARIABLE feature test macro.

Returned value

If successful, they return the next representable value in the specified format after *x* in the direction of *y*.

If...	Then...
x equals *y*	*y* (of type *x*) is returned.

If...	Then...
x is less than *y*	the next representable value after *x* is returned.
x is greater than *y*	the largest representable floating-point number less than *x* is returned.
x is finite and the correct function value would overflow	a range error occurs and +/-HUGE_VAL, +/-HUGE_VALF, or +/-HUGE_VALL (with the same sign as *x*) is returned by nexttoward(), nexttowardf() or nexttowardl() respectively.
x does not equal *y* and the correct subroutine value is subnormal, 0, or underflows	a range error occurs and either the correct function value (if representable) or 0.0 is returned.
x or *y* is a NaN	a NaN is returned.

Example

```
/*
 * This program illustrates the use of nexttoward() function
 */
#define _ISOC99_SOURCE
#include <math.h>
#include <stdio.h>


void PrintBytes(char *str, double x)
{
  static union {
    unsigned char bytes[sizeof(double)];
    double val;
  } dbl;

  int i;
  dbl.val = x;

  printf("%s ",str);
  for (i=0; i<sizeof(double); ++i) {
      printf("%02x", dbl.bytes[i]);
  }
  printf("\n");
}

void main() {

  double      nextvalue;
  double      x=1.5;
  long double  y=2.0;

  printf("Illustrates the nexttoward() function\n");

  printf("\nTest1 (x<y)   x = %f  y = %Lf\n",x,y);
  PrintBytes("x in hex =",x);
  nextvalue = nexttoward(x,y);
  printf ("nexttoward(x,y) = %f\n", nextvalue);
  PrintBytes("nexttoward(x,y) in hex =",nextvalue);

  x=1.5; y=1.0;
  printf("\nTest2 (x>y)   x = %f  y = %Lf\n",x,y);
  nextvalue = nexttoward(x,y);
  printf ("nexttoward(x,y) = %f\n", nextvalue);
  PrintBytes("nexttoward(x,y) in hex =",nextvalue);
}
```

```
Output

Illustrates the nexttoward() function

Test1 (x<y)   x = 1.500000  y = 2.000000
x in hex = 3ff8000000000000
nexttoward(x,y) = 1.500000
nexttoward(x,y) in hex = 3ff8000000000001

Test2 (x>y)   x = 1.500000  y = 1.000000
nexttoward(x,y) = 1.500000
nexttoward(x,y) in hex = 3ff7ffffffffffff
```

Related information

- "math.h" on page 44
- "copysign(), copysignf(), copysignl() — Copy the sign from one floating-point number to another" on page 326
- "nan(), nanf(), nanl() — Return quiet NaN" on page 1128
- "nextafter(), nextafterf(), nextafterl() — Next representable double float" on page 1136

nexttowardd32(), nexttowardd64(), nexttowardd128() — Calculate the next representable value

Standards

Standards / Extensions	C or C++	Dependencies
C/C++ DFP	both	z/OS V1.8

Format

```
#define __STDC_WANT_DEC_FP__
#include <math.h>

_Decimal32  nexttowardd32(_Decimal32 x, _Decimal128 y);
_Decimal64  nexttowardd64(_Decimal64 x, _Decimal128 y);
_Decimal128 nexttowardd128(_Decimal128 x, _Decimal128 y);
_Decimal32  nexttoward(_Decimal32 x, _Decimal128 y); /*C++ only*/
_Decimal64  nexttoward(_Decimal64 x, _Decimal128 y); /*C++ only*/
_Decimal128 nexttoward(_Decimal128 x, _Decimal128 y);/*C++ only*/
```

General description

The nexttoward() family of functions compute the next representable decimal floating-point value following *x* in the direction of *y*.

Notes:

1. To use IEEE decimal floating-point, the hardware must have the Decimal Floating-Point Facility installed.
2. These functions work in IEEE decimal floating-point format. See "IEEE Decimal Floating-Point" for more information.

Returned value

If successful, they return the next representable value in the specified format after x in the direction of *y*.

If...	Then...
x equals *y*	*y* (of type *x*) is returned.
x is less than *y*	the next representable value after *x* is returned.
x is greater than *y*	the largest representable decimal floating-point number less than *x* is returned.
x is finite and the correct function value would overflow	a range error occurs and ±HUGE_VAL_D32, ±HUGE_VAL_D64, or ±HUGE_VAL_D128 (with the same sign as *x*) is returned by nexttowardd32(), nexttowardd64() or nexttowardd128(), respectively.
x does not equal *y* and the correct subroutine value is subnormal, 0, or underflows	a range error occurs and either the correct function value (if representable) or 0.0 is returned.
x or *y* is a NaN	a NaN is returned.

Example

```
/* CELEBN08

   This example illustrates the nexttowardd32() function.

*/

#define __STDC_WANT_DEC_FP__
#include <math.h>
#include <stdio.h>


static void try_nt(_Decimal32 x, _Decimal128 y)
{
  _Decimal32 r = nexttowardd32(x, y);
  printf("nexttowardd32(%12.12HG, %12.12DDG) = % 12.12HG\n", x, y, r);
  return;
}


int main(void)
{
  try_nt( 2.000000DF    ,  2.00000001DL    );
  try_nt(-2.000000DF    , -2.00000001DL    );
  try_nt( 2.000000DF    ,  2.00000000DL    );
  try_nt( 2.000000DF    ,  1.99999999DL    );
  try_nt(-2.000000DF    , -1.99999999DL    );
  try_nt( 9.999999E+96DF,  9.99999999E+96DL);
  try_nt( 1.000000E-95DF,  0.99999999E-95DL);

  return 0;
}
```

Related information

- “math.h” on page 44
- “copysignd32(), copysignd64(), copysignd128() — Copy the sign from one floating-point number to another” on page 327
- “nand32(), nand64(), nand128() — Return quiet NaN” on page 1130
- “nextafterd32(), nextafterd64(), nextafterd128() — Next representable decimal floating-point value” on page 1137

- "nexttoward(), nexttowardf(), nexttowardl() — Calculate the next representable value" on page 1139

nftw() — Traverse a file tree

Standards

Standards / Extensions	C or C++	Dependencies
XPG4.2 Single UNIX Specification, Version 3	both	

Format

```
#define _XOPEN_SOURCE_EXTENDED 1
#include <ftw.h>

int nftw(const char *path,
        int (*fn)(const char *, const struct stat *, int, struct FTW *),
        int fd_limit, int flags);
```

General description

The nftw() function recursively descends the directory hierarchy rooted in *path*. It is similar to ftw() except that it takes an additional argument *flags*, which is a bitwise inclusive-OR of zero or more of the following flags:

FTW_CHDIR
If set, nftw() will change the current working directory to each directory as it reports files in that directory. If clear, nftw() will not change the current working directory.

FTW_DEPTH
If set, nftw() will report all files in a directory before reporting the directory itself. If clear, nftw() will report any directory before reporting the files in that directory.

FTW_MOUNT
If set, nftw() will only report files in the same file system as *path*. If clear, nftw() will report all files encountered during the walk.

FTW_PHYS
If set, nftw() performs a physical walk and does not follow symbolic links. If clear, nftw() will follow links instead of reporting them, and will not report the same file twice.

At each file it encounters, nftw() calls the user-supplied function *fn* with four arguments:

- the first argument is the pathname of the object.
- the second argument is a pointer to a stat buffer containing information on the object.
- the third argument is an integer giving additional information. Its value is one of the following:

  FTW_D
  for a directory

  FTW_DNR
  for a directory that cannot be read

FTW_DP
for a directory whose subdirectories have been visited. (This condition will only occur if FTW_DEPTH is included in *flags*.)

FTW_F
for a file

FTW_NS
for an object other than a symbolic link on which stat() could not be successfully executed. If the object is a symbolic link, and stat() failed, it is unspecified whether nftw() passes FTW_SL or FTW_NS to the user-supplied function.

FTW_SL
for a symbolic link

FTW_SLN
for a symbolic link that does not name an existing file. (This condition will only occur if FTW_PHYS is not included in *flags*.)

- the fourth argument is a pointer to an FTW structure. The value of *base* is the offset of the object's filename in the pathname passed as the first argument to *fn()*. The value of *level* indicates depth relative to the root of the walk, where the root level is 0.

The argument *fd_limit* limits the directory depth for the search. At most one file descriptor will be used for each directory level.

Large file support for z/OS UNIX files: Large z/OS UNIX files are supported automatically for AMODE 64 C/C++ applications. AMODE 31 C/C++ applications must be compiled with the option LANGLVL(LONGLONG) and define the _LARGE_FILES feature test macro before any headers are included to enable this function to operate on z/OS UNIX files that are larger than 2 GB in size. File size and offset fields are enlarged to 63 bits in width. Therefore, any other function operating on the file is required to define the _LARGE_FILES feature test macro as well.

Special behavior for C++: Because C and C++ linkage conventions are incompatible, nftw() cannot receive a C++ function pointer as the argument. If you attempt to pass a C++ function pointer to nftw(), the compiler will flag it as an error. You can pass a C or C++ function to nftw() by declaring it as extern “C”.

Returned value

nftw() continues until the first of the following conditions occurs:

- An invocation of *fn()* returns a nonzero value, in which case nftw() returns that value.
- The nftw() function detects an error other than EACCES (see FTW_DNR and FTW_NS above), in which case nftw() returns -1 and sets errno to indicate the error.
- The tree is exhausted, in which case nftw() returns 0.

If unsuccessful, nftw() sets errno to one of the following values. All other errnos returned by nftw() are unchanged.

Error Code
Description

EACCES
Search permission is denied for any component of *path* or read permission is denied for *path*, or *fn()* returns -1 and does not reset errno.

ELOOP
Too many symbolic links were encountered.

EMFILE
OPEN_MAX file descriptors are currently open in the calling process.

ENAMETOOLONG
One of the following error conditions exists:
- Pathname resolution of a symbolic link produced an intermediate result whose length exceeds PATH_MAX.
- The length of *path* exceeds PATH_MAX, or a pathname component is longer than PATH_MAX.

ENFILE
Too many files are currently open in the system.

ENOENT
A component of *path* does not name an existing file or *path* is an empty string.

ENOTDIR
A component of *path* is not a directory.

The errno value might also be set if the function *fn* causes it to be set.

Related information

- "ftw.h" on page 32
- "ftw() — Traverse a file tree" on page 666
- "lstat() — Get status of file or symbolic link" on page 1022
- "opendir() — Open a directory" on page 1158
- "readdir() — Read an entry from a directory" on page 1382
- "stat() — Get file information" on page 1720

nice() — Change priority of a process

Standards

Standards / Extensions	C or C++	Dependencies
XPG4 XPG4.2 Single UNIX Specification, Version 3	both	

Format

```
#define _XOPEN_SOURCE
#include <unistd.h>

int nice(int increment);
```

General description

nice() adds the value of *increment* to the nice value of the calling process. A process's nice value is a nonnegative number for which a more positive value results in a lower CPU priority.

A maximum nice value of 2*{NZERO}-1 and a minimum value of zero are imposed by the system. Requests for values above or below these limits result in the nice value being set to the corresponding limit. Only a process with appropriate privileges can lower the nice value.

The changing of a process's nice value has the equivalent effect on a process's scheduling priority value, since they both represent the process's relative CPU priority. For example, increasing one's nice value to its maximum value of (2*NZERO)-1 has the equivalent effect of setting one's scheduling priority value to its maximum value (19), and will be reflected on the nice(), getpriority(), and setpriority() functions.

Returned value

If successful, nice() return the new nice value minus (NZERO).

If unsuccessful, nice() returns -1 and sets errno to one of the following values:

Error Code
Description

ENOSYS
The system does not support this function.

EPERM
The value of *increment* was negative and the calling process does not have the appropriate privileges.

Because nice() can return the value -1 on successful completion, it is necessary to set the external variable errno to 0 before a call to nice(). If nice() returns -1, then errno can be checked to see if an error occurred or if the value is a legitimate nice value.

Related information

- “limits.h” on page 39
- “unistd.h” on page 82
- “getpriority() — Get process scheduling priority” on page 755
- “setpriority() — Set process scheduling priority” on page 1567

nlist() — Get entries from a name list

Standards

Standards / Extensions	C or C++	Dependencies
z/OS UNIX	both	

Format

```
#include <nlist.h>

int nlist(const char *loadname, struct nlist *np);
```

General description

The nlist() function allows a program to examine the name list in the executable file named by the *loadname* parameter. It selectively extracts a list of values and places them in the array of nlist structures pointed to by the *np* parameter.

The name list specified by the *np* parameter consists of an array of structures containing names of variables, types and values. The list is terminated with an element that has a NULL string in the name structure member. Each variable name is looked up in the name list of the executable file. If the name is found, the type and the value of the name is copied into the nlist structure field. If the name is not found, both the type and value entry will be set to zero.

All entries are set to zero if the specified executable file cannot be read or it does not contain a valid name list.

Notes:

1. The only variable type that will be supported by this version of nlist() is external function.
2. nlist() will extract the offset of the external functions from *loadname*.
3. The type returned in nlist structure will always be 2 to indicate function if the function name is found in *loadname*.
4. *loadname* must be a HFS linear format load module containing main().
5. *loadname* cannot be a dll (dynamic link library) or a fetchable load module.

Returned value

If successful, nlist() returns 0. The offset and type of functions if found will be returned in the nlist structure.

If unsuccessful, nlist() returns -1.

Related information

- "nlist.h" on page 57

nl_langinfo() — Retrieve locale information

Standards

Standards / Extensions	C or C++	Dependencies
XPG4 XPG4.2 Single UNIX Specification, Version 3	both	

Format

```
#include <langinfo.h>

char *nl_langinfo(nl_item item);
```

General description

Retrieves from the current locale the string that describes the requested information specified by *item*.

For a list of macros that define the constants used to identify the information queried in the current locale, see Table 4 on page 36.

Returned value

If successful, nl_langinfo() returns a pointer to a NULL-terminated string containing information concerning the active language or cultural area. The active language or cultural area is determined by the most recent setlocale() call. The array pointed to by the returned value is modified by subsequent calls to the function. The array shall not be modified by the user's program.

If the item is not valid, nl_langinfo() returns a pointer to an empty string.

Example

CELEBN01

```
/* CELEBN01

   This example retrieves the current codeset name using the
   &nll. function.

 */
#include "langinfo.h"
#include "locale.h"
#include "stdio.h"

main() {
   char *codeset;
   setlocale(LC_ALL, "");
   codeset = nl_langinfo(CODESET);
   printf("codeset is %s\n", codeset);
}
```

Related information

- “langinfo.h” on page 36
- “nl_types.h” on page 57
- “localdtconv() — Date and time formatting convention inquiry” on page 981
- “localeconv() — Query numeric conventions” on page 982
- “setlocale() — Set locale” on page 1552

nrand48() — Pseudo-random number generator

Standards

Standards / Extensions	C or C++	Dependencies
XPG4 XPG4.2 Single UNIX Specification, Version 3	both	

Format

```
#define _XOPEN_SOURCE
#include <stdlib.h>

long int nrand48(unsigned short int x16v[3]);
```

General description

The drand48(), erand48(), jrand48(), lrand48(), mrand48() and nrand48() functions generate uniformly distributed pseudo-random numbers using a linear congruential algorithm and 48-bit integer arithmetic.

The functions drand48() and erand48() return nonnegative, double-precision, floating-point values, uniformly distributed over the interval [0.0,1.0).

The functions lrand48() and nrand48() return nonnegative, long integers, uniformly distributed over the interval [0,2**31).

The functions mrand48() and jrand48() return signed long integers, uniformly distributed over the interval [-2**31,2**31).

The nrand48() function generates the next 48-bit integer value in a sequence of 48-bit integer values, X(i), according to the linear congruential formula:

```
X(n+1) = (aX(n) + c)mod(2**48)     n>=0
```

The nrand48() function uses storage provided by the argument array, *x16v*[3], to save the most recent 48-bit integer value in the sequence, X(i). The nrand48() function uses *x16v*[0] for the low-order (rightmost) 16 bits, *x16v*[1] for the middle-order 16 bits, and *x16v*[2] for the high-order 16 bits of this value.

The initial values of a, and c are:

```
a   = 5deece66d (base 16)
c   = b         (base 16)
```

The values a and c, may be changed by calling the lcong48() function. The initial values of a and c are restored if either the seed48() or srand48() function is called.

Special behavior for z/OS UNIX Services: You can make the nrand48() function and other functions in the drand48 family thread-specific by setting the environment variable _RAND48 to the value THREAD before calling any function in the drand48 family.

If you do not request thread-specific behavior for the drand48 family, C/370 serializes access to the storage for X(n), a and c by functions in the drand48 family when they are called by a multithreaded application.

If thread-specific behavior is requested and the nrand48() function is called from thread t, the nrand48() function generates the next 48-bit integer value in a sequence of 48-bit integer values, X(t,i), for the thread according to the linear congruential formula:

```
X(t,n+1) = (a(t)X(t,n) + c(t))mod(2**48)     n>=0
```

The nrand48() function uses storage provided by the argument array, *x16v*[3], to save the most recent 48-bit integer value in the sequence, X(t,i). The nrand48() function uses *x16v*[0] for the low-order (rightmost) 16 bits, *x16v*[1] for the middle-order 16 bits, and *x16v*[2] for the high-order 16 bits of this value.

The initial values of a(t) and c(t) on the thread t are:

```
a(t)   = 5deece66d (base 16)
c(t)   = b         (base 16)
```

The values a(t) and c(t) may be changed by calling the lcong48() function from the thread t. The initial values of a(t) and c(t) are restored if either the seed48() or srand48() function is called from the thread.

Returned value

nrand48() saves the generated 48-bit value, X(n+1), in storage provided by the argument array, *x16v*[3]. nrand48() transforms the generated 48-bit value to a nonnegative, long integer value on the interval [0,2**31) and returns this transformed value.

Special behavior for z/OS UNIX Services: If thread-specific behavior is requested for the drand48 family and nrand48() is called on thread `t`, nrand48() saves the generated 48-bit value, X(t,n+1), in storage provided by the argument array, *x16v*[3]. nrand48() transforms the generated 48-bit value to a nonnegative, long integer value on the interval [0,2**31) and returns this transformed value.

Related information

- "stdlib.h" on page 70
- "drand48() — Pseudo-random number generator" on page 402
- "erand48() — Pseudo-random number generator" on page 426
- "jrand48() — Pseudo-random number generator" on page 928
- "lcong48() — Pseudo-random number initializer" on page 939
- "lrand48() — Pseudo-random number generator" on page 1010
- "mrand48() — Pseudo-random number generator" on page 1101
- "seed48() — Pseudo-random number initializer" on page 1467
- "srand48() — Pseudo-random number initializer" on page 1718

ntohl() — Translate a long integer into host byte order

Standards

Standards / Extensions	C or C++	Dependencies
XPG4.2 Single UNIX Specification, Version 3	both	

Format

XPG4.2:

```
#define _XOPEN_SOURCE_EXTENDED 1
#include <arpa/inet.h>

in_addr_t ntohl(in_addr_t netlong);
```

SUSV3:

```
#define _POSIX_C_SOURCE 200112L
#include <arpa/inet.h>

unit32_t ntohl(uint32_t netlong);
```

Berkeley sockets:

```
#define _OE_SOCKETS
#include <sys/types.h>
#include <netinet/in.h>
#include <arpa/inet.h>

unsigned long ntohl(unsigned long a);
```

General description

The ntohl() function translates a long integer from network byte order to host byte order.

Parameter
Description

a The unsigned long integer to be put into host byte order.

in_addr_t *netlong*
Is typed to the unsigned long integer to be put into host byte order.

Notes:

1. For MVS, host byte order and network byte order are the same.
2. Since this function is implemented as a macro, you need one of the feature test macros and the inet header file.

Returned value

ntohl() returns the translated long integer.

Related information

- "arpa/inet.h" on page 17
- "netinet/in.h" on page 53
- "sys/types.h" on page 75
- "htonl() — Translate address host to network long" on page 818
- "htons() — Translate an unsigned short integer into network byte order" on page 820
- "ntohs() — Translate an unsigned short integer into host byte order"

ntohs() — Translate an unsigned short integer into host byte order

Standards

Standards / Extensions	C or C++	Dependencies
XPG4.2 Single UNIX Specification, Version 3	both	

Format

XPG4.2:

```
#define _XOPEN_SOURCE_EXTENDED 1
#include <arpa/inet.h>

in_port_t ntohs(in_port_t netshort);
```

SUSV3:

```
#define _POSIX_C_SOURCE 200112L
#include <arpa/inet.h>

uint16_t ntohs(uint16_t netshort);
```

Berkeley sockets:

```
#define _OE_SOCKETS
#include <sys/types.h>
#include <arpa/inet.h>
#include <netinet/in.h>

unsigned short ntohs(unsigned short a);
```

General description

The ntohs() function translates a short integer from network byte order to host byte order.

Parameter
Description

a The unsigned short integer to be put into host byte order.

in_port_t *netshort*
Is typed to the unsigned short integer to be put into host byte order.

Notes:

1. For MVS, host byte order and network byte order are the same.
2. Since this function is implemented as a macro, you need one of the feature test macros and the inet header file.

Returned value

ntohs() returns the translated short integer.

Related information

- "arpa/inet.h" on page 17
- "netinet/in.h" on page 53
- "sys/types.h" on page 75
- "htonl() — Translate address host to network long" on page 818
- "htons() — Translate an unsigned short integer into network byte order" on page 820
- "ntohl() — Translate a long integer into host byte order" on page 1150

open() — Open a file

Standards

Standards / Extensions	C or C++	Dependencies
POSIX.1 XPG4 XPG4.2 Single UNIX Specification, Version 3	both	

Format

```
#define _POSIX_SOURCE
#include <fcntl.h>

int open(const char *pathname, int options, ...);
```

General description

Opens a file and returns a number called a *file descriptor*.

The *pathname* argument must be a z/OS UNIX file file name. You can use this file descriptor to refer to the file in subsequent I/O operations, for example, read() or write(). Each file opened by a process gets a new file descriptor.

Restriction: Using this function with FIFOs, POSIX terminals, and character special files requires z/OS XL C programs running POSIX(ON).

The argument *pathname* is a string giving the name of the file you want to open. The integer *options* specifies options for the open operation by taking the bitwise inclusive-OR of symbols defined in the fcntl.h header file. The options indicate whether the file should be accessed for reading, writing, reading and writing, and so on.

An additional argument (`...`) is required if the O_CREAT option is specified in *options*. This argument may be called the *mode* and has the `mode_t` type. It specifies file permission bits to be used when a file is created. All the file permission bits are set to the bits of *mode*, except for those set in the file-mode creation mask of the process. Here is a list of symbols that can be used for a mode.

S_IRGRP
Read permission for the file's group.

S_IROTH
Read permission for users other than the file owner.

S_IRUSR
Read permission for the file owner.

S_IRWXG
Read, write, and search or execute permission for the file's group. S_IRWXG is the bitwise inclusive-OR of S_IRGRP, S_IWGRP, and S_IXGRP.

S_IRWXO
Read, write, and search or execute permission for users other than the file owner. S_IRWXO is the bitwise inclusive-OR of S_IROTH, S_IWOTH, and S_IXOTH.

S_IRWXU
Read, write, and search, or execute, for the file owner; S_IRWXG is the bitwise inclusive-OR of S_IRUSR, S_IWUSR, and S_IXUSR.

S_ISGID
Privilege to set group ID (GID) for execution. When this file is run through an exec function, the effective group ID of the process is set to the group ID of the file, so that the process has the same authority as the file owner rather than the authority of the actual invoker.

S_ISUID
Privilege to set the user ID (UID) for execution. When this file is run through an exec function, the effective user ID of the process is set to the owner of the file, so that the process has the same authority as the file owner rather than the authority of the actual invoker.

S_ISVTX
Indicates shared text. Keep loaded as an executable file in storage.

S_IWGRP
Write permission for the file's group.

S_IWOTH
Write permission for users other than the file owner.

S_IWUSR
Write permission for the file owner.

S_IXGRP
Search permission (for a directory) or execute permission (for a file) for the file's group.

S_IXOTH
Search permission for a directory, or execute permission for a file, for users other than the file owner.

S_IXUSR
Search permission (for a directory) or execute permission (for a file) for the file owner.

Most open operations position a *file offset* (an indicator showing where the next read or write will take place in the file) at the beginning of the file; however, there are options that can change this position. One of the following *must* be specified in the *options* argument of the open() operation:

O_RDONLY
Open for reading only

O_WRONLY
Open for writing only

O_RDWR
Open for both reading and writing

One or more of the following can also be specified in *options*:

O_APPEND
Positions the file offset at the end of the file before each write operation.

O_CREAT
Indicates that the call to open() has a *mode* argument.

If the file being opened already exists O_CREAT has no effect except when O_EXCL is also specified; see O_EXCL following.

If the file being opened does not exist it is created. The user ID is set to the effective ID of the process, and its group ID is set to the group ID of its directory. File permission bits are set according to *mode*.

If O_CREAT is specified and the file did not previously exist a successful open() sets the access time, change time, and modification time for the file. It also updates the change time and modification time fields in the parent directory.

O_EXCL

If both O_EXCL and O_CREAT are specified open() fails if the file already exists. If both O_EXCL and O_CREAT are specified and *pathname* names a symbolic link open() fails regardless of the contents of the symbolic link.

The check for the existence of the file and the creation of the file if it does not exist is atomic with respect to other threads executing open() naming the same filename in the same directory with O_EXCL and O_CREAT set.

O_NOCTTY
If *pathname* specifies a terminal open() does not make the terminal the

controlling terminal of the process (and the session). If O_NOCTTY is not specified the terminal becomes the controlling terminal if the following conditions are true:

- The process is a session leader.
- There is no controlling terminal for the session.
- The terminal is not already a controlling terminal for another session.

O_NONBLOCK
Has different meanings depending on the situation.

- When you are opening a FIFO special file with O_RDONLY or O_WRONLY:

  If O_NONBLOCK is specified a read-only open() returns immediately. A write-only open() returns with an error if no other process has the FIFO open for reading.

  If O_NONBLOCK is not specified a read-only open() blocks until another process opens the FIFO for writing. A write-only open() blocks until another process opens the FIFO for reading.
- When you are opening a character special file that supports a nonblocking open(), O_NONBLOCK controls whether subsequent reads and writes can block.

O_TRUNC
If the file is successfully opened with O_RDWR or O_WRONLY, this will truncate the file to zero length if the file exists and is a regular file. The mode and owner of the file are unchanged. This option should not be used with O_RDONLY. O_TRUNC has no effect on FIFO special files or directories.

If O_TRUNC is specified and the file previously existed a successful open() updates the change time and modification time for the file.

O_SYNC
Force synchronous update. If this flag is 1 every write() operation on the file is written to permanent storage. That is, the file system buffers are forced to permanent storage. See fsync() also.

The program is assured that all data for the file has been written to permanent storage on return from a function which performs a synchronous update,

If *pathname* refers to a STREAM file, *oflag* may be constructed from `O_NONBLOCK` OR-ed with either `O_RDONLY`, `O_WRONLY` or `O_RDWR`. Other flag values are not applicable to STREAMS devices and have no effect on them. The value `O_NONBLOCK` affects the operation of STREAMS drivers and certain functions applied to file descriptors associated with STREAMS files. For STREAMS drivers, the implementation of O_NONBLOCK is device-specific.

Note: z/OS UNIX services do not supply any STREAMS devices or pseudodevices. It is impossible for open() to return a valid STREAMS file descriptor.

The largest value that can be represented correctly in an object of type off_t is established as the offset maximum in the open file description.

Large file support for z/OS UNIX files: Large z/OS UNIX files are supported automatically for AMODE 64 C/C++ applications. AMODE 31 C/C++ applications must be compiled with the option LANGLVL(LONGLONG) and define the

_LARGE_FILES feature test macro before any headers are included to enable this function to operate on z/OS UNIX files that are larger than 2 GB in size. File size and offset fields are enlarged to 63 bits in width. Therefore, any other function operating on the file is required to define the _LARGE_FILES feature test macro as well.

Returned value

If successful, open() returns a file descriptor.

If unsuccessful, open() returns -1 and sets errno to one of the following values:

Error Code
Description

EACCES
Access is denied. Possible reasons include:

- The process does not have search permission on a component in *pathname*.
- The file exists, but the process does not have permission to open the file in the way specified by the flags.
- The file does not exist, and the process does not have write permission on the directory where the file is to be created.
- O_TRUNC was specified, but the process does not have write permission on the file.

EBUSY
The process attempted to open a file that is in use.

EEXIST
O_CREAT and O_EXCL were specified, and either the named file refers to a symbolic link, or the named file already exists.

EINTR
open() was interrupted by a signal.

EINVAL
The *options* parameter does not specify a valid combination of the O_RDONLY, O_WRONLY and O_TRUNC bits.

EIO The *pathname* argument names a STREAMS file and a hang-up or error occurred during the open().

EISDIR
pathname is a directory, and *options* specifies write or read/write access.

ELOOP
A loop exists in symbolic links. This error is issued if the number of symbolic links detected in the resolution of *pathname* is greater than POSIX_SYMLOOP.

EMFILE
The process has reached the maximum number of file descriptors it can have open.

ENAMETOOLONG
pathname is longer than PATH_MAX characters, or some component of *pathname* is longer than NAME_MAX characters while _POSIX_NO_TRUNC is in effect. For symbolic links, the length of the

pathname string substituted for a symbolic link exceeds PATH_MAX. The PATH_MAX and NAME_MAX values can be determined using pathconf().

ENFILE
The system has reached the maximum number of file descriptors it can have open.

ENOENT
Typical causes:

- O_CREAT is not specified, and the named file does not exist.
- O_CREAT is specified, and either the prefix of *pathname* does not exist or the *pathname* argument is an empty string.

ENOMEM
The *pathname* argument names a STREAMS file and the system is unable to allocate resources.

ENOSPC
The directory or file system intended to hold a new file has insufficient space.

ENOSR
The *pathname* argument names a STREAMS-based file and the system is unable to allocate a STREAM.

ENOSYS
For master pseudoterminals, slave initialization did not complete.

ENOTDIR
A component of *pathname* is not a directory.

ENXIO
O_NONBLOCK and O_WRONLY were specified and the named file is a FIFO, but no process has the file open for reading. For a pseudoterminal, the requested minor number exceeds the maximum number supported by the installation.

EPERM
For slave pseudoterminals, permission to open is denied for one of these reasons:

- It is the first open of the slave after the master pseudoterminal was opened, and the user ID associated with the two opening processes is not the same.
- There was an internal error in the security system after the master pseudoterminal was opened.
- The attempt to open the slave used a different pathname than earlier opens used.

EROFS
pathname is on a read-only file system, and one or more of the options O_WRONLY, O_RDWR, O_TRUNC, or O_CREAT (if the file does not exist) was specified.

Example

The following opens an output file for appending:

```
int fd;
fd = open("outfile",O_WRONLY | O_APPEND);
```

The following statement creates a new file with read/write/execute permissions for the creating user. If the file already exists, open() fails.

```
fd = open("newfile",O_WRONLY|O_CREAT|O_EXCL,S_IRWXU);
```

Related information

- "fcntl.h" on page 27
- "close() — Close a file" on page 288
- "creat() — Create a new file or rewrite an existing one" on page 340
- "dup() — Duplicate an open file descriptor" on page 404
- "exec functions" on page 436
- "fcntl() — Control open file descriptors" on page 474
- "fsync() — Write changes to direct-access storage" on page 655
- "lseek() — Change the offset of a file" on page 1020
- "read() — Read from a file or socket" on page 1376
- "stat() — Get file information" on page 1720
- "umask() — Set and retrieve file creation mask" on page 1947
- "write() — Write data on a file or socket" on page 2085

opendir() — Open a directory

Standards

Standards / Extensions	C or C++	Dependencies
POSIX.1 XPG4 XPG4.2 Single UNIX Specification, Version 3	both	

Format

```
#define _POSIX_SOURCE
#include <dirent.h>

DIR *opendir(const char *dirname);
```

General description

Opens a directory so that it can be read with readdir() or __readdir2(). *dirname* is a string giving the name of the directory you want to open. The first readdir() or __readdir2() call reads the first entry in the directory.

Returned value

If successful, opendir() returns a pointer to a DIR object. This object describes the directory and is used in subsequent operations on the directory, in the same way that FILE objects are used in file I/O operations.

If unsuccessful, opendir() returns a NULL pointer and sets errno to one of the following values:

Error Code
Description

EACCES
The process does not have permission to search some component of *dirname*, or it does not have read permission on the directory itself.

ELOOP

A loop exists in the symbolic links. This error is issued if more than POSIX_SYMLOOP (defined in the limits.h header file) symbolic links are encountered during resolution of the *dirname* argument.

EMFILE

The process has too many other file descriptors already open.

ENAMETOOLONG

dirname is longer than **PATH_MAX** characters, or some component of *dirname* is longer than **NAME_MAX** characters while _POSIX_NO_TRUNC is in effect. For symbolic links, the length of the pathname string substituted for a symbolic link exceeds **PATH_MAX**. The **PATH_MAX** and **NAME_MAX** values can be determined using pathconf().

ENFILE

The entire system has too many other file descriptors already open.

ENOENT

The directory *dirname* does not exist.

ENOMEM

There is not enough storage available to open the directory.

ENOTDIR

Some component of the *dirname* pathname is not a directory.

Example

CELEBO01

```
/* CELEBO01

   This example opens a directory.

 */
#define _POSIX_SOURCE
#include <dirent.h>
#include <errno.h>
#include <sys/stat.h>
#include <sys/types.h>
#undef _POSIX_SOURCE
#include <stdio.h>

void traverse(char *fn, int indent) {
  DIR *dir;
  struct dirent *entry;
  int count;
  char path[1025];
  struct stat info;

  for (count=0; count<indent; count++) printf("  ");
  printf("%s\n", fn);

  if ((dir = opendir(fn)) == NULL)
    perror("opendir() error");
  else {
    while ((entry = readdir(dir)) != NULL) {
      if (entry->d_name[0] != '.') {
        strcpy(path, fn);
        strcat(path, "/");
        strcat(path, entry->d_name);
        if (stat(path, &info) != 0)
          fprintf(stderr, "stat() error on %s: %s\n", path,
                  strerror(errno));
        else if (S_ISDIR(info.st_mode))
               traverse(path, indent+1);
      }
    }
    closedir(dir);
  }
}

main() {
  puts("Directory structure:");
  traverse("/etc", 0);
}
```

Output

```
Directory structure:
/etc
  /etc/samples
    /etc/samples/IBM
  /etc/IBM
```

Related information

- "dirent.h" on page 22
- "stdio.h" on page 68
- "sys/types.h" on page 75
- "closedir() — Close a directory" on page 291
- "__opendir2() — Open a directory"
- "readdir() — Read an entry from a directory" on page 1382
- "rewinddir() — Reposition a directory stream to the beginning" on page 1446
- "seekdir() — Set position of directory stream" on page 1469
- "telldir() — Current location of directory stream" on page 1866

__opendir2() — Open a directory

Standards

Standards / Extensions	C or C++	Dependencies
z/OS UNIX	both	OS/390 V2R6

Format

```
#define _OPEN_SYS_DIR_EXT
#include <dirent.h>

DIR *__opendir2(const char *dirname, size_t bufsize);
```

General description

Opens a directory so that it can be read with readdir() or __readdir2(). The first readdir() or __readdir2() call reads the first entry in the directory.

dirname is a string giving the name of the directory you want to open. *bufsize* is the size (in bytes) of the internal work buffer used by readdir() or __readdir2() to hold directory entries. The larger the buffer, the less overhead there will be when reading through large numbers of directory entries. This buffer will exist until the directory is closed. If the specified buffer size is too small, it is ignored. A minimum-size buffer is used instead.

__opendir2() is the same as opendir(), except that the buffer size can be specified as a parameter.

Returned value

If successful, __opendir2() returns a pointer to a DIR object. This object describes the directory and is used in subsequent operations on the directory, in the same way that FILE objects are used in file I/O operations.

If unsuccessful, __opendir2() returns a NULL pointer and sets errno to one of the following values:

Error Code
Description

EACCES
The process does not have permission to search some component of *dirname*, or it does not have read permission on the directory itself.

ELOOP
A loop exists in the symbolic links. This error is issued if more than POSIX_SYMLOOP (defined in the limits.h header file) symbolic links are encountered during resolution of the *dirname* argument.

EMFILE
The process has too many other file descriptors already open.

ENAMETOOLONG
dirname is longer than **PATH_MAX** characters, or some component of *dirname* is longer than **NAME_MAX** characters while _POSIX_NO_TRUNC is in effect. For symbolic links, the length of the pathname string substituted for a symbolic link exceeds **PATH_MAX**. The **PATH_MAX** and **NAME_MAX** values can be determined using pathconf().

ENFILE
The entire system has too many other file descriptors already open.

ENOENT
The directory *dirname* does not exist.

ENOMEM
There is not enough storage available to open the directory using a buffer that is length *bufsize* bytes long.

ENOTDIR
Some component of the *dirname* pathname is not a directory.

Related information

- “dirent.h” on page 22
- “stdio.h” on page 68
- “sys/types.h” on page 75
- “closedir() — Close a directory” on page 291
- “opendir() — Open a directory” on page 1158
- “readdir() — Read an entry from a directory” on page 1382
- “rewinddir() — Reposition a directory stream to the beginning” on page 1446
- “seekdir() — Set position of directory stream” on page 1469
- “telldir() — Current location of directory stream” on page 1866

openlog() — Open the system control log

Standards

Standards / Extensions	C or C++	Dependencies
XPG4.2 Single UNIX Specification, Version 3	both	

Format

```
#define _XOPEN_SOURCE_EXTENDED 1
#include <syslog.h>

void openlog(const char *ident, int logopt, int facility);
```

General description

The openlog() function optionally opens a connection to the logging facility, and sets process attributes that affect subsequent calls to the syslog() function. The argument *ident* is a string that is prefixed to every message. *logopt* is a bit field indicating logging options. Current values of *logopt* are:

LOG_CONS
Write messages to the system console if they cannot be sent to the logging facility. This option is safe to use in processes that have no controlling terminal, since the syslog() function forks before opening the console.

LOG_NDELAY
Open the connection to the logging facility immediately. Normally the open is delayed until the first message is logged. This is useful for programs that need to manage the order in which file descriptors are allocated..

LOG_NOWAIT
Do not wait for child processes that have been forked to log messages onto the console. This option should be used by processes that enable notification of child termination using SIGCHLD, since the syslog() function may otherwise block waiting for a child whose exit status has already been collected.

LOG_ODELAY
Delay open until syslog() is called.

LOG_PID
Log the processID with each message. This is useful for identifying specific processes. In the message header, the processID is surrounded by square brackets. The code point values for the square brackets are taken from code page IBM-1047. The value for the left square bracket is 0xAD. The value for the right square bracket is 0xBD.

The *facility* argument encodes a default facility to be assigned to all messages that do not have an explicit facility already encoded. The initial default facility is as follows:

LOG_USER
Message generated by random processes. This is the default facility identifier if none is specified.

Returned value

openlog() returns no values.

No errors are defined.

Related information

- “syslog.h” on page 72
- “closelog() — Close the control log” on page 292
- “setlogmask() — Set the mask for the control log” on page 1561
- “syslog() — Send a message to the control log” on page 1803

__open_stat() — Open a file and get file status information

Standards

Standards / Extensions	C or C++	Dependencies
z/OS UNIX	both	OS/390 V2R6

Format

```
#define _POSIX_SOURCE
#include <fcntl.h>
#include <sys/stat.h>

int __open_stat(const char *pathname, int options, mode_t mode,
                struct stat *info);
```

General description

Opens a file and returns a number called a *file descriptor*. __open_stat() also returns information about the opened file. __open_stat() is a combination of open() and fstat().

The parameters are:

Parameters
Description

pathname
This parameter is a NULL-terminated character string containing the hierarchical file system (HFS) pathname of the file to be opened.

pathname can begin with or without a slash.

- A pathname beginning with a slash is an absolute pathname. The slash refers to the root directory, and the search for the file starts at the root directory.
- A pathname not beginning with a slash is a relative pathname. The search for the file begins at the working directory.

See “open() — Open a file” on page 1152 for more information about the *pathname* parameter and the types of files that can be opened.

options An integer containing option bits for the open operation. These options are the same as those in the *options* parameter passed to open(). These bits are defined in `fcntl.h`. For a list of these option bits and their meaning, see “open() — Open a file” on page 1152.

mode *mode* is the same as the optional third parameter for open(), which is used when a new file is being created. For __open_stat(), the *mode* parameter is always required. If a new file is not being created, *mode* is ignored, and may be set to 0. When __open_stat() creates a file, the flag bits in *mode* specify the file permissions and other characteristics for the new file. The flag bits in *mode* are defined in `sys/modes.h`. For more information about the *mode* parameter, see "open() — Open a file" on page 1152.

info The *info* parameter points to an area of memory where the system will store information about the file that is opened. This parameter is the same as the *info* parameter in fstat() or stat(). If the file is successfully opened, the system returns file status information in a `stat` structure, as defined in `sys/stat.h`. The elements of this structure are described in "stat() — Get file information" on page 1720.

Returned value

If successful, __open_stat() returns a file descriptor.

If unsuccessful, __open_stat() returns -1 and sets errno to one of the following values:

Error Code
Description

EACCESS
Access to the file was denied. One of the following errors occurred:

- The calling process does not have permission to search one of the directories specified in the *pathname* parameter.
- The calling process does not have permission to open the file in the way specified by the *options* parameter.
- The file does not exist, and the calling process does not have permission to write into files in the directory the file would have been created in.
- The truncate option was specified, but the process does not have write permission for the file.

EAGAIN
Resources were temporarily unavailable.

EBUSY
pathname specifies a master pseudoterminal that is either already in use or for which the corresponding slave is open.

EEXIST
The exclusive create option was specified, but the file already exists.

Use __errno2() to determine the exact reason the error occurred.

EFBIG
A request to create a new file is prohibited because the file size limit for the process is set to 0.

EINTR
The __open_stat() operation was interrupted by a signal.

EINVAL
The *options* parameter does not specify a valid combination of the `O_RDONLY`, `O_WRONLY` and `O_TRUNC` bits, or the file type specified in the *mode* parameter is not valid.

Use __errno2() to determine the exact reason the error occurred.

EISDIR
The file specified by *pathname* is a directory and the *options* parameter specifies write or read/write access.

Use __errno2() to determine the exact reason the error occurred.

ELOOP
A loop exists in symbolic links encountered during resolution of the *pathname* parameter. This error is issued if more than 8 symbolic links are detected in the resolution of *pathname*.

EMFILE
The process has reached the maximum number of file descriptors it can have open.

ENAMETOOLONG
pathname is longer than 1023 characters, or a component of *pathname* is longer than 255 characters. (The system does not support filename truncation.)

ENODEV
Typical causes of this error are:

- An attempt was made to open a character special file for a device not supported by the system.
- An attempt was made to open a character special file for a device that is not yet initialized.

Use __errno2() to determine the exact reason the error occurred.

ENOENT
Typical causes of this error are:

- The request did not specify that the file was to be created, but the file named by *pathname* was not found.
- The request asked for the file to be created, but some component of *pathname* was not found, or the *pathname* parameter was blank.

Use __errno2() to determine the exact reason the error occurred.

ENOSPC
The directory or file system intended to hold a new file has insufficient space.

ENOTDIR
A component of *pathname* is not a directory.

ENXIO
The __open_stat() request specified write-only and nonblock for a FIFO special file, but no process has the file open for reading. For pseudoterminals, this errno can mean that the minor number associated with *pathname* is too big.

EPERM
The caller is not permitted to open the specified slave pseudoterminal or the corresponding master is not yet open. `EPERM` is also returned if the slave is closed with `HUPCL` set and an attempt is made to reopen it.

EROFS
The *pathname* parameter names a file on a read-only file system, but options that would allow the file to be altered were specified: write-only, read/write, truncate, or -- for a new file -- create.

Use __errno2() to determine the exact reason the error occurred.

Related information

- "fcntl.h" on page 27
- "sys/stat.h" on page 75
- "close() — Close a file" on page 288
- "creat() — Create a new file or rewrite an existing one" on page 340
- "fstat() — Get status information about a file" on page 651
- "stat() — Get file information" on page 1720
- "umask() — Set and retrieve file creation mask" on page 1947
- "open() — Open a file" on page 1152

__osname() — Get true operating system name

Standards

Standards / Extensions	C or C++	Dependencies
Language Environment	both	OS/390 V2R10

Format

```
#define _POSIX_SOURCE
#include <sys/utsname.h>

int __osname(struct utsname *name);
```

General description

The __osname() function retrieves information identifying the true operating system you are running on. The argument *name* points to a memory area where a structure describing the true operating system the process is running on can be stored.

The information about the true operating system is returned in a `utsname` structure, which has the following elements:

char *sysname;
The true name of the implementation of the operating system.

char *nodename;
The node name of this particular machine. The node name is set by the SYSNAME sysparm (specified at IPL), and usually differentiates machines running at a single location.

char *release;
The true current release level of the implementation.

char *version;
The true current version level of the release.

char *machine;
The name of the hardware type the system is running on.

Each of the `utsname` structure elements is a normal C string, terminated with a NULL character.

The values returned by the __osname() function are not intended to be used for comparison purposes in order to determine a level of functionality provided by the operating system. This is because the version and release values are not guaranteed to be equal to or greater than the previous implementation.

Table 44 lists the true operating system information returned by the __osname() function.

Table 44. Operating system information returned by the __osname() fuction

Operating system	Sysname	Release	Version
z/OS V2.2	z/OS	02.00	02
z/OS V2.1	z/OS	01.00	02
z/OS V1.13	z/OS	13.00	01
z/OS V1.12	z/OS	12.00	01
z/OS V1.11	z/OS	11.00	01
z/OS V1.10	z/OS	10.00	01
z/OS V1.9	z/OS	09.00	01
z/OS V1.8	z/OS	08.00	01
z/OS V1.7	z/OS	07.00	01
z/OS V1.6	z/OS	06.00	01
z/OS V1.5	z/OS	05.00	01
z/OS V1.4	z/OS	04.00	01
z/OS V1.3	z/OS	03.00	01
z/OS V1.2	z/OS	02.00	01
z/OS V1.1	z/OS	01.00	01
OS/390 V2.10	OS/390	10.00	02

Returned value

If successful, the __osname() function returns a nonnegative value.

If unsuccessful, the __osname() function returns -1 and an errno might be set to indicate the reason for the failure.

Example

CELEBO02

```
/*
   This example gets information about the system you are running on.
 */
#define _POSIX_SOURCE
#include <sys/utsname.h>
#include <stdio.h>
main() {
  struct utsname uts;
  if (__osname(&uts) < 0)
    perror("__osname() error");
  else {
    printf("Sysname:  %s\n", uts.sysname);
    printf("Nodename: %s\n", uts.nodename);
    printf("Release:  %s\n", uts.release);
    printf("Version:  %s\n", uts.version);
    printf("Machine:  %s\n", uts.machine);
  }
}
```

Output

```
Sysname:  z/OS
Nodename: SY1
Release:  02.00
Version:  02
Machine:  2097
```

Related information

- “sys/utsname.h” on page 77
- “uname() — Display current operating system name” on page 1951

__passwd(), __passwd_applid() — Verify or change user password

Standards

Standards / Extensions	C or C++	Dependencies
z/OS UNIX	both	

Format

```
#include <pwd.h>

int __passwd(const char *username, const char *oldpass, const char *newpass);
int __passwd_applid(const char *username, const char *oldpass,
                    const char *newpass, const char *applid);
```

General description

The __passwd() function verifies or changes the *username* password or the password phrase. The *username* is a NULL-terminated character string of 1 to 8 bytes. The *oldpass* is the current password or password phrase for user *username*, and is a NULL-terminated character string of a password or a password phrase. When *newpass* is NULL, then *oldpass* represents the password or password phrase to be verified, and no password or password phrase change is performed. Otherwise, *newpass* is a NULL-terminated character string of a password or a password phrase. Other installation-dependent restrictions on passwords or password phrases may apply, both in terms of length and content. Length restrictions may be imposed by the security product.

The __passwd_applid() function is equivalent to __passwd() with the added feature that it also allows the application identifier (APPLID) to be supplied that will be passed on to the security product to assist with authentication. When applid is NULL or a pointer to NULL, no application identifier will be passed on to the security product.

The function has the following parameters:

Parameter
Description

applid Specifies the application identifier that will be used for authentication with the security product.

If the BPX.DAEMON facility class profile is defined, then all modules within the address space must be loaded from a controlled library. This includes all modules in the application and runtime libraries. See also "Checking Which Module is not Defined to Program Control" in *z/OS UNIX System Services Planning, GA32-0884*.

Returned value

If successful, __passwd() returns 0. When *newpass* is NULL, the password or the password phrase has been verified. When *newpass* is not NULL, the new password or password phrase has been set.

If a user specifies password for *oldpass* and new password phrase for *newpass* or password phrase for *oldpass* and new password for *newpass*, then __passwd() returns -1 and sets errno to EMVSPASSWORD and the current password or password phrase is not changed.

If unsuccessful, __passwd() returns -1 and sets errno to one of the following values:

Error Code
Description

EACCES
The *oldpass* is not authorized.

EINVAL
The *username*, *oldpass*, *newpass*, or *applid* argument is invalid.

EMVSERR
The specified function is not supported in an address space where a load was done from an uncontrolled library.

EMVSEXPIRE
The *oldpass* has expired and no *newpass* has been provided.

EMVSPASSWORD
The *newpass* is not valid, or does not meet the installation-exit requirements.

EMVSSAF2ERR
Internal processing error.

EMVSSAFEXTRERR
An internal SAF/RACF extract error has occurred. A possible reason is that the *username* access has been revoked. errno2 contains the BPX1PWD reason code. For more information, see *z/OS UNIX System Services Programming: Assembler Callable Services Reference*.

ESRCH
The *username* provided is not defined to the security product or does not have an OMVS segment defined.

For more information, refer to the *z/OS UNIX System Services Messages and Codes* , *z/OS UNIX System Services Programming: Assembler Callable Services Reference*

Related information

- “pwd.h” on page 61
- “endpwent() — User database functions” on page 423
- “getpass() — Read a string of characters without echo” on page 748

pathconf() — Determine configurable path name variables

Standards

Standards / Extensions	C or C++	Dependencies
POSIX.1a XPG4 XPG4.2 Single UNIX Specification, Version 3	both	

Format

```
#define _POSIX_SOURCE
#include <unistd.h>

long pathconf(const char *pathname, int varcode);
```

General description

Lets an application determine the value of a configuration variable, *varcode*, associated with a particular file or directory, *pathname*.

The *varcode* argument may be any one of the following symbols, defined in the unistd.h header file, each standing for a configuration variable:

_PC_LINK_MAX
Represents LINK_MAX, the maximum number of links the file can have. If *pathname* is a directory, pathconf() returns the maximum number of links that can be established to the directory itself.

_PC_MAX_CANON
Represents MAX_CANON, the maximum number of bytes in a terminal canonical input line. *pathname* must refer to a character special file for a terminal.

_PC_MAX_INPUT
Represents MAX_INPUT, the maximum number of bytes for which space is available in a terminal input queue. That is, it refers to the maximum number of bytes that a portable application can have the user enter before the application actually reads the input. *pathname* must refer to a character special file for a terminal.

_PC_NAME_MAX
Represents NAME_MAX, the maximum number of characters in a file name (not including any terminating NULL at the end if the file name is stored as a string). This symbol refers only to the file name itself, that is, the last component of the file's path name. pathconf() returns the maximum length of file names.

_PC_PATH_MAX
Represents PATH_MAX, the maximum number of characters in a complete path name (not including any terminating NULL at the end if the path name is stored as a string). pathconf() returns the maximum length of a relative path name.

_PC_PIPE_BUF
Represents PIPE_BUF, the maximum number of bytes that can be written "atomically" to a pipe. If more than this number of bytes is written to a pipe, the operation may take more than one physical write operation and physical read operation to read the data on the other end of the pipe. If *pathname* is a FIFO special file, pathconf() returns the value for the file itself. If *pathname* is a directory, pathconf() returns the value for any FIFOs that exist or that can be created under the directory. If *pathname* is any other kind of file, an errno of EINVAL will be returned, indicating an invalid path name was specified.

_PC_CHOWN_RESTRICTED
Represents _POSIX_CHOWN_RESTRICTED defined in the unistd.h header file, and restricts use of chown() to a process with appropriate privileges. It also changes the group ID of a file to the effective group ID of the process

or to one of its supplementary group IDs. If *pathname* is a directory, pathconf() returns the value for any kind of file under the directory, but not for subdirectories of the directory.

_PC_NO_TRUNC
Represents _POSIX_NO_TRUNC defined in the unistd.h header file, and generates an error if a file name is longer than NAME_MAX. If *pathname* refers to a directory, the value returned by pathconf() applies to all files under that directory.

_PC_VDISABLE
Represents _POSIX_VDISABLE defined in the unistd.h header file. This symbol indicates that terminal special characters can be disabled using this character value, if it is defined; see the callable service tcsetattr() for details. *pathname* must refer to a character special file for a terminal.

_PC_ACL
Returns 1 if an access control mechanism is supported by the security product.

_PC_ACL_ENTRIES_MAX
Returns the maximum number of ACL entries in an ACL for a file or directory that supports ACLs.

Returned value

If successful, pathconf() return the value of the variable requested in *varcode*.

If unsuccessful, pathconf() returns -1. If a particular variable has no limit, such as PATH_MAX, pathconf() returns -1 but does not change errno.

If pathconf() cannot determine an appropriate value, it sets errno to one of the following values:

Error Code
Description

EACCES
The process does not have search permission on some component of the *pathname*.

EINVAL
varcode is not a valid variable code, or the given variable cannot be associated with the specified file.

- If *varcode* refers to MAX_CANON, MAX_INPUT, or _POSIX_VDISABLE, and *pathname* does not refer to a character special file, pathconf() returns -1 and sets errno to EINVAL.
- If *varcode* refers to NAME_MAX, PATH_MAX, or POSIX_NO_TRUNC, and *pathname* does not refer to a directory, pathconf() returns the requested information.
- If *varcode* refers to PC_PIPE_BUF and *pathname* refers to a pipe or a FIFO, the value returned applies to the referenced object itself. If *pathname* refers to a directory, the value returned applies to any FIFOs that exist or can be created within the directory. If *pathname* refers to any other type of file, the function sets errno to EINVAL.

ELOOP
A loop exists in symbolic links. This error is issued if more than POSIX_SYMLOOPS symbolic links are detected in the resolution of *pathname*.

ENAMETOOLONG
pathname is longer than PATH_MAX characters, or some component of *pathname* is longer than NAME_MAX while _POSIX_NO_TRUNC is in effect.

For symbolic links, the length of the path name string substituted for a symbolic link exceeds PATH_MAX.

ENOENT
There is no file named *pathname*, or the *pathname* argument is an empty string.

ENOTDIR
Some component of the *pathname* is not a directory.

Example

CELEBP01

```
/* CELEBP01

   This example determines the maximum number of characters in a file name.

 */
#define _POSIX_SOURCE
#include <errno.h>
#include <unistd.h>
#undef _POSIX_SOURCE
#include <stdio.h>

main() {
  long result;

  errno = 0;
  puts("examining NAME_MAX limit for root filesystem");
  if ((result = pathconf("/", _PC_NAME_MAX)) == -1)
    if (errno == 0)
      puts("There is no limit to NAME_MAX.");
    else perror("pathconf() error");
  else
    printf("NAME_MAX is %ld\n", result);
}
```

Output:

```
examining NAME_MAX limit for root file system
NAME_MAX is 255
```

Related information

- “unistd.h” on page 82
- “fpathconf() — Determine configurable path name variables” on page 581

pause() — Suspend a process pending a signal

Standards

Standards / Extensions	C or C++	Dependencies
POSIX.1 POSIX.4a XPG4 XPG4.2 Single UNIX Specification, Version 3	both	POSIX(ON)

Format

```
#define _POSIX_SOURCE
#include <unistd.h>

int pause(void);
```

General description

Suspends execution of the calling thread. The thread does not resume execution until a signal is delivered, executing a signal handler or ending the thread. Some signals can be blocked by the process's *thread*. See “sigprocmask() — Examine or change a thread” on page 1651 for details.

If an incoming unblocked signal ends the thread, pause() never returns to the caller. If an incoming signal is handled by a signal handler, pause() returns after the signal handler returns.

Returned value

If pause() returns, it always returns -1 and sets errno to EINTR, indicating that a signal was received and handled successfully.

Example

CELEBP02

```
/* CELEBP02

   This example suspends execution and determines the
   current time.

 */
#define _POSIX_SOURCE
#include <unistd.h>
#include <signal.h>
#include <stdio.h>
#include <time.h>

void catcher(int signum) {
  puts("inside catcher...");
}

void timestamp() {
  time_t t;
  time(&t);
  printf("the time is %s", ctime(&t));
}

main() {
  struct sigaction sigact;

  sigemptyset(&sigact.sa_mask);
  sigact.sa_flags = 0;
  sigact.sa_handler = catcher;
  sigaction(SIGALRM, &sigact, NULL);

  alarm(10);
  printf("before pause... ");
  timestamp();
  pause();
  printf("after pause... ");
  timestamp();
}
```

Output:

```
before pause... the time is Fri Jun 16 09:42:29 2006
inside catcher...

after pause... the time is Fri Jun 16 09:42:39 2006
```

Related information

- "unistd.h" on page 82
- "alarm() — Set an alarm" on page 156
- "kill() — Send a signal to a process" on page 931
- "raise() — Raise signal" on page 1371
- "sigaction() — Examine or change a signal action" on page 1611
- "signal() — Handle interrupts" on page 1643
- "sigprocmask() — Examine or change a thread" on page 1651
- "sigsuspend() — Change mask and suspend the thread" on page 1663
- "wait() — Wait for a child process to end" on page 1992

pclose() — Close a pipe stream to or from a process

Standards

Standards / Extensions	C or C++	Dependencies
XPG4 XPG4.2 Single UNIX Specification, Version 3	both	POSIX(ON)

Format

```
#define  _XOPEN_SOURCE
#include <stdio.h>

int pclose(FILE *stream);
```

General description

The pclose() function closes a stream that was opened by popen(), waits for the command specified as an argument in popen() to terminate, and returns the status of the process that was running the shell command. However, if a call caused the termination status to be unavailable to pclose(), then pclose() returns -1 with errno set to ECHILD to report this situation; this can happen if the application calls one of the following functions:

- wait()
- waitid()

- waitpid() with a *pid* argument less than or equal to the process ID of the shell command
- any other function that could do one of the above

In any case, pclose() will not return before the child process created by popen() has terminated.

If the shell command cannot be executed, the child termination status returned by pclose() will be as if the shell command terminated using *exit(127)* or *_exit(127)*.

The pclose() function will not affect the termination status of any child of the calling process other than the one created by popen() for the associated stream.

If the argument *stream* to pclose() is not a pointer to a stream created by popen(), the termination status returned will be -1.

Threading Behavior: The pclose() function can be executed from any thread within the parent process.

Returned value

If successful, pclose() returns the termination status of the shell command.

If unsuccessful, pclose() returns -1 and sets errno to one of the following values:

Error Code
Description

ECHILD
The status of the child process could not be obtained.

Related information

- “stdio.h” on page 68
- “popen() — Initiate a pipe stream to or from a process” on page 1189
- “waitpid() — Wait for a specific child process to end” on page 1997

perror() — Print error message

Standards

Standards / Extensions	C or C++	Dependencies
ISO C POSIX.1 XPG4 XPG4.2 C99 Single UNIX Specification, Version 3 Language Environment	both	

Format

```
#include <stdio.h>

void perror(const char *string);
```

```
#define _OPEN_SYS_UNLOCKED_EXT 1
#include <stdio.h>

void perror_unlocked(const char *string);
```

X/Open:

```
#define _XOPEN_SOURCE_EXTENDED 1
#include <stdio.h>

void perror(const char *string);
```

Berkeley sockets:

```
#define _OE_SOCKETS
#include <stdio.h>

void perror(const char *string);
```

General description

Prints an error message to `stderr`. If *string* is not NULL and it does not point to a NULL character, the *string* pointed to by string is printed to the standard error stream, followed by a colon and a space. The message associated with the value in errno is then printed followed with the errno2 value in parenthesis and a newline character. The content of the message is the same as the content of a string returned by strerror() with the argument errno.

The perror() string shown as: EDC5121I Invalid argument. (errno2=0x0C0F8402).

To produce accurate results, you should ensure that perror() is called immediately after a library function returns with an error; otherwise, subsequent calls may alter the errno value.

If the error is associated with the `stderr` file, a call to perror() is not valid.

There is an environment variable `_EDC_ADD_ERRNO2`, which when set to 0, will remove the append of the current errno2 value at the end of the perror() string shown.

The perror() function will not change the orientation of the stderr stream.

perror_unlocked() is functionally equivalent to perror() with the exception that it is not thread-safe. This function can safely be used in a multithreaded application if and only if it is called while the invoking thread owns the (FILE*) object, as is the case after a successful call to either the flockfile() or ftrylockfile() function.

Returned value

perror() returns no values.

Example

CELEBP03

```
/* CELEBP03

   This example tries to open a stream.
   If the fopen() function fails, the example prints a message and ends
   the program.
```

```
 */
#include <stdio.h>
#include <stdlib.h>

int main(void)
{
   FILE *fh;

   if ((fh = fopen("myfile.dat","r")) == NULL)
   {
      perror("Could not open data file");
      abort();
   }
}
```

The following example tries to open a stream socket. If the socket fails, the example prints a message and ends the program.

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/socket>

int main(void)
{
   ins s;

   if ((s = socket (AF_INET,SOCK_STREAM,0)) <0)
   {
      perror("Could not open socket");
      exit(-1);
   }
}
```

Related information

- “stdio.h” on page 68
- “strerror() — Get pointer to runtime error message” on page 1737

__pid_affinity() — Add or delete process affinity

Standards

Standards / Extensions	C or C++	Dependencies
z/OS UNIX	both	OS/390 V2R6

Format

```
#define  _OPEN_SYS
#include <unistd.h>

int __pid_affinity(int   function_code,
                   pid_t target_pid,
                   pid_t signal_pid,
                   int   signal);
```

General description

The __pid_affinity() function adds or deletes an entry in a process's affinity list. When a process terminates, each process in its affinity list is notified (sent a signal) of the termination. The __pid_affinity() function provides the ability to dynamically create or break an association between two processes that is similar to the notification mechanism between parent and child processes without the processes being related.

The *function_code* can be set to one of the following symbolics, as defined in the unistd.h header file:

__PAF_ADD_PID
Add the process and signal specified by *signal_pid* and *signal* to the affinity list of the process specified by *target_pid*.

__PAF_DELETE_PID
Delete the process and signal specified by *signal_pid* and *signal* from the affinity list of the process specified by *target_pid*.

The *target_pid* identifies the process whose affinity list will be altered.

The *signal_pid* identifies the process that upon termination of the *target_pid* will be sent *signal* signal.

The *signal* identifies the signal that the *signal_pid* process will receive when the *target_pid* process terminates.

Usage notes

1. Either the *Target_Pid* or *Signal_Pid* must contain the PID of the caller's process.
2. The __pid affinity service is limited to adding and deleting entries in the caller's affinity list, or adding and deleting entries that contain the caller's PID (*Signal_Pid*) in other processes affinity list.
3. When the PAF_DELETE_PID# function is specified the *Signal* is ignored. It is not validated and may contain any value.
4. An entry is only deleted (PAF_DELETE_PID# specified) when the *Signal_Pid* matches an entry in the *Target_Pid* process's affinity list.
5. Entries with duplicate PIDs are not allowed in an affinity list. If adding an entry (PAF_ADD_PID# specified) and an entry with a PID that matches the *Signal_Pid* is found the entry is reused. This may result in the loss of a specific signal.
6. No permission is required when adding the caller's PID to another process's affinity list. All processes have permission to send a signal to themselves (raise()).
7. The PIDs specified by the *Target_Pid* and *Signal_Pid* parameters must be greater than 1. Specifying a PID equal to or less than 1 will result in a error.

Returned value

If successful, __pid_affinity() returns 0.

If unsuccessful, __pid_affinity() returns -1 and sets errno to one of the following values:

Error Code
Description

EINVAL
One or more of the following conditions were detected:
- The value specified by *Function_code* is not supported.
- The value specified by *Signal* is not a supported signal.
- *Target_Pid* does not contain a value greater than 1.
- *Signal_Pid* does not contain a value greater than 1.
- The *Signal_Pid* or *Target_Pid* does not specify the caller PID.

EMVSERR
A MVS environmental or internal error has occurred.

EMVSSAF2ERR
An internal SAF/RACF error has occurred.

EPERM
The caller does not have permission to send the signal to the *Signal_Pid* process.

ESRCH
One or more of the following conditions were detected:

- No process corresponding to *Target_Pid* was found.
- No process corresponding to *Signal_Pid* was found.

Related information

- "unistd.h" on page 82
- "kill() — Send a signal to a process" on page 931

pipe() — Create an unnamed pipe

Standards

Standards / Extensions	C or C++	Dependencies
POSIX.1 XPG4 XPG4.2 Single UNIX Specification, Version 3	both	

Format

```
#define _POSIX_SOURCE
#include <unistd.h>

int pipe(int fdinfo[2]);
```

General description

Creates a *pipe*, an I/O channel that a process can use to communicate with another process (in the same process or another process), or in some cases with itself. Data is written into one end of the pipe and read from the other. *fdinfo*[2] points to a memory area where pipe() can store two file descriptors. pipe() stores a file descriptor for the input end of the pipe in *fdinfo*[1], and stores a file descriptor for the output end of the pipe in *fdinfo*[0]. Thus, processes can read from *fdinfo*[0] and write to *fdinfo*[1]. Data written to *fdinfo*[1] is read from *fdinfo*[0] on a first-in-first-out (FIFO) basis.

When pipe() creates a pipe, the O_NONBLOCK and FD_CLOEXEC flags are turned off on both ends of the pipe. You can turn these flags on with fcntl(). See "fcntl() — Control open file descriptors" on page 474 for details.

If pipe() successfully creates a pipe, it updates the access, change, and modification times for the pipe.

It is unspecified whether *fdinfo*[0] is also open for writing and whether *fdinfo*[1] is also open for reading. z/OS UNIX pipes are not STREAMS-based.

Returned value

If successful, pipe() returns 0.

If unsuccessful, pipe() returns -1 and sets errno to one of the following values:

Error Code
Description

EMFILE
Opening the pipe would exceed the limit on the number of file descriptors the process can have open. This limit is given by OPEN_MAX, defined in the limits.h header file.

ENFILE
Opening the pipe would exceed the number of files that the system can have open simultaneously.

ENOMEM
Opening the pipe requires more space than is available.

Example

CELEBP04

```
/* CELEBP04

   This example creates an I/O channel.
   The output shows the data written into one end and read from
   the other.

 */
#define _POSIX_SOURCE
#include <unistd.h>
#include <stdio.h>

void reverse(char *s) {
  char *first, *last, temp;

  first = s;
  last = s+strlen(s)-1;
  while (first != last) {
    temp = *first;
    *(first++) = *last;
    *(last--) = temp;
  }
}

main() {
  char original[]="This is the original string";
  char buf[80];
  int  p1[2], p2[2];

  if (pipe(p1) != 0)
    perror("first pipe() failed");
  else if (pipe(p2) != 0)
    perror("second pipe() failed");
  else if (fork() == 0) {
    close(p1[1]);
    close(p2[0]);
    if (read(p1[0], buf, sizeof(buf)) == -1)
      perror("read() error in parent");
    else {
      reverse(buf);
      if (write(p2[1], buf, strlen(buf)+1) == -1)
```

```
        perror("write() error in child");
    }
    exit(0);
  }
  else {
    close(p1[0]);
    close(p2[1]);
    printf("parent is writing '%s' to pipe 1\n", original);
    if (write(p1[1], original, strlen(original)+1) == -1)
      perror("write() error in parent");
    else if (read(p2[0], buf, sizeof(buf)) == -1)
      perror("read() error in parent");
    else printf("parent read '%s' from pipe 2\n", buf);
  }
}
```

Output:

```
parent is writing 'This is the original string' to pipe 1
parent read 'gnirts lanigiro eht si sihT' from pipe 2
```

Related information

- "unistd.h" on page 82
- "close() — Close a file" on page 288
- "fcntl() — Control open file descriptors" on page 474
- "open() — Open a file" on page 1152
- "read() — Read from a file or socket" on page 1376
- "write() — Write data on a file or socket" on page 2085

__poe() — Port of entry information

Standards

Standards / Extensions	C or C++	Dependencies
z/OS UNIX	both	z/OS V1R5

Format

```
#define _OPEN_SYS
#include <sys/socket.h>

int __poe(__poecb_t *poecbp);
```

General description

The __poe() function allows the application to specify what port of entry (POE) information the system should use in determining various levels of permission checking. The attributes for the port of entry are used by services that perform user ID security authorization (examples are: setuid(), __login(), __passwd()).

Argument *poecbp* is the address of a __poecb_t structure which is used to control the port of entry operation. The __poecb_t structure is defined in <sys/socket.h>. For proper behavior the user should ensure that this structure has been initialized to zeros before it is populated. The elements of the __poecb_t structure are as follows:

__poe_options

Port of entry options. There are scope and action options. The combination defines the behavior of the function.

The scope option values are:

_POE_SCOPE_THREAD
_POE_SCOPE_PROCESS
_POE_SCOPE_SOCKET

Scope options _POE_SCOPE_THREAD, _POE_SCOPE_PROCESS, and _POE_SCOPE_SOCKET are mutually exclusive. One must be specified.

Note: As of z/OS V1R12, scope options _POE_THREAD and _POE_PROCESS have been deprecated and replaced with new names. The old names remain for compatibility and must be used when TARGET compile option is prior to z/OS V1R12.

The action option values are:

_POE_ACTION_READ
_POE_ACTION_WRITE
_POE_ACTION_SETGET

Action options _POE_ACTION_READ, _POE_ACTION_WRITE, and _POE_ACTION_SETGET are mutually exclusive. These are optional.

__poe_entry_type

Port of entry type. The types are:

_POE_SOCKET
Entry is a file descriptor for a socket.

_POE_FILE
Entry is a file descriptor for a non-socket file. Supported file types are character special, FIFO, regular, symbolic link, and directory.

__poe_entry_len

Port of entry length. The lengths are:

_POE_SOCKET_LEN
Length of a file descriptor for a socket.

_POE_FILE_LEN
Length of a file descriptor for a non-socket file.

__poe_entry_ptr
Address of port of entry.

__poe_poeattr
Port of entry attributes. This element is an IocPoeAttr structure as defined in <termios.h>.

The following table summarizes the port of entry operation according to scope and action:

Options		POE Data		Description
Scope	Action	Source	Destination	
Socket	Read	Socket or file descriptor	POE data in struct __poecb_s	POE data is extracted from the file/socket descriptor supplied by the caller and returned to the caller via the struct __poecb_s.
	Write	n/a	n/a	Request fails with EINVAL
	SetGet	n/a	n/a	Request fails with EINVAL
	None	n/a	n/a	Request fails with EINVAL
Process	Read	Process level (OAPB)	POE data in struct __poecb_s	Process level POE data is copied from the OAPB and is returned to the caller via the struct __poecb_s.
	Write	POE data in struct __poecb_s	Process level (OAPB)	POE data received from the caller via the struct __poecb_s is copied to the process level POE data in the OAPB.
	SetGet	Socket or file descriptor	Process level (OAPB) and POE data in struct __poecb_s	POE data is extracted form the file/socket descriptor supplied by the caller. The data is copied to the process level POE data in the OAPB and returned to the caller via the struct __poecb_s.
	None	Socket or file descriptor	Process level (OAPB)	POE data is extracted from the file/socket descriptor supplied by the caller and copied to the process level POE data in the OAPB.

Options		POE Data		Description
Scope	Action	Source	Destination	
Thread	Read	Thread level (OTCB)	POE data in struct __poecb_s	Thread level POE data copied from the OTCB is returned to the caller via the struct __poecb_s.
	Write	POE data in struct __poecb_s	Thread level (OTCB)	POE data received from the caller via the struct __poecb_s is copied to the thread level POE data in the OTCB.
	SetGet	Socket or file descriptor	Thread level (OTCB) and POE data in struct __poecb_s	POE data is extracted form the file/socket descriptor supplied by the caller. The data is copied to the thread level POE data in the OTCB and returned to the caller via the struct __poecb_s.
	None	Socket or file descriptor	Thread level (OTCB)	POE data is extracted from the file/socket descriptor supplied by the caller and copied to the thread level POE data in the OTCB.

The ability to register port of entry is a privileged operation. An installation has two ways of allowing an application to use this service:

1. For the highest level of security, the installation defines the BPX.POE FACILITY class profile. For an application to use this service the user ID it runs under must be given read access to this profile. See *z/OS UNIX System Services Planning* for more information on setting up this profile.
2. For a lower security arrangement, you can assign the user ID under which the application is run a UID of 0 so that it operates as a superuser.

For more detailed information on the usage of this function see *z/OS Planning for Multilevel Security and the Common Criteria* and *z/OS UNIX System Services Programming: Assembler Callable Services Reference*.

Returned value

If successful, __poe() returns 0.

If unsuccessful, __poe() returns -1 and sets errno to one of the following values:

EINVAL
The __poecb_t structure is not correct. Use __errno2() for more details.

EPERM
The calling process does not have the appropriate privileges to read or write the POE attributes.

EFAULT
A bad address was received. Either the *poecbp* parameter or the *__poe_entry_ptr* field in the __poecb_t structure is not a valid address.

Related information

- “sys/socket.h” on page 74
- __poe() (BPX1POE, BPX4POE) -- Port of entry information” in *z/OS UNIX System Services Programming: Assembler Callable Services Reference*.

poll() — Monitor activity on file descriptors and message queues

Standards

Standards / Extensions	C or C++	Dependencies
XPG4.2 Single UNIX Specification, Version 3	both	

Format

Sockets:

```
#define _XOPEN_SOURCE_EXTENDED 1
#include <poll.h>

int poll(struct pollfd fds[], nfds_t nmsgsfds, int timeout);
```

Message queues and sockets:

```
#define _XOPEN_SOURCE_EXTENDED 1
#define _OPEN_MSGQ_EXT
#include <sys/types.h>
#include <sys/time.h>
#include <sys/msg.h>
#include <poll.h>

int poll(void *listptr, nmsgsfds_t nmsgsfds, int timeout);
```

_OPEN_MSGQ_EXT must be defined if message queues are to be monitored.

General description

The poll() function provides applications with a mechanism for multiplexing input/output over the following set of file descriptors:

- regular files
- terminal and pseudoterminal devices
- STREAMS-based files

- sockets
- message queues.
- FIFOs
- pipes

For each member of the array(s) pointed to by *listptr*, poll() examines the given file descriptor or message queue for the event(s) specified in the member. The number of pollmsg structures and the number of pollfd structures in the arrays are specified by *nmsgsfds*. The poll() function identifies those file descriptors on which an application can read or write data, or on which an error event has occurred.

listptr A pointer to an array of pollfd structures, pollmsg structures, or to a pollist structure. Each structure specifies a file descriptor or message queue identifier and the events of interest for this file or message queue. The type of parameter to pass depends on whether you want to monitor file and socket descriptors, message queue identifiers, or both. To monitor socket descriptors only, set the high-order halfword of *nmsgsfds* to 0, the low-order halfword to the number of pollfd structures to be provided, and pass a pointer to an array of pollfd structures. To monitor message queues only, set the low-order halfword of *nmsgsfds* to 0, the high-order halfword to the number of pollmsg structures to be provided, and pass a pointer to an array of pollmsg structures. To monitor both, set *nmsgsfds* as described below, and pass a pointer to a pollist structure. If a pollist structure is to be used, a structure similar to the following should be defined in a user program. The pollfd structure must precede the pollmsg structure.

```
struct pollist {
   struct pollfd fds[3];
   struct pollmsg msgids[2];
   } list;
```

nmsgsfds

The number of pollmsg structures and the number of pollfd structures pointed to by *listptr*.

This parameter is divided into two parts. The first half (the high-order 16 bits) gives the number of pollmsg structures containing message queue identifiers. This number must not exceed the value 32767. The second half (the low-order 16 bits) gives the number of pollfd structures containing file descriptors to check. If either half of the *nmsgsfds* parameter is equal to a value of 0, the corresponding pollmsg structures or pollfd structures is assumed not to be present.

timeout

The amount of time, in milliseconds, to wait for an event to occur.

If none of the defined events have occurred on any selected descriptor, poll() waits at least *timeout* milliseconds for an event to occur on any of the selected descriptors. If the value of *timeout* is 0, poll() returns immediately. If the value of *timeout* is -1, poll() blocks until a requested event occurs or until the call is interrupted.

The above processing also applies to message queues.

Each pollfd or pollmsg structure contains the following fields:
- fd/msgid - open file descriptor or message queue identifier
- events - requested events
- revents - returned events

The events and revents fields are bitmasks constructed by OR-ing a combination of the following event flags:

POLLERR
An error or exceptional condition has occurred. This flag is only valid in the revents bitmask; it is ignored in the events bitmask.

POLLHUP
The device has been disconnected. This event and POLLOUT are mutually exclusive, a stream can never be writable if a hang-up has occurred. However, this event and POLLIN, POLLRDNORM, POLLRDBAND or POLLPRI are not mutually exclusive. This flag is only valid in the revents bitmask. It is ignored in the events member.

POLLIN
Same as POLLRDNORM

POLLNVAL
The specified fd/msgid value is invalid. This flag is only valid in the revents bitmask; it is ignored in the events bitmask.

POLLOUT
Same as POLLWRNORM

POLLPRI
Out-of-band data may be received without blocking.

POLLRDBAND
Data from a nonzero priority band may be read without blocking. For STREAMS, this flag is set in revents even if the message is of zero length.

POLLRDNORM
Normal data may be read without blocking.

POLLWRBAND
Priority data (priority band greater than 0) may be written.

POLLWRNORM
Normal data may be written without blocking.

Note: Poll bits are supported as follows.

Regular Files
Always poll() true for reading and writing. This means that all poll() read and write bits are supported. They will never return with POLLERR or POLLHUP.

FIFOs / PIPEs
Do not have the concept of out-of-band data or priority band data. They support POLLIN, POLLRDNORM, POLLOUT, and POLLWRNORM. They ignore POLLPRI, POLLRDBAND, and POLLWRBAND. They never return POLLERR.

TTYs / OCS
Same support as FIFOs and PIPEs, except that TTYs may return POLLERR.

Sockets
Have the concept of out-of-band data. They support POLLIN, POLLRDNORM, POLLOUT, POLLWRNORM, and POLLPRI for out-of-band data. They ignore POLLRDBAND and POLLWRBAND. They may return POLLERR, and never return POLLHUP.

If the value of fd/msgid is less than 0, events is ignored and revents is set to 0 in that entry on return from poll().

In each pollfd structure, poll() clears the revents member except that where the application requested a report on a condition by setting one of the bits of events listed above, poll() sets the corresponding bit in revents if the requested condition is true. In addition, poll() sets the **POLLERR** flag in revents if the condition is true, even if the application did not set the corresponding bit in events.

The poll() function is not affected by the **O_NONBLOCK** flag.

A file descriptor for a socket that is listening for connections will indicate that it is ready for reading, once connections are available. A file descriptor for a socket that is connecting asynchronously will indicate that it is ready for writing, once a connection has been established.

The following macros are provided to manipulate the *nmsgsfds* parameter and the return value from poll():

Macro Description

_SET_FDS_MSGS(*nmsgsfds, nmsgs, nfds***)**
Sets the high-order halfword of *nmsgsfds* to *nmsgs*, and sets the low-order halfword of *nmsgsfds* to *nfds*.

_NFDS(*n***)**
If the return value *n* from poll() is nonnegative, returns the number of socket descriptors that meet the read, write, and exception criteria. A descriptor may be counted multiple times if it meets more than one given criterion.

_NMSGS(*n***)**
If the return value *n* from poll() is nonnegative, returns the number of message queues that meet the read, write, and exception criteria. A message queue may be counted multiple times if it meets more than one given criterion.

Returned value

If successful, poll() returns a nonnegative value.

A positive value indicates the total number of events that were found to be ready among the message queues and the total number of events that were found to be ready among the file descriptors. The return value is similar to *nmsgsfds* in that the high-order 16 bits of the return value give the number associated with message queues, and the low-order 16 bits give the number associated with file descriptors. Should the number associated with message queues be greater than 32767, only 32767 will be reported. This is to ensure that the return value does not appear to be negative. Should the number associated with file descriptors be greater than 65535, only 65535 will be reported.

If the call timed out and no file descriptors have been selected, poll() returns 0.

If unsuccessful, poll() returns -1 and sets errno to one of the following values:

Error Code
Description

EAGAIN
The allocation of internal data structures failed, but a subsequent request may succeed.

EINTR
A signal was caught during poll().

EINVAL
One of the parameters specified a value that was not correct. Consult the reason code to determine the exact reason the error occurred. The following reason codes can accompany this return code.

- JRWAITFOREVER
- JRINVALIDNFDS
- JRNOFDSTOOMANYQIDS

EIO One of the descriptors in the select mask has become inoperative and it is being repeatedly included in a select even though other operations against this descriptor have been failing with EIO. A socket descriptor, for example, can become inoperative if TCP/IP is shut down. A failure from select can not tell you which descriptor has failed so generally select will succeed and these descriptors will be reported to you as being ready for whatever event they were being selected for. Subsequently when the descriptor is used on a receive or other operation you will receive the EIO failure and can react to the problem with the individual descriptor. In general you would close() the descriptor and remove it from the next select mask. If the individual descriptor's failing return code is ignored though and an inoperative descriptor is repeatedly selected on and used, even though each time it is used that call fails with EIO, eventually the select call itself will fail with EIO.

Related information

- "poll.h" on page 58
- "sys/msg.h" on page 74
- "sys/time.h" on page 75
- "sys/types.h" on page 75
- "accept() — Accept a new connection on a socket" on page 105
- "connect() — Connect a socket" on page 308
- "listen() — Prepare the server for incoming client requests" on page 972
- "msgctl() — Message control operations" on page 1104
- "msgget() — Get message queue" on page 1105
- "msgrcv() — Message receive operation" on page 1108
- "msgsnd() — Message send operations" on page 1113
- "read() — Read from a file or socket" on page 1376
- "recv() — Receive data on a socket" on page 1398
- "select(), pselect() — Monitor activity on files or sockets and message queues" on page 1470
- "selectex() — Monitor activity on files or sockets and message queues" on page 1479
- "send() — Send data on a socket" on page 1494
- "write() — Write data on a file or socket" on page 2085

popen() — Initiate a pipe stream to or from a process

Standards

Standards / Extensions	C or C++	Dependencies
XPG4 XPG4.2 Single UNIX Specification, Version 3	both	POSIX(ON)

Format

```
#define  _XOPEN_SOURCE
#include <stdio.h>

FILE *popen(const char *command, const char *mode);
```

General description

The popen() function executes the command specified by the string *command*. It creates a pipe between the calling program and the executed command, and returns a pointer to a stream that can be used to either read from or write to the pipe.

The environment of the executed command will be as if a child process were created within the popen() call using fork(), and the child invoked the sh utility using the call:

```
execl("/bin/sh", "sh", "-c", command, (char *)0);
```

The popen() function ensures that any streams from previous popen() calls that remain open in the parent process are closed in the child process.

The *mode* argument to popen() is a string that specifies I/O mode:

1. If *mode* is **r**, file descriptor **STDOUT_FILENO** will be the writable end of the pipe when the child process is started. The file descriptor *fileno(stream)* in the calling process, where *stream* is the stream pointer returned by popen(), will be the readable end of the pipe.
2. If *mode* is **w**, file descriptor **STDIN_FILENO** will be the readable end of the pipe when the child process is started. The file descriptor *fileno(stream)* in the calling process, where *stream* is the stream pointer returned by popen(), will be the writable end of the pipe.
3. If *mode* is any other value, a NULL pointer is returned and errno is set to EINVAL.

After popen(), both the parent and the child process will be capable of executing independently before either terminates.

Because open files are shared, a mode *r* command can be used as an input filter and a mode *w* command as an output filter.

Buffered reading before opening an input filter (that is, before popen()) may leave the standard input of that filter mispositioned. Similar problems with an output filter may be prevented by buffer flushing with fflush().

A stream opened with popen() should be closed by pclose().

The behavior of popen() is specified for values of *mode* of *r* and *w*. *mode* values of *rb* and *wb* are supported but are not portable.

If the shell command cannot be executed, the child termination status returned by pclose() will be as if the shell command terminated using *exit(127)* or *_exit(127)*.

If the application calls waitpid() with a *pid* argument greater than *0*, and it still has a stream that was created with popen() open, it must ensure that *pid* does not refer to the process started by popen()

The stream returned by popen() will be designated as byte-oriented.

Special behavior for file tagging and conversion: When the FILETAG(,AUTOTAG) runtime option is specified, the pipe opened for communication between the parent and child process by popen() will be tagged with the writer''s program CCSID upon first I/O. For example, if popen(some_command, "r") were specified, then the stream returned by the popen() would be tagged in the child process'' program CCSID.

Returned value

If successful, popen() returns a pointer to an open stream that can be used to read or write to a pipe.

If unsuccessful, popen() returns a NULL pointer and sets errno to one of the following values:

Error Code
  Description

EINVAL
  The *mode* argument is invalid.

popen() may also set errno values as described by spawn(), fork(), or pipe().

Related information

- “stdio.h” on page 68
- “fflush() — Write buffer to file” on page 523
- “fork() — Create a new process” on page 571
- “pclose() — Close a pipe stream to or from a process” on page 1174
- “pipe() — Create an unnamed pipe” on page 1179
- “system() — Execute a command” on page 1805

posix_openpt() — Open a pseudo-terminal device

Standards

Standards / Extensions	C or C++	Dependencies
Single UNIX Specification, Version 3	both	z/OS V1.9

Format

```
#define _XOPEN_SOURCE 600
#include <stdlib.h>
#include <fcntl.h>

int posix_openpt(int oflag);
```

General description

The posix_openpt() function establishes a connection between a master device for a pseudo-terminal and a file descriptor. The file descriptor is used by other I/O functions that refer to that pseudo-terminal.

The file status flags and file access modes of the open file description are set according to the value of oflag.

Values for oflag are constructed by a bitwise-inclusive OR of flags from the following list, defined in <fcntl.h>:

O_RDWR
Open for reading and writing.

O_NOCTTY
If set posix_openpt() will not cause the terminal device to become the controlling terminal for the process.

The behavior of other values for the oflag argument is unspecified.

Argument
Description

oflag The value of the file status flags and file access modes |of the open file description.

Returned value

Upon successful completion, the posix_openpt() function opens a master pseudo-terminal device and returns a non-negative integer representing the lowest numbered unused file descriptor. Otherwise, -1 is returned and errno set to indicate the error.

Error Code
Description

EMFILE
{OPEN_MAX} file descriptors are currently open in the calling process.

ENFILE
The maximum allowable number of files is currently open in the system.

EINVAL
The value of oflag is not valid.

EAGAIN
Out of pseudo-terminal resources.

Example

CELEBP71

```
/* CELEBP71

   This example demonstrates how to use posix_openpt() to open a
   master psuedo-terminal device.

   Expected output:
   The master psuedo-terminal id is [first available descriptor]

*/
#define _XOPEN_SOURCE 600
#include <stdlib.h>
#include <fcntl.h>
#include <stdio.h>

void main() {
   int fd;

   fd = posix_openpt(O_RDWR | O_NOCTTY);
   if (fd == -1)
```

```
      perror("Error opening a terminal.\n");
   else
      printf("The master psuedo-terminal id is %d\n",fd);
}
```

Related information

pow(), powf(), powl() — Raise to power

Standards

Standards / Extensions	C or C++	Dependencies
ISO C POSIX.1 XPG4 XPG4.2 ISO/ANSI C++ C99 Single UNIX Specification, Version 3 C++ TR1 C99	both	

Format

```
#include <math.h>

double pow(double x, double y);
double pow(double x, int y);                    /* C++ only */
float pow(float x, int y);                      /* C++ only */
float pow(float x, float y);                    /* C++ only */
long double pow(long double x, int y);          /* C++ only */
long double pow(long double x, long double y);  /* C++ only */
float powf(float x, float y);
long double powl(long double x, long double y);
```

General description

The pow(), powf(), and powl() functions calculate the value of *x* to the power of *y*.

Note: These functions work in both IEEE Binary Floating-Point and hexadecimal floating-point formats. See "IEEE binary floating-point" on page 94 for more information about IEEE Binary Floating-Point.

Restriction: The powf() function does not support the _FP_MODE_VARIABLE feature test macro.

Returned value

If successful, the pow(), powf(), and powl() functions return the value of *x* to the power of *y*.

If *y* is 0, the function returns 1.

If *x* is negative and *y* is non-integral, the function sets errno to EDOM and returns -HUGE_VAL. If the correct value is outside the range of representable values, ±HUGE_VAL is returned according to the sign of the value, and the value of ERANGE is stored in errno.

Special behavior for IEEE: If *x* is negative or 0, then the *y* parameter must be an integer. If *y* is 0, the function returns 1.0 for all *x* parameters.

If an overflow occurs, the function returns HUGE_VAL and sets errno to ERANGE.

If both *x* and *y* are negative, the function returns NaNQ and sets errno to EDOM.

If *x* is 0 and *y* is negative, the function returns HUGE_VAL and does not modify errno, but powf sets errno to ERANGE.

Note: When environment variable _EDC_SUSV3 is set to 2, and if *x* is 0 and *y* is negative, the function returns -HUGE_VAL and sets errno to ERANGE.

Example

CELEBP05

```
/* CELEBP05

   This example calculates the value of 2**3.

 */
#include <math.h>
#include <stdio.h>

int main(void)
{
   double x, y, z;

   x = 2.0;
   y = 3.0;
   z = pow(x,y);

   printf("%lf to the power of %lf is %lf\n", x, y, z);
}
```

Output:

```
2.000000 to the power of 3.000000 is 8.000000
```

Related information

- “math.h” on page 44
- “exp(), expf(), expl() — Calculate exponential function” on page 448
- “log(), logf(), logl() — Calculate natural logarithm” on page 990
- “log10(), log10f(), log10l() — Calculate base 10 logarithm” on page 1000

powd32(), powd64(), powd128() — Raise to power

Standards

Standards / Extensions	C or C++	Dependencies
C/C++ DFP	both	z/OS V1.8

Format

```
#define __STDC_WANT_DEC_FP__
#include <math.h>

_Decimal32  powd32(_Decimal32 x, _Decimal32 y);
_Decimal64  powd64(_Decimal64 x, _Decimal64 y);
_Decimal128 powd128(_Decimal128 x, _Decimal128 y);
```

```
_Decimal32  pow(_Decimal32 x, _Decimal32 y);   /* C++ only */
_Decimal64  pow(_Decimal64 x, _Decimal64 y);   /* C++ only */
_Decimal128 pow(_Decimal128 x, _Decimal128 y); /* C++ only */
```

General description

The powd32(), powd64(), and powd128() functions calculate the value of *x* to the power of *y*.

Notes:

1. To use IEEE decimal floating-point, the hardware must have the Decimal Floating-Point Facility installed.
2. These functions work in IEEE decimal floating-point format. See "IEEE Decimal Floating-Point" for more information.

Returned value

If successful, the powd32(), powd64(), and powd128() functions return the value of *x* to the power of *y*.

If *x* is negative or 0, then the *y* parameter must be an integer. If *y* is 0, the function returns 1.0 for all *x* parameters.

If an overflow occurs, the function returns HUGE_VAL_D32, HUGE_VAL_D64, or HUGE_VAL_D128 and sets errno to ERANGE.

If *x* is negative and *y* is not an integer, the function returns NaNQ and sets errno to EDOM.

If *x* is 0 and *y* is negative, the function returns HUGE_VAL_D32, HUGE_VAL_D64, or HUGE_VAL_D128 but does not modify errno.

Example

```
/* CELEBP59

   This example illustrates the powd64() function

*/

#define  __STDC_WANT_DEC_FP__
#include <math.h>
#include <stdio.h>

int main(void)
{
   _Decimal64 x, y, z;

   x = 2.0DD;
   y = 3.0DD;
   z = powd64(x, y);

   printf("%Df to the power of %Df is %Df\n", x, y, z);
}
```

Related information

- “math.h” on page 44
- “expd32(), expd64(), expd128() — Calculate exponential function” on page 449
- “logd32(), logd64(), logd128() — Calculate natural logarithm” on page 994

- “log10d32(), log10d64(), log10d128() — Calculate base 10 logarithm” on page 1002
- “pow(), powf(), powl() — Raise to power” on page 1193

__pow_i() — Raise to a power (R**I)

Standards

Standards / Extensions	C or C++	Dependencies
Language Environment	both	z/OS V1R7

Format

```
#include <math.h>

double __pow_i(double x, int y);
```

General description

The __pow_i() function calculates the value of *x* to the power of *y* and is a C interface to the Language Environment Math Service CEESDXPI. Information about the Language Environment math service CEESDXPI can be found in the following publications:

- *z/OS Language Environment Programming Guide*
- *z/OS Language Environment Programming Reference*
- *z/OS V2R1.0 Language Environment Concepts Guide*

Note: The following table shows the viable formats for these functions. See “IEEE binary floating-point” on page 94 for more information about IEEE Binary Floating-Point.

Function	Hex	IEEE
__pow_i	X	X

Returned value

If successful, __pow_i() returns the value of *x* to the power of *y*.

If...	Then...
x is not equal to 0 and *y* is 0	1 is returned.
x is 0 and *y* is positive	0 is returned.
x and *y* are 0	0 is returned and errno is set to EDOM.
x is 0 and *y* is negative	±HUGE_VAL is returned and errno is set to EDOM.
x and *y* cause an overflow	HUGE_VAL is returned.

Related information

- “__pow_ii() — Raise to a power (I**I)” on page 1197

__pow_ii() — Raise to a power (I**I)

Standards

Standards / Extensions	C or C++	Dependencies
Language Environment	both	z/OS V1R7

Format

```
#include <math.h>

int __pow_ii(int x, int y);
```

General description

The __pow_ii() function calculates the value of *x* to the power of *y* and is a C interface to the LE Math Service CEESIXPI. Information about the LE Math Service CEESIXPI can be found in the following publications:

- *z/OS Language Environment Programming Guide*
- *z/OS Language Environment Programming Reference*
- *z/OS V2R1.0 Language Environment Concepts Guide*

Note: The following table shows the viable formats for these functions. See "IEEE binary floating-point" on page 94 for more information about IEEE Binary Floating-Point.

Function	Hex	IEEE
__pow_ii	X	X

Returned value

If successful, __pow_ii() returns the value of *x* to the power of *y*.

If...	Then...
x is not equal to 0 and *y* is 0	1 is returned.
x is 0 and *y* is positive	0 is returned.
x is 0 and *y* is negative	INT_MAX is returned and errno is set to EDOM.
x and *y* are 0	0 is returned and errno is set to EDOM.
x is 1 and *y* is negative	1 is returned.
x is -1 and *y* is negative	±1 is returned.
x is greater than 1 and *y* is negative	0 is returned and errno is set to EDOM.
x is less than -1 and *y* is negative	0 is returned and errno is set to EDOM.
The values of *x* and *y* cause an overflow and *x* is less than 0 or *y* is odd.	errno is set to ERANGE and the function returns INT_MIN
The values of *x* and *y* cause an overflow and *x* is greater than 0 or *y* is even.	errno is set to ERANGE and the function returns INT_MAX

Related information

- "__pow_i() — Raise to a power (R**I)" on page 1196

pread() — Read from a file or socket without file pointer change

Standards

Standards / Extensions	C or C++	Dependencies
Single UNIX Specification, Version 2 Single UNIX Specification, Version 3	both	OS/390 V2R10

Format

```
#define _XOPEN_SOURCE 500
#include <unistd.h>

ssize_t pread(int fildes, void *buf, size_t nbyte, off_t offset);
```

General description

The pread() function performs the same action as read(), except that it reads from a given position in the file without changing the file pointer.

The first three arguments to pread() are the same as read(), with the addition of a fourth argument *offset* for the desired position inside the file.

An error result is returned for any attempt to perform a pread() on a file that is incapable of a seek action.

For regular files, no data transfer will occur past the offset maximum established in the open file description associated with *fildes*.

Returned value

If successful, pread() returns a non-negative integer indicating the number of bytes actually read.

If unsuccessful, pread() returns -1 and sets errno to one of the following values:

Error Code
Description

EAGAIN
O_NONBLOCK is set to 1, but data was not available for reading.

EBADF
fildes is not a valid file or socket descriptor.

ECONNRESET
A connection was forcibly closed by a peer.

EFAULT
Using the *buf* and *nbyte* parameters would result in an attempt to access memory outside the caller's address space.

EINTR
pread() was interrupted by a signal that was caught before any data was available.

EINVAL
nbyte contains a value that is less than 0, or the request is invalid or not supported, or the STREAM or multiplexer referenced by *fildes* is linked (directly or indirectly) downstream from a multiplexer.

The *offset* argument is invalid. The value is negative.

EIO The process is in a background process group and is attempting to read from its controlling terminal, and either the process is ignoring or blocking the SIGTTIN signal or the process group of the process is orphaned. For sockets, an I/O error occurred.

ENOBUFS
Insufficient system resources are available to complete the call.

ENOTCONN
A receive was attempted on a connection-oriented socket that is not connected.

ENXIO
A request was outside the capabilities of the device.

EOVERFLOW
The file is a regular file and an attempt was made to read or write at or beyond the offset maximum associated with the file.

ESPIPE
fildes is associated with a pipe or FIFO.

ETIMEDOUT
The connection timed out during connection establishment, or due to a transmission timeout on active connection.

EWOULDBLOCK
socket is in nonblocking mode and data is not available to read. or the SO_RCVTIMEO timeout value was been reached before data was available.

Related information

- “unistd.h” on page 82
- “pwrite() — Write data on a file or socket without file pointer change” on page 1361
- “read() — Read from a file or socket” on page 1376

printf() — Format and write data

The information for this function is included in “fprintf(), printf(), sprintf() — Format and write data” on page 588.

pselect() - Monitor activity on files or sockets and message queues

The information for this function is included in “select(), pselect() — Monitor activity on files or sockets and message queues” on page 1470.

pthread_atfork() - Register fork handlers

Standards

Standards / Extensions	C or C++	Dependencies
Single UNIX Specification, Version 3	both	z/OS V1R9 POSIX(ON)

Format

```
#define _UNIX03_THREADS
#include <pthread.h>

int pthread_atfork(void (*prepare)(void), void (*parent)(void),
void (*child)(void));
```

General description

The *pthread_atfork()* function registers fork handlers to be called before and after *fork()*, in the context of the thread that called *fork()*. Fork handler functions may be named for execution at the following three points in thread processing:

- The *prepare* handler is called before *fork()* processing commences.
- The *parent* handler is called after *fork()* processing completes in the parent process.
- The *child* handler is called after *fork()* processing completes in the child process.

If any argument to *pthread_atfork()* is NULL, the call does not register a handler to be invoked at the point corresponding to that argument.

The order of calls to *pthread_atfork()* is significant. The *parent* and *child* fork handlers are called in the order in which they were established by calls to *pthread_atfork()*. The *prepare* fork handlers are called in the opposite order.

The intended purpose of *pthread_atfork()* is to provide a mechanism for maintaining the consistency of mutex locks between parent and child processes. The handlers are expected to be straightforward programs, designed simply to manage the synchronization variables and must return to ensure that all registered handlers are called. Historically, the prepare handler acquired needed mutex locks, and the parent and child handlers released them. Unfortunately, this usage is not practical on the z/OS platform and is not guaranteed to be portable in the current UNIX standards. When the parent process is multi-threaded (invoked *pthread_create()* at least once), the child process can only safely call async-signal-safe functions before it invokes an *exec()* family function. This restriction was added to the POSIX.1 standard in 1996. Because functions such as *pthread_mutex_lock()* and *pthread_mutex_unlock()* are not async-signal-safe, unpredictable results may occur if they are used in a child handler.

Special behavior for z/OS XL C: The C Library *pthread_atfork()* function has the following restrictions:

- Any fork handler registered by a fetched module that has been released is removed from the list at the time of release. See “fetch() — Get a load module” on page 507,“fetchep() — Share writable static” on page 519, and “release() — Delete a load module” on page 1424 for details about fetching and releasing modules.
- Any handler registered in an explicitly loaded DLL (using *dllload()* or *dlopen()*) that has been freed (using *dllfree()* or *dlclose()*) is removed from the list, except when the DLL has also been implicitly loaded.
- Use of non-C subroutines or functions as fork handlers will result in undefined behavior.

Special behavior for z/OS XL C++:

- All of the behaviors listed under "Special Behavior for z/OS XL C."

- The *pthread_atfork()* function cannot receive C++ function pointers compiled as 31-bit non-XPLINK objects, because C and C++ linkage conventions are incompatible in that environment. If you attempt to pass a C++ function pointer to pthread_atfork(), the compiler will flag it as an error. In C++, you must ensure that all handlers registered for use by pthread_atfork() have C linkage by declaring them as extern "C".

Returned value

If successful, pthread_atfork() returns zero; otherwise, it returns an error number.

Error Code
Description

ENOMEM
Insufficient table space exists to record the fork handler addresses.

Example

CELEBP60

```
/* CELEBP60 */
/* Example using SUSv3 pthread_atfork() interface */

#define _UNIX03_THREADS 1

#include <pthread.h>
#include <stdio.h>
#include <unistd.h>
#include <fcntl.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <stdlib.h>
#include <errno.h>

char fn_c[] = "childq.out";
char fn_p[] = "parentside.out";
int  fd_c;
int  fd_p;

void prep1(void)  {
  char buff[80] = "prep1\n";
  write(4,buff,sizeof(buff));
}

void prep2(void)  {
  char buff[80] = "prep2\n";
  write(4,buff,sizeof(buff));
}

void prep3(void)  {
  char buff[80] = "prep3\n";
  write(4,buff,sizeof(buff));
}


void parent1(void)  {
  char buff[80] = "parent1\n";
  write(4,buff,sizeof(buff));
}

void parent2(void)  {
  char buff[80] = "parent2\n";
  write(4,buff,sizeof(buff));
}
```

```
void parent3(void)  {
  char buff[80] = "parent3\n";
  write(4,buff,sizeof(buff));
}


void child1(void)  {
  char buff[80] = "child1\n";
  write(3,buff,sizeof(buff));
}

void child2(void)  {
  char buff[80] = "child2\n";
  write(3,buff,sizeof(buff));
}

void child3(void)  {
  char buff[80] = "child3\n";
  write(3,buff,sizeof(buff));
}

void *thread1(void *arg) {

  printf("Thread1: Hello from the thread.\n");

}


int main(void)
{
  pthread_t thid;
  int       rc, ret;
  pid_t     pid;
  int       status;
  char   header[30] = "Called Child Handlers\n";


  if (pthread_create(&thid, NULL, thread1, NULL) != 0) {
    perror("pthread_create() error");
    exit(3);
  }

  if (pthread_join(thid, NULL) != 0) {
    perror("pthread_join() error");
    exit(5);
  } else {
    printf("IPT: pthread_join success!  Thread 1 should be finished now.\n");
    printf("IPT: Prepare to fork!!!\n");
  }


  /*-----------------------------------------*/
  /*|  Start atfork handler calls in parent  */
  /*-----------------------------------------*/
  /* Register call 1 */
  rc = pthread_atfork(&prep1, &parent2, &child3);
  if (rc != 0) {
     perror("IPT: pthread_atfork() error [Call #1]");
     printf("  rc= %d, errno: %d, ejr: %08x\n", rc, errno, __errno2());
  }


  /* Register call 2 */
  rc = pthread_atfork(&prep2, &parent3, &child1);
  if (rc != 0) {
     perror("IPT: pthread_atfork() error [Call #2]");
     printf("  rc= %d, errno: %d, ejr: %08x\n", rc, errno, __errno2());
  }


  /* Register call 3 */
  rc = pthread_atfork(&prep3, &parent1, NULL);
  if (rc != 0) {
     perror("IPT: pthread_atfork() error [Call #3]");
     printf("  rc= %d, errno: %d, ejr: %08x\n", rc, errno, __errno2());
  }

  /* Create output files to expose the execution of fork handlers. */
  if ((fd_c = creat(fn_c, S_IWUSR)) < 0)
    perror("creat() error");
  else
    printf("Created %s and assigned fd= %d\n", fn_c, fd_c);
  if ((ret = write(fd_c,header,30)) == -1)
    perror("write() error");
  else
    printf("Write() wrote %d bytes in %s\n", ret, fn_c);

  if ((fd_p = creat(fn_p, S_IWUSR)) < 0)
    perror("creat() error");
  else
    printf("Created %s and assigned fd= %d\n", fn_p, fd_p);
  if ((ret = write(fd_p,header,30)) == -1)
    perror("write() error");
  else
    printf("Write() wrote %d bytes in %s\n", ret, fn_p);

  pid = fork();

  if (pid < 0)
    perror("IPT: fork() error");
  else {
    if (pid == 0) {
      printf("Child: I am the child!\n");
      printf("Child: My PID= %d, parent= %d\n", (int)getpid(),
              (int)getppid());
      exit(0);

    } else {
      printf("Parent: I am the parent!\n");
      printf("Parent: My PID= %d, child PID= %d\n", (int)getpid(), (int)pid);

      if (wait(&status) == -1)
        perror("Parent: wait() error");
      else if (WIFEXITED(status))
             printf("Child exited with status: %d\n",WEXITSTATUS(status));
           else
             printf("Child did not exit successfully\n");

    close(fd_c);
    close(fd_p);

    }

  }


}
```

Related information

- “pthread.h” on page 58
- “pthread_create() — Create a thread” on page 1261
- “fork() — Create a new process” on page 571

- "atexit() — Register program termination function" on page 199

pthread_attr_destroy() — Destroy the thread attributes object

Standards

Standards / Extensions	C or C++	Dependencies
POSIX.4a Single UNIX Specification, Version 3	both	POSIX(ON)

Format

```
#define _OPEN_THREADS
#include <pthread.h>

int pthread_attr_destroy(pthread_attr_t *attr);
```

SUSV3:

```
#define _UNIX03_THREADS
#include <pthread.h>

int pthread_attr_destroy(pthread_attr_t *attr);
```

General description

Removes the definition of the thread attributes object. An error results if a thread attributes object is used after it has been destroyed.

attr is a pointer to a thread attribute object initialized by pthread_attr_init().

You can use a thread attribute object to manage the characteristics of threads in your application. It defines the set of values to be used for the thread during its creation. By establishing a thread attribute object, you can create many threads with the same set of characteristics, without defining those characteristics for each thread. You can define more than one thread attribute object.

If a thread attribute object is shared between threads, the application must provide the necessary synchronization because a thread attribute object is defined in the application's storage.

Returned value

If successful, pthread_attr_destroy() returns 0.

If unsuccessful, pthread_attr_destroy() returns -1.

There are no documented errno values. Use perror() or strerror() to determine the cause of the error.

Special behavior for Single UNIX Specification, Version 3: If unsuccessful, pthread_attr_destroy() returns an error number to indicate the error.

Example

CELEBP06

```
/* CELEBP06 */
#define _OPEN_THREADS
#include <stdio.h>
#include <pthread.h>

void *thread1(void *arg) {
  pthread_exit(NULL);
}

int main()
{
   pthread_t      thid;
   pthread_attr_t attr;

   if (pthread_attr_init(&attr) == -1) {
      perror("error in pthread_attr_init");
      exit(1);
   }

   if (pthread_create(&thid, &attr, thread1, NULL) == -1) {
      perror("error in pthread_create");
      exit(2);
   }

   if (pthread_detach(&thid) == -1) {
      perror("error in pthread_detach");
      exit(4);
   }

   if (pthread_attr_destroy(&attr) == -1) {
      perror("error in pthread_attr_destroy");
      exit(5);
   }
   exit(0);
}
```

Related information

- "pthread.h" on page 58
- "pthread_attr_init() — Initialize a thread attribute object" on page 1218
- "pthread_create() — Create a thread" on page 1261

pthread_attr_getdetachstate() — Get the detach state attribute

Standards

Standards / Extensions	C or C++	Dependencies
POSIX.4a Single UNIX Specification, Version 3	both	POSIX(ON)

Format

```
#define _OPEN_THREADS
#include <pthread.h>

int pthread_attr_getdetachstate(pthread_attr_t *attr);
```

SUSV3:

```
#define _UNIX03_THREADS
#include <pthread.h>
int pthread_attr_getdetachstate(const pthread_attr_t *attr,
                                int *detachstate);
```

General description

Returns the current value of the detachstate attribute for the thread attribute object, *attr*, that is created by pthread_attr_init(). The detachstate attribute values are:

0 Undetached. An undetached thread will keep its resources after termination.

1 Detached. A detached thread will have its resources automatically freed by the system at termination. Thus, you cannot get the thread's termination status (or wait for the thread to terminate) by using pthread_join().

You can use a thread attribute object to manage the characteristics of threads in your application. It defines the set of values to be used for the thread during its creation. By establishing a thread attribute object, you can create many threads with the same set of characteristics, without defining those characteristics for each thread. You can define more than one thread attribute object.

Returned value

If successful, pthread_attr_getdetachstate() returns the detachstate (0 or 1).

If unsuccessful, pthread_attr_getdetachstate() returns -1.

There are no documented errno values. Use perror() or strerror() to determine the cause of the error.

Special behavior for Single UNIX Specification, Version 3: If unsuccessful, pthread_attr_getdetachstate() returns an error number to indicate the error.

Example

CELEBP07

```
/* CELEBP07 */
#define _OPEN_THREADS
#include <stdio.h>
#include <pthread.h>

int main()
{
   pthread_attr_t attr;
   char           typ[12];

   if (pthread_attr_init(&attr) == -1) {
      perror("error in pthread_attr_init");
      exit(1);
   }

   switch(pthread_attr_getdetachstate(&attr)) {
      default:
         perror("error in pthread_attr_getdetachstate()");
         exit(2);
      case 0:
         strcpy(typ, "undetached");
         break;
      case 1:
         strcpy(typ, "detached");
   }
   printf("The detach state is %s.\n", typ);

   if (pthread_attr_destroy(&attr) == -1) {
      perror("error in pthread_attr_destroy");
      exit(2);
   }
   exit(0);
}
```

CELEBP61

```
/* CELEBP61 */
/* Example using SUSv3 pthread_attr_getdetachstate() interface */

#define _UNIX03_THREADS 1
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <pthread.h>
#include <errno.h>


int main(void)
{
  pthread_attr_t attr;
  int            rc, newstate, foundstate;
  char           state[12];

  if (pthread_attr_init(&attr) == -1) {
     perror("error in pthread_attr_init");
     exit(1);
  }

  newstate = PTHREAD_CREATE_DETACHED;
  pthread_attr_setdetachstate(&attr, newstate);

  rc = pthread_attr_getdetachstate(&attr,&foundstate);
  switch(foundstate) {
      case PTHREAD_CREATE_JOINABLE:
         strcpy(state,"joinable");
         break;
      case PTHREAD_CREATE_DETACHED:
         strcpy(state,"detached");
         break;
      default:
         printf("pthread_attr_getdetachstate returned: %d\n", rc);
         printf("Error: %d, Errno_Jr: %08x\n", errno, __errno2());
         exit(2);
  }

  printf("Threads created with this attribute object are %s.\n",state);

  if (pthread_attr_destroy(&attr) == -1) {
     perror("error in pthread_attr_destroy");
     exit(3);
  }

  exit(0);
}
```

Related information

- "pthread.h" on page 58
- "pthread_attr_init() — Initialize a thread attribute object" on page 1218
- "pthread_attr_setdetachstate() — Set the detach state attribute" on page 1220
- "pthread_create() — Create a thread" on page 1261

pthread_attr_getguardsize() - Get guardsize attribute

Standards

Standards / Extensions	C or C++	Dependencies
Single UNIX Specification, Version 3	both	z/OS V1R9 POSIX(ON)

Format

```
#define _UNIX03_THREADS
#include <pthread.h>

int pthread_attr_getguardsize(const pthread_attr_t *__restrict__ attr,
                              size_t ** __restrict__ guardsize);
```

General description

pthread_attr_getguardsize() gets the guardsize attribute from *attr* and stores it into *guardsize*.

attr is a pointer to a thread attribute object initialized by pthread_attr_init().

The retrieved guardsize always matches the size stored by *pthread_attr_setguardsize()*, despite internal adjustments for rounding to multiples of the PAGESIZE system variable.

Returned value

If successful, *pthread_attr_getguardsize()* returns 0; otherwise it returns an error number.

Error Number
Description

EINVAL
The value specified by *attr* does not refer to an initialized thread attribute object.

Example

```
/* CELEBP62 */
/* Example using SUSv3 pthread_attr_getguardsize() interface */

#define _XOPEN_SOURCE 600

#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <limits.h>
#include <errno.h>


int main(void)
{
   pthread_attr_t attr;
   int            rc;
   size_t         guardsize;

   if (pthread_attr_init(&attr) == -1) {
      perror("error in pthread_attr_init");
      exit(1);
   }

   printf("Set guardsize to value of PAGESIZE.\n");
   rc = pthread_attr_setguardsize(&attr, PAGESIZE);
   if (rc != 0) {
      printf("pthread_attr_setguardsize returned: %d\n", rc);
      printf("Error: %d, Errno_Jr: %08x\n", errno, __errno2());
      exit(2);
   } else {
      printf("Set guardsize is %d\n", PAGESIZE);
   }

   rc = pthread_attr_getguardsize(&attr, &guardsize);
   if (rc != 0) {
      printf("pthread_attr_getguardsize returned: %d\n", rc);
      printf("Error: %d, Errno_Jr: %08x\n", errno, __errno2());
      exit(3);
   } else {
      printf("Retrieved guardsize is %d\n", guardsize);
   }

   rc = pthread_attr_destroy(&attr);
   if (rc != 0) {
      perror("error in pthread_attr_destroy");
      exit(4);
   }

   exit(0);
}
```

Related information

- “pthread.h” on page 58
- “pthread_attr_init() — Initialize a thread attribute object” on page 1218
- “pthread_attr_setguardsize() - Set guardsize attribute” on page 1222
- “pthread_attr_destroy() — Destroy the thread attributes object” on page 1204
- "Thread stack attributes" in the *z/OS XL C/C++ Programming Guide*

pthread_attr_getschedparam() - Get scheduling parameter attributes

Standards

Standards / Extensions	C or C++	Dependencies
Single UNIX Specification, Version 3	both	z/OS V1R9 POSIX(ON)

Format

```
#define _UNIX03_THREADS
#include <pthread.h>
#include <sched.h>

int pthread_attr_getschedparam(const pthread_attr_t *__restrict__ attr,
                               struct sched_param *__restrict__ param);
```

General description

pthread_attr_getschedparam() gets the scheduling priority attribute from *attr* and stores it into *param*.

attr is a pointer to a thread attribute object initialized by pthread_attr_init().

param points to a user-defined scheduling parameter object into which pthread_attr_getschedparam() copies the thread scheduling priority attribute.

Returned value

If successful, pthread_attr_getschedparam() returns 0; otherwise it returns an error number.

Error Number
Description

EINVAL
The value specified by attr does not refer to an initialized thread attribute object.

Example

```
/* CELEBP63 */
/* Example using SUSv3 pthread_attr_getschedparam() interface */

#define _UNIX03_THREADS 1
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <errno.h>

int main(void)
{
   pthread_attr_t attr;
   int             rc;
   struct sched_param param;

   param.sched_priority = 999;

   if (pthread_attr_init(&attr) == -1) {
      perror("error in pthread_attr_init");
      exit(1);
   }

   rc = pthread_attr_setschedparam(&attr, &param);
   if (rc != 0) {
      printf("pthread_attr_setschedparam returned: %d\n", rc);
      printf("Error: %d, Errno_Jr: %08x\n", errno, __errno2());
      exit(2);
   } else {
      printf("Set schedpriority to %d\n", param.sched_priority);
   }

   param.sched_priority = 0;

   rc = pthread_attr_getschedparam(&attr, &param);
   if (rc != 0) {
      printf("pthread_attr_getschedparam returned: %d\n", rc);
      printf("Error: %d, Errno_Jr: %08x\n", errno, __errno2());
      exit(3);
   } else {
      printf("Retrieved schedpriority of %d\n", param.sched_priority);
   }

   rc = pthread_attr_destroy(&attr);
   if (rc != 0) {
      perror("error in pthread_attr_destroy");
      exit(4);
```

```
    }

    exit(0);
}
```

Related information

- “pthread.h” on page 58
- “pthread_attr_init() — Initialize a thread attribute object” on page 1218
- “pthread_attr_setschedparam() - Set scheduling parameter attributes” on page 1224
- “pthread_attr_destroy() — Destroy the thread attributes object” on page 1204

pthread_attr_getstack() - Get stack attribute

Standards

Standards / Extensions	C or C++	Dependencies
Single UNIX Specification, Version 3	both	z/OS V1R9 POSIX(ON)

Format

```
#define _UNIX03_THREADS
#include <pthread.h>

int pthread_attr_getstack(const pthread_attr_t *__restrict__ attr,
  void ** __restrict__ addr, size_t * __restrict__ size);
```

General description

The *pthread_attr_getstack()* function gets both the base (lowest addressable) storage address and size of the initial stack segment from a thread attribute structure and stores them into addr and size respectively.

attr is a pointer to a thread attribute object initialized by pthread_attr_init().

addr is a pointer to the user-defined location where this function will place the base address of the initial stack segment.

size points to the user-defined location where this function will store the size of the initial stack segment.

Note: An XPLINK application uses two stacks, an upward-growing stack and a downward-growing stack. The "size" argument refers to the size of the downward-growing stack.

Returned value

If successful, *pthread_attr_getstack()* returns 0; otherwise it returns an error number.

Error Number
Description

EINVAL
The value specified by *attr* does not refer to an initialized thread attribute object.

Example

```
/* CELEBP69 */
/* Example using SUSv3 pthread_attr_getstack() interface */

#define _UNIX03_THREADS 1

#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <limits.h>
#include <errno.h>

int main(void)
{
   pthread_attr_t attr;
   int            rc;

   void  *mystack;
   size_t mystacksize = 2 * PTHREAD_STACK_MIN;

   if (pthread_attr_init(&attr) == -1) {
      perror("error in pthread_attr_init");
      exit(1);
   }

   /* Get a big enough stack and align it on 4K boundary. */
   mystack = malloc(PTHREAD_STACK_MIN * 3);
   if (mystack != NULL) {
      printf("Using PTHREAD_STACK_MIN to align stackaddr %x.\n", mystack);
      mystack = (void *)((((long)mystack + (PTHREAD_STACK_MIN - 1)) /
                          PTHREAD_STACK_MIN) * PTHREAD_STACK_MIN);
   } else {
      perror("Unable to acquire storage.");
      exit(2);
   }

   printf("Setting stackaddr to %x\n", mystack);
   printf("Setting stacksize to %x\n", mystacksize);
   rc = pthread_attr_setstack(&attr, mystack, mystacksize);
   if (rc != 0) {
      printf("pthread_attr_setstack returned: %d\n", rc);
      printf("Error: %d, Errno_Jr: %08x\n", errno, __errno2());
      exit(3);
   } else {
      printf("Set stackaddr to %x\n", mystack);
      printf("Set stacksize to %x\n", mystacksize);
   }

   rc = pthread_attr_destroy(&attr);
   if (rc != 0) {
      perror("error in pthread_attr_destroy");
      printf("Returned: %d, Error: %d\n", rc, errno);
      printf("Errno_Jr: %x\n", __errno2());
      exit(4);
   }

   exit(0);
}
```

Related information

- “pthread.h” on page 58
- “pthread_attr_init() — Initialize a thread attribute object” on page 1218
- “pthread_attr_setstack() - Set stack attribute” on page 1225
- “pthread_attr_destroy() — Destroy the thread attributes object” on page 1204
- "Thread stack attributes" in the *z/OS XL C/C++ Programming Guide*

pthread_attr_getstackaddr() - Get stackaddr attribute

Standards

Standards / Extensions	C or C++	Dependencies
Single UNIX Specification, Version 3	both	z/OS V1R9 POSIX(ON)

Format

```
#define _UNIX03_THREADS
#include <pthread.h>

int pthread_attr_getstackaddr(const pthread_attr_t *__restrict__ attr,
                              void ** __restrict__ addr);
```

General description

The *pthread_attr_getstackaddr()* function gets the stackaddr attribute from *attr* and stores it into *addr*. The stackaddr attribute holds the storage location of the created thread's initial stack segment.

attr is a pointer to a thread attribute object initialized by pthread_attr_init().

Note: The *pthread_attr_getstackaddr*() function is provided for historical reasons and is marked obsolescent in the Single UNIX Specification, Version 3 (SUSv3). New applications should use the newer function *pthread_attr_getstack()*, which provides functionality compatible with the SUSv3 standard.

Returned value

If successful, *pthread_attr_getstackaddr()* returns 0; otherwise it returns an error number.

Error Number
Description

EINVAL
The value specified by *attr* does not refer to an initialized thread attribute object.

Example

```
/* CELEBP64 */
/* Example using SUSv3 pthread_attr_getstackaddr() interface */

#define _UNIX03_THREADS 1

#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <limits.h>
#include <errno.h>

int main(void)
{
   pthread_attr_t attr;
   int            rc;
   void  *stackaddr;
   void  *mystack;
   if (pthread_attr_init(&attr) == -1) {
      perror("error in pthread_attr_init");
      exit(1);
   }

   /* Get a big enough stack and align it on 4K boundary. */
   mystack = malloc(PTHREAD_STACK_MIN * 2);
   if (mystack != NULL) {
      printf("Using PTHREAD_STACK_MIN to align stackaddr %x.\n", mystack);
      mystack = (void *)((((long)mystack + (PTHREAD_STACK_MIN - 1)) /
                          PTHREAD_STACK_MIN) * PTHREAD_STACK_MIN);
   } else {
      perror("Unable to acquire storage.");
      exit(2);
   }

   printf("Setting stackaddr to %x\n", mystack);
   rc = pthread_attr_setstackaddr(&attr, mystack);
   if (rc != 0) {
      printf("pthread_attr_setstackaddr returned: %d\n", rc);
      printf("Error: %d, Errno_Jr: %08x\n", errno, __errno2());
      exit(3);
   } else {
      printf("Set stackaddr to %x\n", mystack);
   }

   rc = pthread_attr_getstackaddr(&attr, &stackaddr);
   if (rc != 0) {
      printf("pthread_attr_getstackaddr returned: %d\n", rc);
      printf("Error: %d, Errno_Jr: %08x\n", errno, __errno2());
      exit(4);
   } else {
      printf("Retrieved stackaddr is %x\n", stackaddr);
   }

   rc = pthread_attr_destroy(&attr);
   if (rc != 0) {
      perror("error in pthread_attr_destroy");
      printf("Returned: %d, Error: %d\n", rc, errno);
      printf("Errno_Jr: %x\n", __errno2());
      exit(5);
   }

   exit(0);
}
```

Related information

- “pthread.h” on page 58
- “pthread_attr_init() — Initialize a thread attribute object” on page 1218
- “pthread_attr_setstackaddr() - Set stackaddr attribute” on page 1227
- “pthread_attr_destroy() — Destroy the thread attributes object” on page 1204
- "Thread stack attributes" in the *z/OS XL C/C++ Programming Guide*

pthread_attr_getstacksize() — Get the thread attribute stacksize object

Standards

Standards / Extensions	C or C++	Dependencies
POSIX.4a Single UNIX Specification, Version 3	both	POSIX(ON)

Format

```
#define _OPEN_THREADS
#include <pthread.h>

int pthread_attr_getstacksize(pthread_attr_t *attr, size_t *stacksize);
```

SUSV3

```
#define _UNIX03_THREADS
#include <pthread.h>
int pthread_attr_getstacksize(const pthread_attr_t * __restrict__attr,
                              size_t * __restrict__stacksize);
```

General description

Gets the value, in bytes, of the `stacksize` attribute for the thread attribute object, *attr*, that is created by pthread_attr_init(). This function returns the value in the variable pointed to by *stacksize*.

You can use a thread attribute object to manage the characteristics of threads in your application. It defines the set of values to be used for the thread during its creation. By establishing a thread attribute object, you can create many threads with the same set of characteristics, without defining those characteristics for each thread. You can define more than one thread attribute object.

Note: An XPLINK application uses two stacks, an upward-growing stack and a downward-growing stack. The "stacksize" refers to the size of the downward-growing stack.

Returned value

If successful, pthread_attr_getstacksize() returns 0 and stores the `stacksize` attribute value in *stacksize*.

If unsuccessful, pthread_attr_getstacksize() returns -1.

There are no documented errno values. Use perror() or strerror() to determine the cause of the error.

Special behavior for Single UNIX Specification, Version 3: If unsuccessful, pthread_attr_getstacksize() returns an error number to indicate the error.

Example

CELEBP08

```
/* CELEBP08 */
#define _OPEN_THREADS
#include <stdio.h>
#include <pthread.h>

int main()
{
   pthread_attr_t attr;
   size_t size;

   if (pthread_attr_init(&attr) == -1) {
      perror("error in pthread_attr_init");
      exit(1);
   }
   if (pthread_attr_getstacksize(&attr, &size) == -1) {
     perror("error in pthread_attr_getstackstate()");
     exit(2);
   }
   printf("The stack size is %d.\n", (int) size);

   if (pthread_attr_destroy(&attr) == -1) {
      perror("error in pthread_attr_destroy");
      exit(2);
   }
   exit(0);
}
```

Output:

```
The stack size is 524288.
```

Related information

- "pthread.h" on page 58
- "pthread_attr_init() — Initialize a thread attribute object" on page 1218
- "pthread_attr_setstacksize() — Set the stacksize attribute object" on page 1230
- "pthread_create() — Create a thread" on page 1261

pthread_attr_getsynctype_np() — Get thread sync type

Standards

Standards / Extensions	C or C++	Dependencies
z/OS UNIX	both	POSIX(ON)

Format

```
#define  _OPEN_SYS
#include <pthread.h>

int pthread_attr_getsynctype_np(pthread_attr_t *attr);
```

General description

The pthread_attr_getsynctype_np function returns the current synctype setting of the *attr* thread attribute object.

The *synctype* can be set to one of the following symbolics, as defined in the pthread.h header file:

__PTATSYNCHRONOUS
Can only create as many threads as TCBs available (or as many threads are available, depending on which number is smaller).

__PTATASYNCHRONOUS
Allows threads to be queued, that is, can create more threads than TCBs are available up to limit of how many threads are available. The queued threads will be released as TCBs become available.

Returned value

If successful, pthread_attr_getsynctype_np() returns the synctype value of the thread attribute object.

If unsuccessful, pthread_attr_getsynctype_np() returns -1.

There are no documented errno values. Use perror() or strerror() to determine cause of the error.

Related information

- “pthread.h” on page 58
- “pthread_attr_init() — Initialize a thread attribute object” on page 1218
- “pthread_attr_setsynctype_np() — Set thread sync type” on page 1231

pthread_attr_getweight_np() — Get weight of thread attribute object

Standards

Standards / Extensions	C or C++	Dependencies
z/OS UNIX	both	POSIX(ON)

Format

```
#define _OPEN_THREADS
#define _OPEN_SYS
#include <pthread.h>

int pthread_attr_getweight_np(pthread_attr_t *attr);
```

General description

Obtains the current weight of the thread setting of the thread attributes object, *attr*. The symbols for weight are defined in the pthread.h include file. The following weights are supported:

__MEDIUM_WEIGHT
The executing task can be reused when the thread exits.

__HEAVY_WEIGHT
When this exits, the associated MVS task can no longer request threads to process.

You can use a thread attribute object to manage the characteristics of threads in your application. It defines the set of values to be used for the thread during its creation. By establishing a thread attribute object, you can create many threads with the same set of characteristics, without defining those characteristics for each and every thread. You can define more than one thread attribute object.

Returned value

If successful, pthread_attr_getweight_np() returns the value of the weight of the thread attribute.

If unsuccessful, pthread_attr_getweight_np() returns -1.

There are no documented errno values. Use perror() or strerror() to determine the cause of the error.

Example

CELEBP09

```
/* CELEBP08 */
#define _OPEN_THREADS
#define _OPEN_SYS          /* Needed to identify __HEAVY_WEIGHT AND
```

```
                                 __MEDIUM WEIGHT  */
#include <stdio.h>
#include <pthread.h>

int main()
{
   pthread_attr_t attr;
   char           weight[12];

   if (pthread_attr_init(&attr) == -1) {
      perror("error in pthread_attr_init");
      exit(1);
   }

   switch(pthread_attr_getweight_np(&attr)) {
      default:
         perror("error in pthread_attr_getweight_np()");
         exit(2);
      case __HEAVY_WEIGHT:
         strcpy(weight, "heavy");
         break;
      case __MEDIUM_WEIGHT:
         strcpy(weight, "medium");
   }
   printf("The thread weight is %s.\n", weight);

   if (pthread_attr_destroy(&attr) == -1) {
      perror("error in pthread_attr_destroy");
      exit(2);
   }
   exit(0);
}
```

Output:

```
The thread weight is heavy.
```

Related information

- "pthread.h" on page 58
- "pthread_attr_init() — Initialize a thread attribute object"
- "pthread_attr_setweight_np() — Set weight of thread attribute object" on page 1232
- "pthread_create() — Create a thread" on page 1261

pthread_attr_init() — Initialize a thread attribute object

Standards

Standards / Extensions	C or C++	Dependencies
POSIX.4a Single UNIX Specification, Version 3	both	POSIX(ON)

Format

```
#define _OPEN_THREADS
#include <pthread.h>

int pthread_attr_init(pthread_attr_t *attr);
```

SUSV3:

```
#define _UNIX03_THREADS
#include <pthread.h>

int pthread_attr_init(pthread_attr_t *attr);
```

General description

Initializes *attr* with the default thread attributes, whose defaults are:

stacksize
Inherited from the STACK runtime option

detachstate
Undetached

synch Synchronous

weight
Heavy

Using a thread attribute object, you can manage the characteristics of threads in your application. It defines the set of values to be used for the thread during its creation. By establishing a thread attribute object, you can create many threads with the same set of characteristics, without defining those characteristics for each thread. You can define more than one thread attribute object. All threads are of equal priority.

If a thread attribute object is shared between threads, the application must provide the necessary synchronization because a thread attribute object is defined in the application's storage.

Note: An XPLINK application uses two stacks, an upward-growing stack and a downward-growing stack. "stacksize" always refers to the size of the upward-growing stack. The size of the downward-growing stack is inherited from the THREADSTACK runtime option.

Returned value

If successful, pthread_attr_init() returns 0.

If unsuccessful, pthread_attr_init() returns -1 and sets errno to one of the following values:

Error Code
Description

ENOMEM
Not enough memory is available to create the thread attribute object.

Special behavior for Single UNIX Specification, Version 3: If unsuccessful, pthread_attr_init() returns an error number to indicate the error.

Example

CELEBP10

```
/* CELEBP10 */
#define _OPEN_THREADS
#include <stdio.h>
#include <pthread.h>
```

```
void *thread1(void *arg)
{
   printf("hello from the thread\n");
   pthread_exit(NULL);
}

int main()
{
   int            rc, stat;
   pthread_attr_t attr;
   pthread_t      thid;

   rc = pthread_attr_init(&attr);
   if (rc == -1) {
      perror("error in pthread_attr_init");
      exit(1);
   }

   rc = pthread_create(&thid, &attr, thread1, NULL);
   if (rc == -1) {
      perror("error in pthread_create");
      exit(2);
   }

   rc = pthread_join(thid, (void *)&stat);
   exit(0);
}
```

Related information

- “pthread.h” on page 58
- “pthread_attr_destroy() — Destroy the thread attributes object” on page 1204
- “pthread_create() — Create a thread” on page 1261

pthread_attr_setdetachstate() — Set the detach state attribute

Standards

Standards / Extensions	C or C++	Dependencies
POSIX.4a Single UNIX Specification, Version 3	both	POSIX(ON)

Format

```
#define _OPEN_THREADS
#include <pthread.h>

int pthread_attr_setdetachstate(pthread_attr_t *attr, int *detachstate);
```

SUSV3:

```
#define _UNIX03_THREADS
#include <pthread.h>
int pthread_attr_setdetachstate(pthread_attr_t *attr, int detachstate);
```

General description

Alters the current *detachstate* setting of a thread attributes object, which can be set to *PTHREAD_CREATE_JOINABLE* or *PTHREAD_CREATE_DETACHED*.

0 Causes all the threads created with *attr* to be in an undetached state. An undetached thread will keep its resources after termination.

1 Causes all the threads created with *attr* to be in a detached state. A detached thread will have its resources automatically freed by the system at termination. Thus, you cannot get the thread's termination status, or wait for the thread to terminate by using pthread_join().

You can use a thread attribute object to manage the characteristics of threads in your application. It defines the set of values to be used for the thread during its creation. By establishing a thread attribute object, you can create many threads with the same set of characteristics, without defining those characteristics for each thread. You can define more than one thread attribute object.

Returned value

If successful, pthread_attr_setdetachstate() returns 0.

If unsuccessful, pthread_attr_setdetachstate() returns -1.

Error Code
Description

EINVAL
The value of detachstate was not valid or the value specified by attr does not refer to an initialized thread attribute object.

Special behavior for Single UNIX Specification, Version 3: If unsuccessful, pthread_attr_setdetachstate() returns an error number to indicate the error.

Example

CELEBP11

```
/* CELEBP11 */
#define _OPEN_THREADS
#include <stdio.h>
#include <pthread.h>

void **stat;
void *thread1(void *arg)
{
   printf("hello from the thread\n");
   pthread_exit((void *)0);
}

int main()
{
   int            ds, rc;
   size_t         s1;
   pthread_attr_t attr;
   pthread_t      thid;

   rc = pthread_attr_init(&attr);
   if (rc == -1) {
      perror("error in pthread_attr_init");
      exit(1);
   }

   ds = 0;
   rc = pthread_attr_setdetachstate(&attr, &ds);
   if (rc == -1) {
      perror("error in pthread_attr_setdetachstate");
      exit(2);
   }

   rc = pthread_create(&thid, &attr, thread1, NULL);
```

```
    if (rc == -1) {
       perror("error in pthread_create");
       exit(3);
    }

    rc = pthread_join(thid, stat);
    exit(0);
}
```

Related information

- "pthread.h" on page 58
- "pthread_attr_getdetachstate() — Get the detach state attribute" on page 1205
- "pthread_attr_init() — Initialize a thread attribute object" on page 1218
- "pthread_create() — Create a thread" on page 1261

pthread_attr_setguardsize() - Set guardsize attribute

Standards

Standards / Extensions	C or C++	Dependencies
Single UNIX Specification, Version 3	both	z/OS V1R9 POSIX(ON)

Format

```
#define _UNIX03_THREADS
#include <pthread.h>

int pthread_attr_setguardsize(pthread_attr_t *attr, size_t guardsize);
```

General description

pthread_attr_setguardsize() sets the guardsize attribute in *attr* using the value of guardsize.

attr is a pointer to a thread attribute object initialized by pthread_attr_init().

This function stores the guardsize attribute in the thread attribute object for subsequent calls to *pthread_attr_getguardsize()*, but no further action is taken. The guardsize attribute is ignored during thread creation.

Note:

The Single UNIX Specification, Version 3 expects a guard area of at least guardsize bytes, but permits the rounding of guardsize to a multiple of the system variable, PAGESIZE. Stack management in z/OS UNIX sets a guard area of PAGESIZE in 31-bit applications and of (PAGESIZE * PAGESIZE) bytes in AMODE64. These values are the default for the guard size attribute. Requests for larger guard areas will fail with EINVAL.

A zero guardsize requests that no guard area be provided. However, z/OS UNIX stack management requires a guard area. Therefore, this request cannot be satisfied, although pthread_attr_setguardsize() will tolerate a guardsize of zero.

Returned value

If successful, *pthread_attr_setguardsize()* returns 0; otherwise, it returns an error number.

Error Code
Description

EINVAL
The parameter *guardsize* is not valid or the value specified by *attr* does not refer to an initialized thread attribute object.

Example

```
/* CELEBP66 */
/* Example using SUSv3 pthread_attr_setguardsize() interface */

#define _XOPEN_SOURCE 600

#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <limits.h>
#include <errno.h>


int main(void)
{
   pthread_attr_t attr;
   int            rc;

   if (pthread_attr_init(&attr) == -1) {
      perror("error in pthread_attr_init");
      exit(1);
   }

   printf("Set guardsize to value of PAGESIZE.\n");
   rc = pthread_attr_setguardsize(&attr, PAGESIZE);
   if (rc != 0) {
      printf("pthread_attr_setguardsize returned: %d\n", rc);
      printf("Error: %d, Errno_Jr: %08x\n", errno, __errno2());
      exit(2);
   } else {
      printf("Set guardsize is %d\n", PAGESIZE);
   }

   rc = pthread_attr_destroy(&attr);
   if (rc != 0) {
      perror("error in pthread_attr_destroy");
      exit(3);
   }

   exit(0);
}
```

Related information

- “pthread.h” on page 58
- “pthread_attr_destroy() — Destroy the thread attributes object” on page 1204
- “pthread_attr_init() — Initialize a thread attribute object” on page 1218
- “pthread_attr_getguardsize() - Get guardsize attribute” on page 1208
- "Thread stack attributes" in the *z/OS XL C/C++ Programming Guide*.

pthread_attr_setschedparam() - Set scheduling parameter attributes

Standards

Standards / Extensions	C or C++	Dependencies
Single UNIX Specification, Version 3	both	z/OS V1R9 POSIX(ON)

Format

```
#define _UNIX03_THREADS
#include <pthread.h>
#include <sched.h>

int pthread_attr_setschedparam(pthread_attr_t *__restrict__ attr,
                    const struct sched_param *__restrict__ param);
```

General description

pthread_attr_setschedparam() sets the scheduling priority attribute in *attr* using the value from *param*.

attr is a pointer to a thread attribute object initialized by pthread_attr_init().

param points to a user-defined scheduling parameter object used by *pthread_attr_setschedparam()* as the source of the thread scheduling priority attribute to set in *attr*. The scheduling priority member of a sched_param structure is declared as an int.

If successful, the sched_priority from *param* is available for subsequent calls to the pthread_getschedparam() function. However, z/OS UNIX takes no other action based on the value of the scheduling priority stored in *attr*.

Returned value

If successful, *pthread_attr_setschedparam()* returns 0; otherwise, it returns an error number.

Error Code
Description

EINVAL
The value specified by *attr* does not refer to an initialized thread attribute object.

Example

```
/* CELEBP67 */
/* Example using SUSv3 pthread_attr_setschedparam() interface */

#define _UNIX03_THREADS 1
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <errno.h>

int main(void)
{
   pthread_attr_t attr;
   int            rc;
   struct sched_param param;
   param.sched_priority = 999;

   if (pthread_attr_init(&attr) == -1) {
      perror("error in pthread_attr_init");
      exit(1);
   }

   rc = pthread_attr_setschedparam(&attr, &param);
   if (rc != 0) {
      printf("pthread_attr_setschedparam returned: %d\n", rc);
      printf("Error: %d, Errno_Jr: %08x\n", errno, __errno2());
      exit(2);
   } else {
      printf("Set schedpriority to %d\n", param.sched_priority);
   }

   rc = pthread_attr_destroy(&attr);
   if (rc != 0) {
      perror("error in pthread_attr_destroy");
      exit(3);
   }

   exit(0);
}
```

Related information

- "pthread.h" on page 58
- "sched.h" on page 62
- "pthread_attr_destroy() — Destroy the thread attributes object" on page 1204
- "pthread_attr_init() — Initialize a thread attribute object" on page 1218
- "pthread_attr_getschedparam() - Get scheduling parameter attributes" on page 1209

pthread_attr_setstack() - Set stack attribute

Standards

Standards / Extensions	C or C++	Dependencies
Single UNIX Specification, Version 3	both	z/OS V1R9 POSIX(ON)

Format

```
#define _UNIX03_THREADS
#include <pthread.h>

int pthread_attr_setstack(pthread_attr_t *attr, void *addr, size_t size);
```

General description

The *pthread_attr_setstack()* function sets the *stackaddr* and *stacksize* attributes in *attr* from the values of *addr* and *size* respectively.

When a thread is created, the *stackaddr* attribute locates the base (lowest addressable byte) of the created thread's initial stack segment. The *stacksize* attribute is the size, in bytes, of the initial stack segment allocated for the thread.

attr is a pointer to a thread attribute object initialized by pthread_attr_init().

addr is the memory location to use for the initial stack segment and must be aligned appropriately to be used as a stack. For 31-bit applications, stack alignment is on a 4K boundary; in AMODE64, the alignment is on a one megabyte boundary.

size must be at least as large as PTHREAD_STACK_MIN. This constant is defined in the <limits.h> header.

The minimum stacksize in 31-bit is 4096 (4K) and in 64-bit 1048576 (1M). In addition, the system will allocate an equivalent-sized guardpage. There is no specified maximum stacksize. If more storage is requested than the system can satisfy at pthread creation, then pthread_create() will fail and return EINVAL.

Usage notes

1. An XPLINK application uses two stacks, an upward-growing stack and a downward-growing stack. The "size" argument refers to the size of the downward-growing stack.
2. The Language Environment storage report tolerates but does not maintain statistics on application-managed stacks. Also, the runtime storage option, suboption for dsa initialization does not support application-managed stacks.

Returned value

If successful, *pthread_attr_setstack()* returns 0; otherwise, it returns an error number.

Error Code
Description

EINVAL
Can be one of the following error conditions:

- size is less than PTHREAD_STACK_MIN
- addr does not have proper alignment to be used as a stack
- (addr + size) lacks proper alignment
- attr does not refer to an initialized thread attribute object.

Example

```
/* CELEBP65 */
/* Example using SUSv3 pthread_attr_setstack() interface */

#define _UNIX03_THREADS 1

#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <limits.h>
#include <errno.h>

int main(void)
{
   pthread_attr_t attr;
   int            rc;

   void * stackaddr;
   void  *mystack;

   size_t stacksize;
   size_t mystacksize = 2 * PTHREAD_STACK_MIN;

   if (pthread_attr_init(&attr) == -1) {
      perror("error in pthread_attr_init");
      exit(1);
   }

   /* Get a big enough stack and align it on 4K boundary. */
   mystack = malloc(PTHREAD_STACK_MIN * 3);
   if (mystack != NULL) {
      printf("Using PTHREAD_STACK_MIN to align stackaddr %x.\n", mystack);
      mystack = (void *)((((long)mystack + (PTHREAD_STACK_MIN - 1)) /
                         PTHREAD_STACK_MIN) * PTHREAD_STACK_MIN);
   } else {
      perror("Unable to acquire storage.");
      exit(2);
   }

   printf("Setting stackaddr to %x\n", mystack);
   printf("Setting stacksize to %x\n", mystacksize);
   rc = pthread_attr_setstack(&attr, mystack, mystacksize);
   if (rc != 0) {
      printf("pthread_attr_setstack returned: %d\n", rc);
      printf("Error: %d, Errno_Jr: %08x\n", errno, __errno2());
      exit(3);
   } else {
      printf("Set stackaddr to %x\n", mystack);
      printf("Set stacksize to %x\n", mystacksize);
   }

   rc = pthread_attr_getstack(&attr, &stackaddr, &stacksize);
   if (rc != 0) {
      printf("pthread_attr_getstack returned: %d\n", rc);
      printf("Error: %d, Errno_Jr: %08x\n", errno, __errno2());
      exit(4);
   } else {
      printf("Retrieved stackaddr is %x\n", mystack);
      printf("Retrieved stacksize is %x\n", mystacksize);
   }

   rc = pthread_attr_destroy(&attr);
   if (rc != 0) {
      perror("error in pthread_attr_destroy");
      printf("Returned: %d, Error: %d\n", rc, errno);
      printf("Errno_Jr: %x\n", __errno2());
      exit(5);
   }

   exit(0);
}
```

Related information

- “pthread.h” on page 58
- “pthread_attr_destroy() — Destroy the thread attributes object” on page 1204
- “pthread_attr_init() — Initialize a thread attribute object” on page 1218
- “pthread_attr_getstack() - Get stack attribute” on page 1211
- For a complete set of restrictions on *addr* and *size*, see the topic about thread stack attributes in *z/OS XL C/C++ Programming Guide*

pthread_attr_setstackaddr() - Set stackaddr attribute

Standards

Standards / Extensions	C or C++	Dependencies
Single UNIX Specification, Version 3	both	z/OS V1R9 POSIX(ON)

Format

```
#define _UNIX03_THREADS
#include <pthread.h>

int pthread_attr_setstackaddr(pthread_attr_t *attr, void *addr);
```

General description

The *pthread_attr_setstackaddr()* function sets the stackaddr attribute in *attr* using the value of *addr*.

attr is a pointer to a thread attribute object initialized by pthread_attr_init().

addr is the lowest addressable byte of the memory designated for use as the initial stack segment. It must have at least PTHREAD_STACK_MIN storage allocated. The PTHREAD_STACK_MIN constant is defined in <limits.h>. The addr value must also be aligned with the stack frame size, a multiple of 4K in 31-bit applications and one megabyte in AMODE 64.

The thread must have permission to read and write to all pages within the stack referenced by *addr*.

A stacksize is required at pthread creation. If the value is not present in the thread attribute object, the stacksize will default to PTHREAD_STACK_MIN. Subsequent calls to *pthread_attr_setstacksize()* can overwrite the stacksize prior to pthread creation.

Usage notes

1. The *pthread_attr_setstackaddr()* function is provided for historical reasons. It is marked obsolescent in the Single UNIX Specification, Version 3 (SUSv3). New applications should use the newer function *pthread_attr_setstack()*, which provides functionality compatible with the SUSv3 standard.
2. An attribute object with the stackaddr attribute set may not be used more than once, unless it is destroyed and reinitialized,or its stackaddr attribute changed. For more details, see "Thread stack attributes" in the z/OS X/L C/C++ Programming Guide.
3. An XPLINK application uses two stacks, an upward-growing stack and a downward-growing stack. The variable *addr* always refers to lowest addressable byte of the downward-growing stack.
4. The Language Environment storage report tolerates but does not maintain statistics on application-managed stacks. Also, the runtime storage option, suboption for dsa initialization does not support application-managed stacks.

Returned value

If successful, *pthread_attr_setstackaddr()* returns 0; otherwise, it returns an error number.

Error Code
Description

EINVAL
The value of *addr* does not have proper alignment to be used as a stack or the value specified by *attr* does not refer to an initialized thread attribute object.

Example

```
/* CELEBP68 */
/* Example using SUSv3 pthread_attr_setstackaddr() interface */

#define _UNIX03_THREADS 1

#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <limits.h>
#include <errno.h>

int main(void)
{
   pthread_attr_t attr;
   int             rc;
   void  *mystack;

   if (pthread_attr_init(&attr) == -1) {
      perror("error in pthread_attr_init");
      exit(1);
   }

   /* Get a big enough stack and align it on 4K boundary. */
   mystack = malloc(PTHREAD_STACK_MIN * 2);
   if (mystack != NULL) {
      printf("Using PTHREAD_STACK_MIN to align stackaddr %x.\n", mystack);
      mystack = (void *)((((long)mystack + (PTHREAD_STACK_MIN - 1)) /
                         PTHREAD_STACK_MIN) * PTHREAD_STACK_MIN);
   } else {
      perror("Unable to acquire storage.");
      exit(2);
   }

   printf("Setting stackaddr to %x\n", mystack);
   rc = pthread_attr_setstackaddr(&attr, mystack);
   if (rc != 0) {
      printf("pthread_attr_setstackaddr returned: %d\n", rc);
      printf("Error: %d, Errno_Jr: %08x\n", errno, __errno2());
      exit(3);
   } else {
      printf("Set stackaddr to %x\n", mystack);
   }

   rc = pthread_attr_destroy(&attr);
   if (rc != 0) {
      perror("error in pthread_attr_destroy");
      printf("Returned: %d, Error: %d\n", rc, errno);
      printf("Errno_Jr: %x\n", __errno2());
      exit(4);
   }

   exit(0);
}
```

Related information

- “pthread.h” on page 58
- “pthread_attr_destroy() — Destroy the thread attributes object” on page 1204
- “pthread_attr_init() — Initialize a thread attribute object” on page 1218
- “pthread_attr_getstackaddr() - Get stackaddr attribute” on page 1213
- "Thread stack attributes" in the *z/OS XL C/C++ Programming Guide*

pthread_attr_setstacksize() — Set the stacksize attribute object

Standards

Standards / Extensions	C or C++	Dependencies
POSIX.4a Single UNIX Specification, Version 3	both	POSIX(ON)

Format

```
#define _OPEN_THREADS
#include <pthread.h>

int pthread_attr_setstacksize(pthread_attr_t *attr, size_t stacksize);
```

SUSV3:

```
#define _UNIX03_THREADS
#include <pthread.h>

int pthread_attr_setstacksize(pthread_attr_t *attr, size_t stacksize);
```

General description

Sets the stacksize, in bytes, for the thread attribute object, *attr*. *stacksize* is the initial stack size. Other stack characteristics, like stack increment size, are inherited from the STACK64/THREADSTACK64 runtime option.

You can use a thread attribute object to manage the characteristics of threads in your application. It defines the set of values to be used for the thread during its creation. By establishing a thread attribute object, you can create many threads with the same set of characteristics, without defining those characteristics for each thread. You can define more than one thread attribute object.

Usage notes

1. An XPLINK application uses two stacks, an upward-growing stack and a downward-growing stack. The "stacksize" refers to the size of the downward-growing stack.
2. When using Single UNIX Specification, Version thread support, the minimum stacksize in 31-bit is 4096 (4K) and in 64-bit 1048576 (1M). In addition, the system will allocate an equivalent-sized guardpage. There is no specified maximum stacksize. If more storage is requested than the system can satisfy at pthread creation, then pthread_create() will fail and return EINVAL.

Returned value

If successful, pthread_attr_setstacksize() returns 0.

If unsuccessful, pthread_attr_setstacksize() returns -1.

Error Code
Description

EINVAL
The value of stacksize is less than PTHREAD_STACK_MIN, or the value specified by attr does not refer to an initialized thread attribute object.

Special behavior for Single UNIX Specification, Version 3:

If unsuccessful, pthread_attr_setstacksize() returns an error number to indicate the error.

Example

CELEBP12

```
/* CELEBP12 */
#define _OPEN_THREADS
#include <stdio.h>
#include <pthread.h>

void *thread1(void *arg)
{
   printf("hello from the thread\n");
   pthread_exit(NULL);
}

int main()
{
   int            rc, stat;
   size_t         s1;
   pthread_attr_t attr;
   pthread_t      thid;

   rc = pthread_attr_init(&attr);
   if (rc == -1) {
      perror("error in pthread_attr_init");
      exit(1);
   }

   s1 = 4096;
   rc = pthread_attr_setstacksize(&attr, s1);
   if (rc == -1) {
      perror("error in pthread_attr_setstacksize");
      exit(2);
   }

   rc = pthread_create(&thid, &attr, thread1, NULL);
   if (rc == -1) {
      perror("error in pthread_create");
      exit(3);
   }

   rc = pthread_join(thid, (void *)&stat);
   exit(0);
}
```

Related information

- “pthread.h” on page 58
- “pthread_attr_getstacksize() — Get the thread attribute stacksize object” on page 1214
- “pthread_attr_init() — Initialize a thread attribute object” on page 1218
- “pthread_create() — Create a thread” on page 1261

pthread_attr_setsynctype_np() — Set thread sync type

Standards

Standards / Extensions	C or C++	Dependencies
z/OS UNIX	both	POSIX(ON)

Format

```
#define  _OPEN_SYS
#include <pthread.h>

int pthread_attr_setsynctype_np(pthread_attr_t *attr, int synctype);
```

General description

The pthread_attr_setsynctype_np function allows you to alter the synctype setting of the *attr* thread attribute object.

The *synctype* can be set to one of the following symbolics, as defined in the pthread.h header file:

__PTATSYNCHRONOUS
Can only create as many threads as TCBs available (or as many threads are available, depending on which number is smaller).

__PTATASYNCHRONOUS
Allows threads to be queued, that is, can create more threads than TCBs are available up to limit of how many threads are available. The queued threads will be released as TCBs become available. While threads are on the queue, they can still be affected by other pthread functions.

Returned value

If successful, pthread_attr_setsynctype_np() returns 0.

If unsuccessful, pthread_attr_setsynctype_np() returns -1.

There are no documented errno values. Use perror() or strerror() to determine cause of the error.

Related information

- "pthread.h" on page 58
- "pthread_attr_getsynctype_np() — Get thread sync type" on page 1216
- "pthread_attr_init() — Initialize a thread attribute object" on page 1218
- "pthread_attr_setweight_np() — Set weight of thread attribute object"

pthread_attr_setweight_np() — Set weight of thread attribute object

Standards

Standards / Extensions	C or C++	Dependencies
z/OS UNIX	both	POSIX(ON)

Format

```
#define _OPEN_THREADS
#define _OPEN_SYS
#include <pthread.h>

int pthread_attr_setweight_np(pthread_attr_t *attr, int threadweight);
```

General description

Alter the current weight of the thread setting of the thread attribute object, *attr*.

threadweight can be set to one of the following two symbols for the weight of the thread, as defined in the pthread.h header file.

__LIGHT_WEIGHT
Not supported.

__MEDIUM_WEIGHT
Each thread runs on a task. Upon exiting, if another thread is not queued to run, the task waits for some other thread to issue a pthread_create(), and the thread then runs on that task. The thread is assumed to cleanup all resources it used.

__HEAVY_WEIGHT
The task is attached on pthread_create() and terminates upon a pthread_exit(). Full MVS EOT resource cleanup occurs when exiting. When this exits, the associated MVS task can no longer request threads to process.

You can use a thread attribute object to manage the characteristics of threads in your application. It defines the set of values to be used for the thread during its creation. By establishing a thread attribute object, you can create many threads with the same set of characteristics, without defining those characteristics for each thread. You can define more than one thread attribute object.

Returned value

If successful, pthread_attr_setweight_np() returns 0.

If unsuccessful, pthread_attr_setweight_np() returns -1.

There are no documented errno values. Use perror() or strerror() to determine the cause of the error.

Example

CELEBP13

```
/* CELEBP13 */
#define _OPEN_THREADS
#define _OPEN_SYS      /*  Needed to identify __MEDIUM_WEIGHT  */
#include <stdio.h>
#include <pthread.h>

void *thread1(void *arg)
{
   printf("hello from the thread\n");
   pthread_exit((void *)0);
}

int main()
{
   int            rc, stat;
   pthread_attr_t attr;
   pthread_t      thid;

   rc = pthread_attr_init(&attr);
   if (rc == -1) {
      perror("error in pthread_attr_init");
      exit(1);
   }

   rc = pthread_attr_setweight_np(&attr, __MEDIUM_WEIGHT);
   if (rc == -1) {
      perror("error in pthread_attr_setweight_np");
      exit(2);
   }

   rc = pthread_create(&thid, &attr, thread1, NULL);
   if (rc == -1) {
      perror("error in pthread_create");
      exit(3);
   }

   rc = pthread_join(thid, (void *)&stat);
   exit(0);
}
```

Related information

- "pthread.h" on page 58
- "pthread_attr_getweight_np() — Get weight of thread attribute object" on page 1217
- "pthread_attr_init() — Initialize a thread attribute object" on page 1218
- "pthread_create() — Create a thread" on page 1261

pthread_cancel() — Cancel a thread

Standards

Standards / Extensions	C or C++	Dependencies
POSIX.4a Single UNIX Specification, Version 3	both	POSIX(ON)

Format

```
#define _OPEN_THREADS
#include <pthread.h>

int pthread_cancel(pthread_t thread);
```

SUSV3:

```
#define _UNIX03_THREADS
#include <pthread.h>

int pthread_cancel(pthread_t thread);
```

General description

Requests that a thread be canceled. The thread to be canceled controls when this cancelation request is acted on through the cancelability state and type.

The cancelability states can be:

PTHREAD_INTR_DISABLE
The thread cannot be canceled.

PTHREAD_INTR_ENABLE
The thread can be canceled, but it is subject to type.

The cancelability types can be:

PTHREAD_INTR_CONTROLLED
The thread can be canceled, but only at specific points of execution:

- When waiting on a condition variable, which is pthread_cond_wait() or pthread_cond_timedwait()
- When waiting for the end of another thread, which is pthread_join()
- While waiting for an asynchronous signal, which is sigwait()
- Testing specifically for a cancel request, which is pthread_testintr()
- When suspended because of POSIX functions or one of the following C standard functions: close(), fcntl(), open() pause(), read(), tcdrain(), tcsetattr(), sigsuspend(), sigwait(), sleep(), wait(), or write()

PTHREAD_INTR_ASYNCHRONOUS
The thread can be canceled at any time.

A thread that is joined on a thread that is canceled has a status of -1 returned to it. For more information, refer to “pthread_join() — Wait for a thread to end” on page 1275.

pthread_t is the data type used to uniquely identify a thread. It is returned by pthread_create() and used by the application in function calls that require a thread identifier.

Note: A thread in mutex wait will not be interrupted by a signal, and therefore not canceled.

Special behavior for C++: Destructors for automatic objects on the stack will be run when a thread is canceled. The stack is unwound and the destructors are run in reverse order.

Special behavior for SUSv3 : Single UNIX Standard, Version 3 defines new symbols for cancelability state and type. These are equivalent to the symbols described above and must be used when compiling in the SUSv3 namespace. The symbols for state are PTHREAD_CANCEL_ENABLE and PTHREAD_CANCEL_DISABLE. Symbols for type are PTHREAD_CANCEL_DEFERRED and PTHREAD_CANCEL_ASYNCHRONOUS.

Returned value

If successful, pthread_cancel() returns 0. Success indicates that the pthread_cancel() request has been issued. The thread to be canceled may still execute because of its interruptibility state.

If unsuccessful, pthread_create() returns -1 and sets errno to one of the following values:

Error Code
Description

EINVAL
The specified thread is not valid.

ESRCH
The specified thread does not refer to a currently existing thread.

Special behavior for Single UNIX Specification, Version 3:

If unsuccessful, pthread_cancel() returns an error number to indicate the error.

Example

CELEBP14

```
/* CELEBP14 */
#define _OPEN_THREADS
#include <errno.h>
#include <pthread.h>
#include <stdio.h>
#include <string.h>
#include <unistd.h>

int  thstatus;

void * thread(void *arg)
{
 puts("thread has started. now sleeping");
 while (1)
   sleep(1);
}

main(int argc, char *argv[])
{
 pthread_t      thid;
 void           *status;

 if ( pthread_create(&thid, NULL, thread, NULL) != 0) {
   perror("pthread_create failed");
   exit(2);
 }

 if ( pthread_cancel(thid) == -1 ) {
   perror("pthread_cancel failed");
   exit(3);
 }

 if ( pthread_join(thid, &status)== -1 ) {
   perror("pthread_join failed");
   exit(4);
 }

 if ( status == (int *)-1 )
   puts("thread was cancelled");
 else
   puts("thread was not cancelled");

 exit(0);
}
```

Output:

```
thread has started. now sleeping
thread was canceled
```

Related information

- "pthread.h" on page 58
- "pthread_cond_timedwait() — Wait on a condition variable" on page 1246
- "pthread_cond_wait() — Wait on a condition variable" on page 1249
- "pthread_exit() — Exit a thread" on page 1267
- "pthread_join() — Wait for a thread to end" on page 1275
- "pthread_setcancelstate() — Set a thread cancelability state format" on page 1328
- "pthread_setcanceltype() — Set a thread cancelability type format" on page 1329
- "pthread_setintr() — Set a thread cancelability state" on page 1330
- "pthread_setintrtype() — Set a thread cancelability type" on page 1333
- "pthread_testcancel() — Establish a cancelation point" on page 1342

- "pthread_testintr() — Establish a cancelability point" on page 1342

pthread_cleanup_pop() — Remove a cleanup handler

Standards

Standards / Extensions	C or C++	Dependencies
POSIX.4a Single UNIX Specification, Version 3	both	POSIX(ON)

Format

```
#define _OPEN_THREADS
#include <pthread.h>

void pthread_cleanup_pop(int execute);
```

General description

Removes the specified *routine* in the last executed pthread_cleanup_push() statement from the top of the calling thread's cleanup stack.

The *execute* parameter specifies whether the cleanup routine that is popped should be run or just discarded. If the value is nonzero, the cleanup routine is executed.

pthread_cleanup_push() and pthread_cleanup_pop() must appear in pairs in the program within the same lexical scope, or undefined behavior will result.

When the thread ends, all pushed but not yet popped cleanup routines are popped from the cleanup stack and executed in last-in-first-out (LIFO) order. This occurs when the thread:

- Calls pthread_exit()
- Does a return from the start routine (that gets controls as a result of a pthread_create())
- Is canceled because of a pthread_cancel()

Returned value

pthread_cleanup_pop() returns no values.

This function is used as a statement.

If an error occurs while a pthread_cleanup_pop() statement is being processed, a termination condition is raised.

There are no documented errno values. Use perror() or strerror() to determine the cause of an error.

Example

CELEBP15

```
/* CELEBP15 */
#define _OPEN_THREADS
#include <pthread.h>
#include <stdio.h>
```

```
int iteration;

void noise_maker(void *arg) {
  printf("hello from noise_maker in iteration %d!\n", iteration);
}

void *thread(void *arg) {
  pthread_cleanup_push(noise_maker, NULL);
  pthread_cleanup_pop(iteration == 1 ? 0 : 1);
}

main() {
  pthread_t thid;
  void * ret;

  for (iteration=1; iteration<=2; iteration++) {

    if (pthread_create(&thid, NULL, thread, NULL) != 0) {
      perror("pthread_create() error");
      exit(1);
    }

    if (pthread_join(thid, &ret) != 0){
      perror("pthread_join() error");
      exit(2);
    }
/*
    if (pthread_detach(&thid) != 0) {
      perror("pthread_detach() error");
      exit(3);
    }
 */
  }
}
```

Output:

```
hello from noise_maker in iteration 2!
```

Related information

- "pthread.h" on page 58
- "pthread_cancel() — Cancel a thread" on page 1234
- "pthread_cleanup_push() — Establish a cleanup handler"
- "pthread_exit() — Exit a thread" on page 1267

pthread_cleanup_push() — Establish a cleanup handler

Standards

Standards / Extensions	C or C++	Dependencies
POSIX.4a Single UNIX Specification, Version 3	both	POSIX(ON)

Format

```
#define _OPEN_THREADS
#include <pthread.h>

void pthread_cleanup_push(void (*routine)(void *arg), void *arg);
```

General description

Pushes the specified *routine* onto the calling thread's cleanup stack. The cleanup handler is executed as a result of a pthread_cleanup_pop(), with a nonzero value for the *execute* parameter.

When the thread ends, all pushed but not yet popped cleanup routines are popped from the cleanup stack and executed in last-in-first-out (LIFO) order. This occurs when the thread:

- Calls pthread_exit()
- Does a return from the start routine
- Is canceled because of a pthread_cancel()

pthread_cleanup_push() and pthread_cleanup_pop() must appear in pairs and within the same lexical scope, or undefined behavior will result.

Special behavior for C++: Because C and C++ linkage conventions are incompatible, pthread_cleanup_push() cannot receive a C++ function pointer as the start routine function pointer If you attempt to pass a C++ function pointer to pthread_cleanup_push(), the compiler will flag it as an error. You can pass a C or C++ function to pthread_cleanup_push() by declaring it as extern "C".

Returned value

pthread_cleanup_push() returns no values.

This function is used as a statement.

If an error occurs while a pthread_cleanup_push() statement is being processed, a termination condition is raised.

There are no documented errno values. Use perror() or strerror() to determine the cause of an error.

Example

CELEBP16

```
/* CELEBP16 */
#define _OPEN_THREADS

#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

int footprint=0;

void *thread(void *arg) {
  char *storage;

  if ((storage = (char*) malloc(80)) == NULL) {
    perror("malloc() failed");
    exit(6);
  }

  /* Plan to release storage even if thread doesn't exit normally */

  pthread_cleanup_push(free, storage);
  puts("thread has obtained storage and is waiting to be cancelled");
  footprint++;
  while (1)
    sleep(1);

  pthread_cleanup_pop(1);
}

main() {
  pthread_t thid;

  if (pthread_create(&thid, NULL, thread, NULL) != 0) {
    perror("pthread_create() error");
    exit(1);
  }

  while (footprint == 0)
    sleep(1);

  puts("IPT is cancelling thread");

  if (pthread_cancel(thid) != 0) {
    perror("pthread_cancel() error");
    exit(3);
  }

  if (pthread_join(thid, NULL) != 0) {
    perror("pthread_join() error");
    exit(4);
  }
}
```

Output:

```
thread has obtained storage and is waiting to be canceled
IPT is canceling thread
```

Related information

- "pthread.h" on page 58
- "pthread_cancel() — Cancel a thread" on page 1234
- "pthread_cleanup_pop() — Remove a cleanup handler" on page 1237
- "pthread_exit() — Exit a thread" on page 1267

pthread_cond_broadcast() — Broadcast a condition

Standards

Standards / Extensions	C or C++	Dependencies
POSIX.4a Single UNIX Specification, Version 3	both	POSIX(ON)

Format

```
#define _OPEN_THREADS
#include <pthread.h>

int pthread_cond_broadcast(pthread_cond_t *cond);
```

SUSV3:

```
#define _UNIX03_THREADS
#include <pthread.h>

int pthread_cond_broadcast(pthread_cond_t *cond);
```

General description

Unblock all threads that are blocked on the specified condition variable, *cond*. If more than one thread is blocked, the order in which the threads are unblocked is unspecified.

pthread_cond_broadcast() has no effect if there are no threads currently blocked on *cond*.

Returned value

If successful, pthread_cond_broadcast() returns 0.

If unsuccessful, pthread_cond_broadcast() returns -1 and sets errno to one of the following values:

Error Code
Description

EINVAL
The value specified by *cond* does not refer to an initialized condition variable.

Special behavior for Single UNIX Specification, Version 3:

If unsuccessful, pthread_cond_broadcast() returns an error number to indicate the error.

Example

CELEBP17

```
/* CELEBP17 */
#define _OPEN_THREADS
#include <pthread.h>
#include <stdio.h>

main() {
  pthread_cond_t cond;

  if (pthread_cond_init(&cond, NULL) != 0) {
    perror("pthread_cond_init() error");
    exit(1);
  }

  if (pthread_cond_broadcast(&cond) != 0) {
    perror("pthread_cond_broadcast() error");
    exit(2);
  }

  if (pthread_cond_destroy(&cond) != 0) {
    perror("pthread_cond_destroy() error");
    exit(3);
  }
}
```

Related information

- “pthread.h” on page 58
- “pthread_cond_init() — Initialize a condition variable” on page 1243
- “pthread_cond_signal() — Signal a condition” on page 1245
- “pthread_cond_timedwait() — Wait on a condition variable” on page 1246

- “pthread_cond_wait() — Wait on a condition variable” on page 1249

pthread_cond_destroy() — Destroy the condition variable object

Standards

Standards / Extensions	C or C++	Dependencies
POSIX.4a Single UNIX Specification, Version 3	both	POSIX(ON)

Format

```
#define _OPEN_THREADS
#include <pthread.h>

int pthread_cond_destroy(pthread_cond_t *cond);
```

SUSV3:

```
#define _UNIX03_THREADS
#include <pthread.h>

int pthread_cond_destroy(pthread_cond_t *cond);
```

General description

Destroys the condition variable object specified by *cond*.

A condition variable object identifies a condition variable. Condition variables are used in conjunction with mutexes to protect shared resources.

Returned value

If successful, pthread_cond_destroy() returns 0.

If unsuccessful, pthread_cond_destroy() returns -1 and sets errno to one of the following values:

Error Code
Description

EBUSY
An attempt was made to destroy the object referenced by *cond* while it is referenced by another thread.

EINVAL
The value specified by *cond* is not valid.

Special behavior for Single UNIX Specification, Version 3: If unsuccessful, pthread_cond_destroy() returns an error number to indicate the error.

Example

CELEBP18

```
/* CELEBP18 */
#define _OPEN_THREADS
#include <pthread.h>
#include <stdio.h>

main() {
  pthread_cond_t cond;

  if (pthread_cond_init(&cond, NULL) != 0) {
    perror("pthread_cond_init() error");
    exit(1);
  }

  if (pthread_cond_destroy(&cond) != 0) {
    perror("pthread_cond_destroy() error");
    exit(2);
  }
}
```

Related information

- "pthread.h" on page 58
- "pthread_cond_broadcast() — Broadcast a condition" on page 1240
- "pthread_cond_init() — Initialize a condition variable"
- "pthread_cond_signal() — Signal a condition" on page 1245
- "pthread_cond_timedwait() — Wait on a condition variable" on page 1246
- "pthread_cond_wait() — Wait on a condition variable" on page 1249

pthread_cond_init() — Initialize a condition variable

Standards

Standards / Extensions	C or C++	Dependencies
POSIX.4a Single UNIX Specification, Version 3	both	POSIX(ON)

Format

```
#define _OPEN_THREADS
#include <pthread.h>

int pthread_cond_init(pthread_cond_t *cond, pthread_condattr_t *attr);
```

SUSV3:

```
#define _UNIX03_THREADS
#include <pthread.h>
int pthread_cond_init(pthread_cond_t * __restrict__cond,
                      pthread_condattr_t * __restrict__attr);
```

General description

Initializes the condition variable referenced by *cond* with attributes referenced by *attr*. If *attr* is NULL, the default condition variable attributes are used.

Returned value

If successful, pthread_cond_init() returns 0.

If unsuccessful, pthread_cond_init() returns -1 and sets errno to one of the following values:

Error Code
: **Description**

ENOMEM
: There is not enough memory to initialize the condition variable.

EAGAIN
The system lacked the necessary resources (other than memory) to initialize another condition variable.

EBUSY
The implementation has detected an attempt to reinitialize the object referenced by cond, a previously initialized, but not yet destroyed, condition variable.

EINVAL
The value specified by attr is invalid.

Special behavior for Single UNIX Specification, Version 3: If unsuccessful, pthread_cond_init() returns an error number to indicate the error.

Usage notes

The _OPEN_SYS_MUTEX_EXT feature switch can be optionally included. If the feature is set, then significantly larger pthread_cond_t objects will be defined. The feature is used for the management of mutex and condition variables in shared memory. If the feature switch is set in the define of the condition variables in shared memory, then the same feature switch must be set in the define of the mutex associated with the condition variables.

If the supplied extended pthread_cond_t object is not in shared memory, pthread_cond_init() will treat the object as a non-shared object, since it is not accessible to any other process.

If the _OPEN_SYS_MUTEX_EXT feature switch is set, a shared condition variable is tied to the specified mutex for the life of the condition variable and mutex the very first time a pthread_cond_wait() or pthread_cond_timedwait() is issued. No other mutex can be associated with the specified condition variable or vise versa until the condition variable or mutex is destroyed.

It is recommended that you define and initialize pthread_cond_t objects in the same compile unit. If you pass a pthread_cond_t object around to be initialized, make sure the initialization code has been compiled with the same _OPEN_SYS_MUTEX_EXT feature setting as the code that defines the object.

The following sequence may cause storage overlay with unpredictable results:

1. Declare or define a pthread_cond_t object (in shared storage) without #define of the _OPEN_SYS_MUTEX_EXT feature. The created pthread_cond_t object is standard size (i.e. small) without the _OPEN_SYS_MUTEX_EXT feature defined.
2. Pass the pthread_cond_t object to another code unit, which was compiled with the _OPEN_SYS_MUTEX_EXT feature defined, to be initialized as a shared object. The pthread_cond_t initialization generally involves the following steps:
   a. pthread_condattr_init()
   b. pthread_condattr_setpshared(). This step sets the attribute of the pthread_cond_t as PTHREAD_PROCESS_SHARED and designates the object to be of extended size.
   c. pthread_cond_init(). This step initializes the passed-in (small) pthread_cond_t object as if it is an extended object, causing storage overlay.

Example

CELEBP19

```
/* CELEBP19 */
#define _OPEN_THREADS
#include <pthread.h>
#include <stdio.h>

main() {
  pthread_cond_t cond;

  if (pthread_cond_init(&cond, NULL) != 0) {
    perror("pthread_cond_init() error");
    exit(1);
  }

  if (pthread_cond_destroy(&cond) != 0) {
    perror("pthread_cond_destroy() error");
    exit(2);
  }
}
```

Related information

- “pthread.h” on page 58
- “pthread_condattr_init() — Initialize a condition attribute object” on page 1256
- “pthread_cond_broadcast() — Broadcast a condition” on page 1240
- “pthread_cond_signal() — Signal a condition”
- “pthread_cond_timedwait() — Wait on a condition variable” on page 1246
- “pthread_cond_wait() — Wait on a condition variable” on page 1249

pthread_cond_signal() — Signal a condition

Standards

Standards / Extensions	C or C++	Dependencies
POSIX.4a Single UNIX Specification, Version 3	both	POSIX(ON)

Format

```
#define _OPEN_THREADS
#include <pthread.h>

int pthread_cond_signal(pthread_cond_t *cond);
```

SUSV3:

```
#define _UNIX03_THREADS
#include <pthread.h>

int pthread_cond_signal(pthread_cond_t *cond);
```

General description

Unblock at least one thread that is blocked on the specified condition variable, *cond*. If more than one thread is blocked, the order in which the threads are unblocked is unspecified.

pthread_cond_signal() will have no effect if there are no threads currently blocked on *cond*.

Returned value

If successful, pthread_cond_signal() returns 0.

If unsuccessful, pthread_cond_signal() returns -1 and sets errno to one of the following values:

Error Code
Description

EINVAL
The value specified by *cond* does not refer to an initialized condition variable.

Special behavior for Single UNIX Specification, Version 3: If unsuccessful, pthread_cond_signal() returns an error number to indicate the error.

Example

CELEBP20

```
/* CELEBP20 */
#define _OPEN_THREADS
#include <pthread.h>
#include <stdio.h>

main() {
  pthread_cond_t cond;

  if (pthread_cond_init(&cond, NULL) != 0) {
    perror("pthread_cond_init() error");
    exit(1);
  }

  if (pthread_cond_signal(&cond) != 0) {
    perror("pthread_cond_broadcast() error");
    exit(2);
  }

  if (pthread_cond_destroy(&cond) != 0) {
    perror("pthread_cond_destroy() error");
    exit(3);
  }
}
```

Related information

- "pthread.h" on page 58
- "pthread_cond_broadcast() — Broadcast a condition" on page 1240
- "pthread_cond_init() — Initialize a condition variable" on page 1243
- "pthread_cond_timedwait() — Wait on a condition variable"
- "pthread_cond_wait() — Wait on a condition variable" on page 1249

pthread_cond_timedwait() — Wait on a condition variable

Standards

Standards / Extensions	C or C++	Dependencies
POSIX.4a Single UNIX Specification, Version 3	both	POSIX(ON)

Format

```
#define _OPEN_THREADS
#include <pthread.h>

int pthread_cond_timedwait(pthread_cond_t *cond, pthread_mutex_t *mutex,
                           const struct timespec *abstime);
```

SUSV3:

```
#define _UNIX03_THREADS
#include <pthread.h>
int pthread_cond_timedwait(pthread_cond_t * __restrict__cond,
                           pthread_mutex_t * __retrict__mutex,
                           const struct timespec * __restrict__abstime);
```

General description

Allows a thread to wait on a condition variable until satisfied or until a specified time occurs. pthread_cond_timedwait() is the same as pthread_cond_wait() except it returns an error if the absolute time, specified by *abstime*, satisfies one of these conditions:
- Passes before *cond* is signaled or broadcasted
- Has already been passed at the time of the call

When such timeouts occur, pthread_cond_timedwait() reacquires the mutex, referenced by *mutex* (created by pthread_mutex_init()).

The two elements within the struct *timespec* are defined as follows:

tv_sec The time to wait for the condition signal. It is expressed in seconds from midnight, January 1, 1970 UTC. The value specified must be greater than or equal to current calendar time expressed in seconds since midnight, January 1, 1970 UTC and less than 2,147,483,648 seconds.

tv_nsec
The time in nanoseconds to be added to tv_sec to determine when to stop waiting. The value specified must be greater than or equal to zero (0) and less than 1,000,000,000 (1,000 million).

Returned value

If successful, pthread_cond_timedwait() returns 0.

If unsuccessful, pthread_cond_timedwait() returns -1 and sets errno to one of the following values:

Error Code
Description

EAGAIN
For a private condition variable, the time specified by *abstime* has passed.

EINVAL
Can be one of the following error conditions:
- The value specified by *cond* is not valid.
- The value specified by *mutex* is not valid.
- The value specified by *abstime* (`tv_sec`) is not valid.
- The value specified by *abstime* (`tv_nsec`) is not valid.
- Different mutexes were specified for concurrent operations on the same condition variable.

ETIMEDOUT

For a shared condition variable, the time specified by *abstime* has passed.

Note: In SUSV3, pthread_cond_timedwait() also returns ETIMEDOUT for a private condition variable, when the time specified by *abstime* has passed.

EPERM

The mutex was not owned by the current thread at the time of the call.

Special behavior for Single UNIX Specification, Version 3: If unsuccessful, pthread_cond_timedwait() returns an error number to indicate the error.

Usage notes

If the condition variable is shared (PTHREAD_PROCESS_SHARED), the mutex must also be shared, with the _OPEN_SYS_MUTEX_EXT feature defined when the mutex was created and initialized.

If the condition variable is private (PTHREAD_PROCESS_PRIVATE), the mutex must also be private.

If the condition variable is shared, all calls to pthread_cond_wait() or pthread_cont_timedwait() for a given condition variable must use the same mutex for the life of the process, or until both the condition variable and mutex are destroyed (using pthread_cond-destroy() and pthread_mutex_destroy()).

Example

CELEBP21

```
/* CELEBP21 */
#define _OPEN_THREADS
#include <pthread.h>
#include <stdio.h>
#include <time.h>
#include <errno.h>

main() {
  pthread_cond_t cond;
  pthread_mutex_t mutex;
  time_t T;
  struct timespec t;

  if (pthread_mutex_init(&mutex, NULL) != 0) {
    perror("pthread_mutex_init() error");
    exit(1);
  }

  if (pthread_cond_init(&cond, NULL) != 0) {
    perror("pthread_cond_init() error");
    exit(2);
  }

  if (pthread_mutex_lock(&mutex) != 0) {
    perror("pthread_mutex_lock() error");
    exit(3);
  }

  time(&T);
  t.tv_sec = T + 2;
  printf("starting timedwait at %s", ctime(&T));
  if (pthread_cond_timedwait(&cond, &mutex, &t) != 0)
```

```
    if (errno == EAGAIN)
      puts("wait timed out");
    else {
      perror("pthread_cond_timedwait() error");
      exit(4);
    }

  time(&T);
  printf("timedwait over at %s", ctime(&T));
}
```

Output:

```
starting timedwait at Fri Jun 16 10:44:00 2006
wait timed out
timedwait over at Fri Jun 16 10:44:02 2006
```

Related information

- "pthread.h" on page 58
- "pthread_cond_broadcast() — Broadcast a condition" on page 1240
- "pthread_cond_signal() — Signal a condition" on page 1245
- "pthread_cond_wait() — Wait on a condition variable"
- "pthread_mutex_init() — Initialize a mutex object" on page 1286

pthread_cond_wait() — Wait on a condition variable

Standards

Standards / Extensions	C or C++	Dependencies
POSIX.4a Single UNIX Specification, Version 3	both	POSIX(ON)

Format

```
#define _OPEN_THREADS
#include <pthread.h>

int pthread_cond_wait(pthread_cond_t *cond, pthread_mutex_t *mutex);
```

SUSV3:

```
#define _UNIX03_THREADS
#include <pthread.h>
int pthread_cond_wait(pthread_cond_t * __restrict__cond,
                      pthread_mutex_t * __restrict__mutex);
```

General description

Blocks on a condition variable. It must be called with *mutex* locked by the calling thread, or undefined behavior will result. A mutex is locked using pthread_mutex_lock().

cond is a condition variable that is shared by threads. To change it, a thread must hold the *mutex* associated with the condition variable. The pthread_cond_wait() function releases this *mutex* before suspending the thread and obtains it again before returning.

The pthread_cond_wait() function waits until a pthread_cond_broadcast() or a pthread_cond_signal() is received. For more information on these functions, refer to "pthread_cond_broadcast() — Broadcast a condition" on page 1240 and to “pthread_cond_signal() — Signal a condition” on page 1245.

Returned value

If successful, pthread_cond_wait() returns 0.

If unsuccessful, pthread_cond_wait() returns -1 and sets errno to one of the following values:

Error Code
Description

EINVAL
Different mutexes were specified for concurrent operations on the same condition variable.

EPERM
The mutex was not owned by the current thread at the time of the call.

Special behavior for Single UNIX Specification, Version 3: If unsuccessful, pthread_cond_wait() returns an error number to indicate the error.

Usage notes

If the condition variable is shared (PTHREAD_PROCESS_SHARED), the mutex must also be shared, with the _OPEN_SYS_MUTEX_EXT feature defined when the mutex was created and initialized.

If the condition variable is private (PTHREAD_PROCESS_PRIVATE), the mutex must also be private.

If the condition variable is shared, all calls to pthread_cond_wait() or pthread_cont_timedwait() for a given condition variable must use the same mutex for the life of the process, or until both the condition variable and mutex are destroyed (using pthread_cond-destroy() and pthread_mutex_destroy()).

Example

CELEBP22

```
/* CELEBP22 */
#define _OPEN_THREADS
#include <pthread.h>
#include <stdio.h>
#include <time.h>
#include <unistd.h>

pthread_cond_t cond;
pthread_mutex_t mutex;

int footprint = 0;

void *thread(void *arg) {
  time_t T;

  if (pthread_mutex_lock(&mutex) != 0) {
    perror("pthread_mutex_lock() error");
    exit(6);
  }
  time(&T);
  printf("starting wait at %s", ctime(&T));
  footprint++;
  if (pthread_cond_wait(&cond, &mutex) != 0) {
    perror("pthread_cond_timedwait() error");
    exit(7);
  }
  time(&T);
  printf("wait over at %s", ctime(&T));
}

main() {
  pthread_t thid;
  time_t T;
  struct timespec t;

  if (pthread_mutex_init(&mutex, NULL) != 0) {
    perror("pthread_mutex_init() error");
    exit(1);
  }

  if (pthread_cond_init(&cond, NULL) != 0) {
    perror("pthread_cond_init() error");
    exit(2);
  }

  if (pthread_create(&thid, NULL, thread, NULL) != 0) {
    perror("pthread_create() error");
    exit(3);
  }

  while (footprint == 0)
    sleep(1);

  puts("IPT is about ready to release the thread");
  sleep(2);

  if (pthread_cond_signal(&cond) != 0) {
    perror("pthread_cond_signal() error");
    exit(4);
  }

  if (pthread_join(thid, NULL) != 0) {
    perror("pthread_join() error");
    exit(5);
  }
}
```

Output:

```
starting wait at Fri Jun 16 10:54:06 2006
IPT is about ready to release the thread
wait over at Fri Jun 16 10:54:09 2006
```

Related information

- “pthread.h” on page 58
- “pthread_cond_broadcast() — Broadcast a condition” on page 1240
- “pthread_cond_signal() — Signal a condition” on page 1245
- “pthread_cond_timedwait() — Wait on a condition variable” on page 1246

pthread_condattr_destroy() — Destroy condition variable attribute object

Standards

Standards / Extensions	C or C++	Dependencies
POSIX.4a Single UNIX Specification, Version 3	both	POSIX(ON)

Format

```
#define _OPEN_THREADS
#include <pthread.h>

int pthread_condattr_destroy(pthread_condattr_t *attr);
```

SUSV3:

```
#define _UNIX03_THREADS
#include <pthread.h>

int pthread_condattr_destroy(pthread_condattr_t *attr);
```

General description

Destroys a condition attribute object. Condition-variable attribute objects are similar to mutex attribute objects because you can use them to manage the characteristics of condition variables in your application. They define the set of values to be used for the condition variable during its creation.

pthread_condattr_init() is used to define a condition variable attribute object. pthread_condattr_destroy() is used to remove the definition of the condition variable attribute object. These functions are provided for portability purposes.

You can define a condition variable without using these functions by supplying a NULL parameter during the pthread_cond_init() call. For more details, refer to "pthread_cond_init() — Initialize a condition variable" on page 1243.

Returned value

If successful, pthread_condattr_destroy() returns 0.

If unsuccessful, pthread_condattr_destroy() returns -1 and sets errno to one of the following values:

Error Code
  Description

EINVAL
  The value specified by *attr* is not valid.

Special behavior for Single UNIX Specification, Version 3: If unsuccessful, pthread_condattr_destroy() returns an error number to indicate the error.

Example

CELEBP23

```
/* CELEBP23 */
#define _OPEN_THREADS
#include <pthread.h>
#include <stdio.h>

main() {
  pthread_condattr_t cond;

  if (pthread_condattr_init(&cond) != 0) {
    perror("pthread_condattr_init() error");
    exit(1);
  }

  if (pthread_condattr_destroy(&cond) != 0) {
    perror("pthread_condattr_destroy() error");
    exit(2);
  }
}
```

Related information

- "pthread.h" on page 58
- "pthread_condattr_init() — Initialize a condition attribute object" on page 1256
- "pthread_cond_init() — Initialize a condition variable" on page 1243
- "pthread_mutex_init() — Initialize a mutex object" on page 1286

pthread_condattr_getkind_np() — Get kind attribute from a condition variable attribute object

Standards

Standards / Extensions	C or C++	Dependencies
z/OS UNIX	both	POSIX(ON)

Format

```
#define _OPEN_THREADS
#define _OPEN_SYS
#include <pthread.h>

int pthread_condattr_getkind_np(pthread_condattr_t *attr, int *kind);
```

General description

Gets the attribute *kind* for the condition variable attribute object *attr*. Condition variable attribute objects are similar to mutex attribute objects. You can use them to manage the characteristics of condition variables in your application. They define the set of values for the condition variable during its creation.

The valid values for the attribute *kind* are:

__COND_DEFAULT
No defined attributes.

__COND_NODEBUG
State changes to this condition variable will *not* be reported to the debug interface, even though it is present.

Returned value

If successful, pthread_condattr_getkind_np() returns 0.

If unsuccessful, pthread_condattr_getkind_np() returns -1 and sets errno to one of the following values:

Error Code
Description

EINVAL
The value specified for *attr* is not valid.

Example

CELEBP24

```
/* CELEBP24 */
#pragma runopts(TEST(ALL))

#ifndef _OPEN_THREADS
#define _OPEN_THREADS
#define _OPEN_SYS          /* Needed to identify __COND_NODEBUG and
                              __COND_DEFAULT */
#endif

#include <stdio.h>
#include <pthread.h>

pthread_condattr_t attr;

int kind;

main() {
    if (pthread_condattr_init(&attr) == -1) {
       perror("pthread_condattr_init()");
       exit(1);
    }

    if (pthread_condattr_setkind_np(&attr, __COND_NODEBUG) == -1) {
       perror("pthread_condattr_setkind_np()");
       exit(1);
    }

    if (pthread_condattr_getkind_np(&attr, &kind) == -1) {
    exit(1);
    }

    switch(kind) {

      case __COND_DEFAULT:
        printf("\ncondition variable will have no defined attributes");
        break;

      case __COND_NODEBUG:
        printf("\ncondition variable will have nodebug attribute");
        break;

      default:
        printf("\nattribute kind value returned by \
pthread_condattr_getkind_no() unrecognized");

    }

   exit(0);
}
```

Related information

- “pthread.h” on page 58
- “pthread_condattr_init() — Initialize a condition attribute object” on page 1256
- “pthread_condattr_setkind_np() — Set kind attribute from a condition variable attribute object” on page 1258

pthread_condattr_getpshared() — Get the process-shared condition variable attribute

Standards

Standards / Extensions	C or C++	Dependencies
Language Environment Extension Single UNIX Specification, Version 3	both	z/OS V1R2

Format

```
#define _OPEN_THREADS
#define _OPEN_SYS_MUTEX_EXT
#include <pthread.h>

int  pthread_condattr_getpshared(const pthread_condattr_t *attr,
                                 int *pshared);
```

SUSV3:

```
#define _UNIX03_THREADS
#include <pthread.h>
int pthread_condattr_getpshared(const pthread_condattr_t * __restrict__attr,
                                int * __restrict__pshared);
```

General description

Gets the attribute *pshared* for the condition variable attribute object *attr*. By using *attr*, you can determine its process-shared value for a condition variable.

Valid values for the attribute *pshared* are:

Value Description

PTHREAD_PROCESS_SHARED
Permits a condition variable to be operated upon by any thread that has access to the memory where the condition variable is allocated; even if the condition variable is allocated in memory that is shared by multiple processes.

PTHREAD_PROCESS_PRIVATE
A condition variable can only be operated upon by threads created within the same process as the thread that initialized the condition variable. If threads of differing processes attempt to operate on such a condition variable, only the process to initialize the condition variable will succeed. When a new process is created by the parent process it will receive a different copy of the private condition variable which can only be used to serialize between threads in the child process.

Note: This is the default value of *pshared*

Returned value

If successful, 0 is returned. If unsuccessful, -1 is returned and the errno value is set. The following is the value of errno:

Value Description

EINVAL
The value specified for *attr* is not valid.

Special behavior for Single UNIX Specification, Version 3: If unsuccessful, pthread_condattr_getpshared() returns an error number to indicate the error.

Related information

- "pthread.h" on page 58
- "pthread_condattr_setpshared() — Set the process-shared condition variable attribute" on page 1259
- "pthread_mutexattr_getpshared() — Get the process-shared mutex attribute" on page 1298
- "pthread_mutexattr_setpshared() — Set the process-shared mutex attribute" on page 1305
- "pthread_rwlockattr_getpshared() — Get the processed-shared read or write lock attribute" on page 1320
- "pthread_rwlockattr_setpshared() — Set the process-shared read or write lock attribute" on page 1322

pthread_condattr_init() — Initialize a condition attribute object

Standards

Standards / Extensions	C or C++	Dependencies
POSIX.4a Single UNIX Specification, Version 3	both	POSIX(ON)

Format

```
#define _OPEN_THREADS
#include <pthread.h>

int pthread_condattr_init(pthread_condattr_t *attr);
```

SUSV3:

```
#define _UNIX03_THREADS
#include <pthread.h>

int pthread_condattr_init(pthread_condattr_t *attr);
```

General description

Establishes the default values for the condition variables that will be created. A condition attribute (*condattr*) object contains various condition variable characteristics. You can set up a template of these characteristics and then create a set of condition variables with similar characteristics.

Condition variable attribute objects are similar to mutex attribute objects. You can use them to manage the characteristics of condition variables in your application.

They define the set of values to be used for the condition variable during its creation. For a valid condition variable attribute, refer to "pthread_condattr_setkind_np() -- Set Kind Attribute from a Condition Variable Attribute Object and pthread_condattr_setpshared() --Set the Process-Shared Condition Variable Attribute

pthread_condattr_init() is used to define a condition variable attribute object. pthread_condattr_destroy() is used to remove the definition of the condition variable attribute object. These functions are provided for portability purposes.

You can define a condition variable without using these functions by supplying a NULL parameter during the pthread_cond_init() call. For more details, refer to "pthread_cond_init() — Initialize a condition variable" on page 1243.

Returned value

If successful, pthread_condattr_init() returns 0.

If unsuccessful, pthread_condattr_init() returns -1 and sets errno to one of the following values:

Error Code
    Description

ENOMEM
    There is not enough memory to initialize the condition variable attributes object.

Special behavior for Single UNIX Specification, Version 3: If unsuccessful, pthread_condattr_init() returns an error number to indicate the error.

Example

CELEBP25

```
/* CELEBP25 */
#define _OPEN_THREADS
#include <pthread.h>
#include <stdio.h>

main() {
  pthread_condattr_t cond;

  if (pthread_condattr_init(&cond) != 0) {
    perror("pthread_condattr_init() error");
    exit(1);
  }

  if (pthread_condattr_destroy(&cond) != 0) {
    perror("pthread_condattr_destroy() error");
    exit(2);
  }
}
```

Related information

- "pthread.h" on page 58
- "pthread_cond_init() — Initialize a condition variable" on page 1243
- "pthread_condattr_getpshared() — Get the process-shared condition variable attribute" on page 1255
-

- "pthread_condattr_setpshared() — Set the process-shared condition variable attribute" on page 1259
- "pthread_mutex_init() — Initialize a mutex object" on page 1286

pthread_condattr_setkind_np() — Set kind attribute from a condition variable attribute object

Standards

Standards / Extensions	C or C++	Dependencies
z/OS UNIX	both	POSIX(ON)

Format

```
#define _OPEN_THREADS
#define _OPEN_SYS
#include <pthread.h>

int pthread_condattr_setkind_np(pthread_condattr_t *attr, int kind);
```

General description

Sets the attribute *kind* for the condition variable attribute object *attr*. Condition variable attribute objects are similar to mutex attribute objects. You can use them to manage the characteristics of condition variables in your application. They define the set of values to be used for the condition variable during its creation.

The valid values for the attribute *kind* are:

__COND_DEFAULT
No defined attributes.

__COND_NODEBUG
State changes to this condition variable will *not* be reported to the debug interface, even though it is present.

Returned value

If successful, pthread_condattr_setkind_np() returns 0.

If unsuccessful, pthread_condattr_setkind_np() returns -1 and sets errno to one of the following values:

Error Code
Description

EINVAL
The value specified for *attr* or *kind* is not valid.

Example

CELEBP26

```
/* CELEBP26 */
#pragma runopts(TEST(ALL))

#ifndef _OPEN_THREADS
#define _OPEN_THREADS
#define _OPEN_SYS     /*  Needed to identify __COND_NODEBUG and
                          __COND_DEFAULT  */
```

```
#endif

#include <stdio.h>
#include <pthread.h>

pthread_condattr_t attr;

int kind;

main() {
    if (pthread_condattr_init(&attr) == -1) {
       perror("pthread_condattr_init()");
       exit(1);
    }

    if (pthread_condattr_setkind_np(&attr, __COND_NODEBUG) == -1) {
       perror("pthread_condattr_setkind_np()");
       exit(1);
    }

    if (pthread_condattr_getkind_np(&attr, &kind) == -1) {
    exit(1);
    }

    switch(kind) {

      case __COND_DEFAULT:
        printf("\ncondition variable will have no defined attributes");
        break;

      case __COND_NODEBUG:
        printf("\ncondition variable will have nodebug attribute");
        break;

      default:
        printf("\nattribute kind value returned by \
pthread_condattr_getkind_no() unrecognized");
    }

   exit(0);
}
```

Related information

- "pthread.h" on page 58
- "pthread_condattr_init() — Initialize a condition attribute object" on page 1256
- "pthread_condattr_getkind_np() — Get kind attribute from a condition variable attribute object" on page 1253

pthread_condattr_setpshared() — Set the process-shared condition variable attribute

Standards

Standards / Extensions	C or C++	Dependencies
Language Environment Extension Single UNIX Specification, Version 3	both	z/OS V1R2

Format

```
#define _OPEN_THREADS
#define _OPEN_SYS_MUTEX_EXT
#include <pthread.h>
```

```
int  pthread_condattr_setpshared(pthread_condattr_t *attr,
                                 int pshared);
```

SUSV3:

```
#define _UNIX03_THREADS
#define _OPEN_SYS_MUTEX_EXT
#include <pthread.h>

int  pthread_condattr_setpshared(pthread_condattr_t *attr,
                                 int pshared);
```

General description

Sets the attribute *pshared* for the condition variable attribute object *attr*.

A condition variable attribute object (*attr*) allows you to manage the characteristics of condition variables in your application by defining a set of values to be used for a condition variable during its creation. By establishing a condition variable attribute object, you can create many condition variables with the same set of characteristics, without needing to define the characteristics for each and every condition variable. By using *attr*, you can define its process-shared value for a condition variable.

Valid values for the attribute *pshared* are:

Value Description

PTHREAD_PROCESS_SHARED
Permits a condition variable to be operated upon by any thread that has access to the memory where the condition variable is allocated; even if the condition variable is allocated in memory that is shared by multiple processes.

PTHREAD_PROCESS_PRIVATE
A condition variable can only be operated upon by threads created within the same process as the thread that initialized the condition variable. If threads of differing processes attempt to operate on such a condition variable, only the process to initialize the condition variable will succeed. When a new process is created by the parent process it will receive a different copy of the private condition variable which can only be used to serialize between threads in the child process.

Note: This is the default value of *pshared*.

Returned value

If successful, 0 is returned. If unsuccessful, -1 is returned and the errno value is set. The following is the value of errno:

Value Description

EINVAL
The value specified for *attr* is not valid.

Special behavior for Single UNIX Specification, Version 3: If unsuccessful, pthread_condattr_setpshared() returns an error number to indicate the error.

Usage notes

It is recommended that you define and initialize pthread_cond_t objects in the same compile unit. If you pass a pthread_cond_t object around to be initialized, make sure the initialization code has been compiled with the same _OPEN_SYS_MUTEX_EXT feature setting as the code that defines the object.

The following sequence may cause storage overlay with unpredictable results:

1. Declare or define a pthread_cond_t object (in shared storage) without #define of the _OPEN_SYS_MUTEX_EXT feature. The created pthread_cond_t object is standard size (i.e. small) without the _OPEN_SYS_MUTEX_EXT feature defined.
2. Pass the pthread_cond_t object to another code unit, which was compiled with the _OPEN_SYS_MUTEX_EXT feature defined, to be initialized as a shared object. The pthread_cond_t initialization generally involves the following steps:
   a. pthread_condattr_init()
   b. pthread_condattr_setpshared(). This step sets the attribute of the pthread_cond_t as PTHREAD_PROCESS_SHARED and designates the object to be of extended size.
   c. pthread_cond_init(). This step initializes the passed-in (small) pthread_cond_t object as if it is an extended object, causing storage overlay.

Related information

- "pthread.h" on page 58
- "pthread_condattr_getpshared() — Get the process-shared condition variable attribute" on page 1255
- "pthread_mutexattr_getpshared() — Get the process-shared mutex attribute" on page 1298
- "pthread_mutexattr_setpshared() — Set the process-shared mutex attribute" on page 1305
- "pthread_rwlockattr_getpshared() — Get the processed-shared read or write lock attribute" on page 1320
- "pthread_rwlockattr_setpshared() — Set the process-shared read or write lock attribute" on page 1322

pthread_create() — Create a thread

Standards

Standards / Extensions	C or C++	Dependencies
POSIX.4a Single UNIX Specification, Version 3	both	POSIX(ON)

Format

```
#define _OPEN_THREADS
#include <pthread.h>

int pthread_create(pthread_t *thread, pthread_attr_t *attr,
                   void *(*start_routine) (void *arg), void *arg);
```

SUSV3

```
#define _UNIX03_THREADS
#include <pthread.h>
int pthread_create(pthread_t * __restrict__thread,
                   const pthread_attr_t *attr,
                   void *(*start_routine) (void *arg),
                   void * __restrict__arg);
```

General description

Creates a new thread within a process, with attributes defined by the thread attribute object, *attr*, that is created by pthread_attr_init().

If *attr* is NULL, the default attributes are used. See "pthread_attr_init() — Initialize a thread attribute object" on page 1218 for a description of the thread attributes and their defaults. If the attributes specified by *attr* are changed later, the thread's attributes are not affected.

pthread_t is the data type used to uniquely identify a thread. It is returned by pthread_create() and used by the application in function calls that require a thread identifier.

The thread is created running *start_routine*, with *arg* as the only argument. If pthread_create() completes successfully, *thread* will contain the ID of the created thread. If it fails, no new thread is created, and the contents of the location referenced by *thread* are undefined.

System default for the thread limit in a process is set by MAXTHREADS in the BPXPRMxx parmlib member.

The maximum number of threads is dependent upon the size of the private area below 16M. pthread_create() inspects this address space before creating a new thread. A realistic limit is 200 to 400 threads.

Special behavior for C++: Because C and C++ linkage conventions are incompatible, pthread_create() cannot receive a C++ function pointer as the start routine function pointer If you attempt to pass a C++ function pointer to pthread_create(), the compiler will flag it as an error. You can pass a C or C++ function to pthread_create() by declaring it as extern "C".

The started thread provides a boundary with respect to the scope of try-throw-catch processing. A throw done in the start routine or a function called by the start routine causes stack unwinding up to and including the start routine (or until caught). The stack unwinding will not go beyond the start routine back into the thread creator. If the exception is not caught, terminate() is called.

The exception stack (for try-throw-catch) are thread-based. The throw of a condition, or re-throw of a condition by a thread does not affect exception processing on another thread, unless the condition is not caught.

Returned value

If successful, pthread_create() returns 0.

If unsuccessful, pthread_create() returns -1 and sets errno to one of the following values:

Error Code
Description

EAGAIN
The system lacks the necessary resources to create another thread.

EINVAL
The value specified by *thread* is null.

ELEMULTITHREADFORK
pthread_create() was invoked from a child process created by calling fork() from a multi-threaded process. This child process is restricted from becoming multi-threaded.

ENOMEM
There is not enough memory to create the thread.

Special behavior for Single UNIX Specification, Version 3: If unsuccessful, pthread_create() returns an error number to indicate the error.

Example

CELEBP27

```
/* CELEBP27 */
#define _OPEN_THREADS
#include <pthread.h>
#include <stdlib.h>
#include <stdio.h>
void *thread(void *arg) {
  char *ret;
  printf("thread() entered with argument '%s'\n", arg);
  if ((ret = (char*) malloc(20)) == NULL) {
    perror("malloc() error");
    exit(2);
  }
  strcpy(ret, "This is a test");
  pthread_exit(ret);
}

main() {
  pthread_t thid;
  void *ret;

  if (pthread_create(&thid, NULL, thread, "thread 1") != 0) {
    perror("pthread_create() error");
    exit(1);
  }

  if (pthread_join(thid, &ret) != 0) {
    perror("pthread_create() error");
    exit(3);
  }

  printf("thread exited with '%s'\n", ret);
}
```

Output:

```
thread() entered with argument 'thread 1'
thread exited with 'This is a test'
```

Related information

- “pthread.h” on page 58
- “pthread_exit() — Exit a thread” on page 1267

- "pthread_join() — Wait for a thread to end" on page 1275

pthread_detach() — Detach a thread

Standards

Standards / Extensions	C or C++	Dependencies
POSIX.4a Single UNIX Specification, Version 3	both	POSIX(ON)

Format

```
#define _OPEN_THREADS
#include <pthread.h>

int pthread_detach(pthread_t *thread);
```

SUSV3:

```
#define _UNIX03_THREADS
#include <pthread.h>
int pthread_detach(pthread_t thread);
```

General description

Allows storage for the thread whose thread ID is in the location *thread* to be reclaimed when that thread ends. This storage is reclaimed on process exit, regardless of whether the thread was detached, and may include storage for *thread*'s return value. If *thread* has not ended, pthread_detach() will not cause it to end.

pthread_t is the data type used to uniquely identify a thread. It is returned by pthread_create() and used by the application in function calls that require a thread identifier.

Returned value

If successful, pthread_detach() returns 0.

If unsuccessful, pthread_detach() returns -1 and sets errno to one of the following values:

Error Code
Description

EINVAL
The value specified by *thread* is not valid.

ESRCH
A value specified by *thread* refers to a thread that is already detached.

Special behavior for Single UNIX Specification, Version 3: If unsuccessful, pthread_detach() returns an error number to indicate the error.

Example

CELEBP28

```
/* CELEBP28 */
#define _OPEN_SYS
#define _OPEN_THREADS
#include <pthread.h>
#include <stdlib.h>
#include <stdio.h>

void *thread(void *arg) {
  char *ret;
  printf("thread() entered with argument '%s'\n", arg);
  if ((ret = (char*) malloc(20)) == NULL) {
    perror("malloc() error");
    exit(2);
  }
  strcpy(ret, "This is a test");
  pthread_exit(ret);
}

main() {
  pthread_t thid;
  void *ret;

  if (pthread_create(&thid, NULL, thread, "thread 1") != 0) {
    perror("pthread_create() error");
    exit(1);
  }

  if (pthread_join_d4_np(thid, &ret) != 0) {
    perror("pthread_create() error");
    exit(3);
  }

  printf("thread exited with '%s'\n", ret);

  if (pthread_detach(&thid) != 0) {
    perror("pthread_detach() error");
    exit(4);
  }

}
```

Output:

```
thread() entered with argument 'thread 1'
thread exited with 'This is a test'
```

Related information

- "pthread.h" on page 58
- "pthread_join() — Wait for a thread to end" on page 1275

pthread_equal() — Compare thread IDs

Standards

Standards / Extensions	C or C++	Dependencies
POSIX.4a Single UNIX Specification, Version 3	both	POSIX(ON)

Format

```
#define _OPEN_THREADS
#include <pthread.h>

int pthread_equal(pthread_t t1, pthread_t t2);
```

General description

Compares the thread IDs of *t1* and *t2*.

pthread_t is the data type used to uniquely identify a thread. It is returned by pthread_create() and used by the application in function calls that require a thread identifier.

Returned value

If *t1* and *t2* are equal, pthread_equal() returns a positive value. Otherwise, the value 0 is returned. If *t1* or *t2* are not valid thread IDs, the behavior is undefined.

If unsuccessful, pthread_equal() returns -1.

There are no documented errno values. Use perror() or strerror() to determine the cause of the error.

Example

CELEBP29

```
/* CELEBP29 */
#define _OPEN_THREADS
#include <pthread.h>
#include <stdio.h>

pthread_t thid, IPT;

void *thread(void *arg) {
  if (pthread_equal(IPT, thid))
    puts("the thread is the IPT...?");
  else
    puts("the thread is not the IPT");
}

main() {

  IPT = pthread_self();

  if (pthread_create(&thid, NULL, thread, NULL) != 0) {
    perror("pthread_create() error");
    exit(1);
  }

  if (pthread_join(thid, NULL) != 0) {
    perror("pthread_create() error");
    exit(3);
  }
}
```

Output:

```
the thread is not the IPT
```

Related information

- “pthread.h” on page 58
- “pthread_create() — Create a thread” on page 1261
- “pthread_self() — Get the caller” on page 1327

pthread_exit() — Exit a thread

Standards

Standards / Extensions	C or C++	Dependencies
POSIX.4a Single UNIX Specification, Version 3	both	POSIX(ON)

Format

```
#define _OPEN_THREADS
#include <pthread.h>

void pthread_exit(void *status);
```

General description

Ends the calling thread and makes *status* available to any thread that calls pthread_join() with the ending thread's thread ID.

As part of pthread_exit() processing, cleanup and destructor routines may be run:

- For details on the cleanup routines, refer to “pthread_cleanup_pop() — Remove a cleanup handler” on page 1237 and “pthread_cleanup_push() — Establish a cleanup handler” on page 1238.
- For details on the destructor routine, refer to “pthread_key_create() — Create thread-specific data key” on page 1279.

Special behavior for C++: Destructors for automatic objects on the stack will be run when a thread is canceled. The stack is unwound and the destructors are run in reverse order.

Returned value

pthread_exit() cannot return to its caller.

There are no documented errno values. Use perror() or strerror() to determine the cause of the error.

Example

CELEBP30

```
/* CELEBP30 */
#define _OPEN_THREADS
#include <pthread.h>
#include <stdlib.h>
#include <stdio.h>

void *thread(void *arg) {
  char *ret;

  if ((ret = (char*) malloc(20)) == NULL) {
    perror("malloc() error");
    exit(2);
  }
  strcpy(ret, "This is a test");
  pthread_exit(ret);
}
```

```
main() {
  pthread_t thid;
  void *ret;

  if (pthread_create(&thid, NULL, thread, NULL) != 0) {
    perror("pthread_create() error");
    exit(1);
  }

  if (pthread_join(thid, &ret) != 0) {
    perror("pthread_create() error");
    exit(3);
  }

  printf("thread exited with '%s'\n", ret);
}
```

Output:

```
thread exited with 'This is a test'
```

Related information

- "pthread.h" on page 58
- "pthread_cleanup_pop() — Remove a cleanup handler" on page 1237
- "pthread_cleanup_push() — Establish a cleanup handler" on page 1238
- "pthread_create() — Create a thread" on page 1261
- "pthread_join() — Wait for a thread to end" on page 1275
- "pthread_key_create() — Create thread-specific data key" on page 1279

pthread_getconcurrency() — Get the level of concurrency

Standards

Standards / Extensions	C or C++	Dependencies
Single UNIX Specification, Version 3	both	z/OS V1R7 POSIX(ON)

Format

```
#define _OPEN_THREADS 2
#include <pthread.h>

int pthread_getconcurrency(void);
```

General description

pthread_getconcurrency() returns the value set by a previous call to pthread_setconcurrency(), or 0 if pthread_setconcurrency() was not previously called.

Returned value

If successful, pthread_getconcurrency() returns the concurrency level set by a previous call to pthread_setconcurrency(); otherwise, 0.

Related information

- "Thread Cancellation" in the *z/OS XL C/C++ Programming Guide*
- "pthread.h" on page 58
- "pthread_setconcurrency() — Set the level of concurrency" on page 1330

pthread_getspecific() — Get the thread-specific value for a key

Standards

Standards / Extensions	C or C++	Dependencies
POSIX.4a Single UNIX Specification, Version 3	both	POSIX(ON)

Format

```
#define _OPEN_THREADS
#include <pthread.h>

int pthread_getspecific(pthread_key_t key, void **value);
```

SUSV3:

```
#define _UNIX03_THREADS
#include <pthread.h>
void *pthread_getspecific(pthread_key_t key);
```

General description

Returns the thread-specific data associated with the specified *key* for the current thread. If no thread-specific data has been set for *key*, the NULL value is returned in *value*.

Many multithreaded applications require storage shared among threads, where each thread has its own unique value. A thread-specific data key is an identifier, created by a thread, for which each thread in the process can set a unique key *value*.

pthread_key_t is a storage area where the system places the key identifier. To create a key, a thread uses pthread_key_create(). This returns the key identifier into the storage area of type *pthread_key_t*. At this point, each of the threads in the application has the use of that key, and can set its own unique value by using pthread_setspecific(). A thread can get its own unique value using pthread_getspecific().

Returned value

When unsuccessful, pthread_getspecific() sets errno to one of the following values:

Error Code
 Description

EINVAL
 The value for *key* is not valid.

Note: In SUSV3, if the key is invalid, pthread_getspecific() returns NULL but does not set or return an errno value.

Example

CELEBP31

```
/* CELEBP31 */
#ifndef _OPEN_THREADS
#define _OPEN_THREADS
#endif
```

```
#include <stdio.h>
#include <stdlib.h>
#include <errno.h>
#include <pthread.h>

#define threads 3
#define BUFFSZ  48
pthread_key_t   key;

void            *threadfunc(void *parm)
{
 int        status;
 void      *value;
 int        threadnum;
 int       *tnum;
 void      *getvalue;
 char       Buffer[BUFFSZ];

 tnum = parm;
 threadnum = *tnum;

 printf("Thread %d executing\n", threadnum);

 if (!(value = malloc(sizeof(Buffer))))
     printf("Thread %d could not allocate storage, errno = %d\n",
                                                  threadnum, errno);
 status = pthread_setspecific(key, (void *) value);
 if ( status <  0) {
    printf("pthread_setspecific failed, thread %d, errno %d",
                                                  threadnum, errno);
    pthread_exit((void *)12);
 }
 printf("Thread %d setspecific value: %d\n", threadnum, value);

 getvalue = 0;
 status = pthread_getspecific(key, &getvalue);
 if ( status <  0) {
    printf("pthread_getspecific failed, thread %d, errno %d",
                                                  threadnum, errno);
    pthread_exit((void *)13);
 }

 if (getvalue != value)
 {
   printf("getvalue not valid, getvalue=%d", (int)getvalue);
   pthread_exit((void *)68);
 }

 pthread_exit((void *)0);
}

void  destr_fn(void *parm)
{

   printf("Destructor function invoked\n");
   free(parm);
}

main() {
 int          getvalue;
 int          status;
 int          i;
 int          threadparm[threads];
 pthread_t    threadid[threads];
 int          thread_stat[threads];
 if ((status = pthread_key_create(&key, destr_fn )) < 0) {
    printf("pthread_key_create failed, errno=%d", errno);
    exit(1);
 }

 /* create 3 threads, pass each its number */
 for (i=0; i<threads; i++) {
    threadparm[i] = i+1;
    status = pthread_create( &threadid[i],
                             NULL,
                             threadfunc,
                             (void *)&threadparm[i]);
    if ( status <  0) {
       printf("pthread_create failed, errno=%d", errno);
       exit(2);
    }
  }

 for ( i=0; i<threads; i++) {
    status = pthread_join( threadid[i], (void *)&thread_stat[i]);
    if ( status <  0) {
       printf("pthread_join failed, thread %d, errno=%d\n", i+1, errno);
    }

    if (thread_stat[i] != 0)   {
        printf("bad thread status, thread %d, status=%d\n", i+1,
                                                   thread_stat[i]);
      }
  }

 exit(0);
}
```

CELEBP70

```
/* CELEBP70 */
/* Example using SUSv3 pthread_getspecific() interface */

#define _UNIX03_THREADS  1

#include <stdio.h>
#include <stdlib.h>
#include <errno.h>
#include <pthread.h>

#define threads 3
#define BUFFSZ  48
pthread_key_t   key;

void            *threadfunc(void *parm)
{
 int        status;
 void      *value;
 int        threadnum;
 int       *tnum;
 void      *getvalue;
 char       Buffer[BUFFSZ];

 tnum = parm;
 threadnum = *tnum;

 printf("Thread %d executing\n", threadnum);

 if (!(value = malloc(sizeof(Buffer))))
     printf("Thread %d could not allocate storage, errno = %d\n",
                                                  threadnum, errno);
 status = pthread_setspecific(key, (void *) value);
 if ( status <  0) {
    printf("pthread_setspecific failed, thread %d, errno %d",
                                                  threadnum, errno);
    pthread_exit((void *)12);
 }
 printf("Thread %d setspecific value: %d\n", threadnum, value);

 getvalue = 0;
 getvalue = pthread_getspecific(key);
 if ( getvalue ==  0) {
    printf("pthread_getspecific failed, thread %d", threadnum);
    printf("   rc= %d, errno %d, ejr %08x\n",(int)getvalue, errno, __errno2());
    pthread_exit((void *)13);
 } else {
   printf("Success!\n");
   printf("Returned value: %d matches set value: %d\n", getvalue, value);
 }

 if (getvalue != value)
 {
   printf("getvalue not valid, getvalue=%d", (int)getvalue);
   pthread_exit((void *)68);
 }

 pthread_exit((void *)0);
}

void  destr_fn(void *parm)
{

   printf("Destructor function invoked\n");
   free(parm);
}

int main(void)
{
  int          status;
  int          i;
  int          threadparm[threads];
  pthread_t    threadid[threads];
  int          thread_stat[threads];


  if ((status = pthread_key_create(&key, destr_fn )) < 0) {
     printf("pthread_key_create failed, errno=%d", errno);
     exit(1);
  }

  /* create 3 threads, pass each its number */
  for (i=0; i<threads; i++) {
      threadparm[i] = i+1;
     status = pthread_create( &threadid[i],
                               NULL,
                              threadfunc,
                              (void *)&threadparm[i]);
     if ( status <  0) {
        printf("pthread_create failed, errno=%d", errno);
        exit(2);
     }
  }

  for ( i=0; i<threads; i++) {
      status = pthread_join( threadid[i], (void *)&thread_stat[i]);
      if ( status <  0) {
         printf("pthread_join failed, thread %d, errno=%d\n", i+1, errno);
      }
      if (thread_stat[i] != 0)   {
         printf("bad thread status, thread %d, status=%d\n", i+1,
                                                   thread_stat[i]);
      }
   }

   exit(0);
}
```

Related information

- "pthread.h" on page 58
- "pthread_getspecific_d8_np() — Get the thread-specific value for a key"
- "pthread_key_create() — Create thread-specific data key" on page 1279
- "pthread_setspecific() — Set the thread-specific value for a key" on page 1336

pthread_getspecific_d8_np() — Get the thread-specific value for a key

Standards

Standards / Extensions	C or C++	Dependencies
POSIX.4a, draft 8	both	POSIX(ON)

Format

```
#define _OPEN_THREADS
#include <pthread.h>

void *pthread_getspecific_d8_np(pthread_key_t key);
```

General description

Returns the thread-specific data associated with the specified *key* for the current thread. If no thread-specific data has been set for *key*, the NULL value is returned.

Many multithreaded applications require storage shared among threads, where each thread has its own unique value. A thread-specific data key is an identifier, created by a thread, for which each thread in the process can set a unique key *value*.

pthread_key_t is a storage area where the system places the key identifier. To create a key, a thread uses pthread_key_create(). This returns the key identifier into the storage area of type *pthread_key_t*. At this point, each of the threads in the application has the use of that key, and can set its own unique value by using pthread_setspecific(). A thread can get its own unique value using pthread_getspecific_d8_np() or pthread_getspecific().

The only difference between pthread_getspecific_d8_np() and pthread_getspecific() is the syntax of the function.

Returned value

When successful, pthread_getspecific_d8_np() returns the thread-specific data value associated with *key*.

When unsuccessful, pthread_getspecific_d8_np() returns NULL and sets errno to one of the following values:

Error Code
Description

EINVAL
The value for *key* is not valid.

Example

```
#ifndef _OPEN_THREADS
#define _OPEN_THREADS
#endif

#include <stdio.h>
#include <stdlib.h>
#include <errno.h>
#include <pthread.h>

#define threads 3
#define BUFFSZ  48
pthread_key_t   key;

void  *threadfunc(void *);
void   destr_fn(void *);

main() {
 int          status;
 int          i;
 int          threadparm[threads];
 pthread_t    threadid[threads];
 int          thread_stat[threads];


 if ((status = pthread_key_create(&key, destr_fn )) < 0) {
    printf("pthread_key_create failed, errno=%d", errno);
    exit(1);
 }

 /* create 3 threads, pass each its number */
 for (i=0; i<threads; i++) {
    threadparm[i] = i+1;
    status = pthread_create( &threadid[i],
                             NULL,
                             threadfunc,
                             (void *)&threadparm[i]);
    if ( status <  0) {
       printf("pthread_create failed, errno=%d", errno);
       exit(2);
    }
  }

 for ( i=0; i<threads; i++) {
    status = pthread_join( threadid[i], (void *)&thread_stat[i]);
    if ( status <  0) {
       printf("pthread_join failed, thread %d, errno=%d\n", i+1, errno);
    }

    if (thread_stat[i] != 0)   {
        printf("bad thread status, thread %d, status=%d\n", i+1,
                                                   thread_stat[i]);
      }
  }

 exit(0);
}


void  *threadfunc(void *parm) {
 int       status;
 int       *void;
 int       threadnum;
 int       *tnum;
 void      *getvalue;
 char      Buffer[BUFFSZ];

 tnum = parm;
 threadnum = *tnum;

 printf("Thread %d executing\n", threadnum);

 if (!(value = malloc(sizeof(Buffer))))
     printf("Thread %d could not allocate storage, errno = %d\n",
                                              threadnum, errno);
 status = pthread_setspecific(key, (void *) value);
 if ( status <  0) {
    printf("pthread_setspecific failed, thread %d, errno %d",
                                              threadnum, errno);
    pthread_exit((void *)12);
 }
 printf("Thread %d setspecific value: %d\n", threadnum, value);

 getvalue = pthread_getspecific_d8_np(key);
 if ( getvalue == NULL) {
    printf("pthread_getspecific_d8_np failed, thread %d, errno %d",
                                              threadnum, errno);
    pthread_exit((void *)13);
 }

 pthread_exit((void *)0);
}


void  destr_fn(void *parm)
{

   printf("Destructor function invoked\n");
   free(parm)
}
```

Related information

- "pthread.h" on page 58
- "pthread_key_create() — Create thread-specific data key" on page 1279
- "pthread_getspecific() — Get the thread-specific value for a key" on page 1269
- "pthread_setspecific() — Set the thread-specific value for a key" on page 1336

pthread_join() — Wait for a thread to end

Standards

Standards / Extensions	C or C++	Dependencies
POSIX.4a Single UNIX Specification, Version 3	both	POSIX(ON)

Format

```
#define _OPEN_THREADS
#include <pthread.h>

int pthread_join(pthread_t thread, void **status);
```

SUSV3:

```
#define _UNIX03_THREADS
#include <pthread.h>

int pthread_join(pthread_t thread, void **status);
```

General description

Allows the calling thread to wait for the ending of the target *thread*.

pthread_t is the data type used to uniquely identify a thread. It is returned by pthread_create() and used by the application in function calls that require a thread identifier.

status contains a pointer to the *status* argument passed by the ending thread as part of pthread_exit(). If the ending thread terminated with a return, *status* contains a pointer to the return value. If the thread was canceled, *status* can be set to -1.

Returned value

If successful, pthread_join() returns 0.

If unsuccessful, pthread_join() returns -1 and sets errno to one of the following values:

Error Code
  Description

EDEADLK
  A deadlock has been detected. This can occur if the target is directly or indirectly joined to the current thread.

EINVAL
  The value specified by *thread* is not valid.

ESRCH
  The value specified by *thread* does not refer to an undetached thread.

Notes:

1. When pthread_join() returns successfully, the target thread has been detached.
2. Multiple threads cannot use pthread_join() to wait for the same target thread to end. If a thread issues pthread_join() for a target thread after another thread has successfully issued pthread_join() for the same target thread, the second pthread_join() will be unsuccessful.
3. If the thread calling pthread_join() is canceled, the target thread is not detached.

Special behavior for Single UNIX Specification, Version 3: If unsuccessful, pthread_join() returns an error number to indicate the error.

Example

CELEBP32

```
/* CELEBP32 */
#define _OPEN_THREADS
#include <pthread.h>
#include <stdlib.h>
#include <stdio.h>

void *thread(void *arg) {
  char *ret;
  printf("thread() entered with argument '%s'\n", arg);
  if ((ret = (char*) malloc(20)) == NULL) {
    perror("malloc() error");
    exit(2);
  }
  strcpy(ret, "This is a test");
  pthread_exit(ret);
}

main() {
  pthread_t thid;
  void *ret;

  if (pthread_create(&thid, NULL, thread, "thread 1") != 0) {
    perror("pthread_create() error");
    exit(1);
  }

  if (pthread_join(thid, &ret) != 0) {
    perror("pthread_create() error");
    exit(3);
  }

  printf("thread exited with '%s'\n", ret);
}
```

Output:

```
thread() entered with argument 'thread 1'
thread exited with 'This is a test'
```

Related information

- “pthread.h” on page 58
- “pthread_create() — Create a thread” on page 1261
- “pthread_cond_wait() — Wait on a condition variable” on page 1249
- “pthread_detach() — Detach a thread” on page 1264

pthread_join_d4_np() — Wait for a thread to end

Standards

Standards / Extensions	C or C++	Dependencies
z/OS UNIX	both	POSIX(ON)

Format

```
#define _OPEN_SYS
#define _OPEN_SYS
#include <pthread.h>

int pthread_join_d4_np(pthread_t thread, void **status);
```

General description

Allows the calling thread to wait for the ending of the target *thread*.

pthread_t is the data type used to uniquely identify a thread. It is returned by pthread_create() and used by the application in function calls that require a thread identifier.

status contains a pointer to the *status* argument passed by the ending thread as part of pthread_exit(). If the ending thread ended by a return, *status* contains a pointer to the return value. If the thread was canceled, *status* can be set to -1.

Returned value

If successful, pthread_join_d4_np() returns 0.

If unsuccessful, pthread_join_d4_np() returns -1 and sets errno to one of the following values:

Error Code
  Description

EDEADLK
  A deadlock has been detected. This can occur if the target is directly or indirectly joined to the current thread.

EINVAL
  The value specified by *thread* is not valid.

ESRCH
  The value specified by *thread* does not refer to an undetached thread.

Notes:

1. When pthread_join_d4_np() returns successfully, the target thread has not been detached.
2. Multiple threads can use pthread_join_d4_np() to wait for the same target thread to end.

Example

CELEBP33

```
/* CELEBP33 */
#define _OPEN_SYS
#define _OPEN_THREADS
#include <pthread.h>
#include <stdlib.h>
#include <stdio.h>

void *thread(void *arg) {
  char *ret;
  printf("thread() entered with argument '%s'\n", arg);
  if ((ret = (char*) malloc(20)) == NULL) {
    perror("malloc() error");
    exit(2);
  }
  strcpy(ret, "This is a test");
  pthread_exit(ret);
}

main() {
  pthread_t thid;
  void *ret;

  if (pthread_create(&thid, NULL, thread, "thread 1") != 0) {
    perror("pthread_create() error");
    exit(1);
  }

  if (pthread_join_d4_np(thid, &ret) != 0) {
    perror("pthread_create() error");
    exit(3);
  }

  printf("thread exited with '%s'\n", ret);

  if (pthread_detach(&thid) != 0) {
    perror("pthread_detach() error");
    exit(4);
  }
}
```

Output:

```
thread() entered with argument 'thread 1'
thread exited with 'This is a test'
```

Related information

- "pthread.h" on page 58
- "pthread_cond_wait() — Wait on a condition variable" on page 1249
- "pthread_create() — Create a thread" on page 1261
- "pthread_detach() — Detach a thread" on page 1264

pthread_key_create() — Create thread-specific data key

Standards

Standards / Extensions	C or C++	Dependencies
POSIX.4a Single UNIX Specification, Version 3	both	POSIX(ON)

Format

```
#define _OPEN_THREADS
#include <pthread.h>

int pthread_key_create(pthread_key_t *key, void (*destructor)(void *));
```

SUSV3:

```
#define _UNIX03_THREADS
#include <pthread.h>

int pthread_key_create(pthread_key_t *key, void (*destructor)(void *));
```

General description

Creates a key identifier, associated with *key*, and returns the key identifier into the storage area of type *pthread_key_t*. At this point, each of the threads in the application has the use of that key, and can set its own unique value by use of pthread_setspecific(). A thread can get its own unique value using pthread_getspecific().

The *destructor* routine may be called when the thread ends. It is called when a non-NULL value has been set for the key for this thread, using pthread_setspecific(), and the thread:

- Calls pthread_exit()
- Does a return from the start routine
- Is canceled because of a pthread_cancel() request.

When called, the destructor routine is passed the value bound to the key by the use of pthread_setspecific().

Special behavior for C++: Because C and C++ linkage conventions are incompatible, pthread_key_create() cannot receive a C++ function pointer as the start routine function pointer If you attempt to pass a C++ function pointer to pthread_key_create(), the compiler will flag it as an error. You can pass a C or C++ function to pthread_key_create() by declaring it as extern "C".

Returned value

If successful, pthread_key_create() returns 0 and stores the newly created key identifier in *key*.

If unsuccessful, pthread_key_create() returns -1 and sets errno to one of the following values:

Error Code
  Description

EAGAIN
  There were not enough system resources to create another thread-specific data key, or the limit is exceeded for the total number of keys per process.

ENOMEM
  There is not enough memory to create *key*.

Special behavior for Single UNIX Specification, Version 3: If unsuccessful, pthread_key_create() returns an error number to indicate the error.

Example

CELEBP34

```
/* CELEBP34 */
#ifndef _OPEN_THREADS
#define _OPEN_THREADS
#endif

#include <stdio.h>
#include <stdlib.h>
#include <errno.h>
#include <pthread.h>

#define threads 3
#define BUFFSZ  48
pthread_key_t   key;

void            *threadfunc(void *parm)
{
 int        status;
 void      *value;
 int        threadnum;
 int       *tnum;
 void      *getvalue;
 char       Buffer[BUFFSZ];

 tnum = parm;
 threadnum = *tnum;

 printf("Thread %d executing\n", threadnum);

 if (!(value = malloc(sizeof(Buffer))))
```

```
    printf("Thread %d could not allocate storage, errno = %d\n",
                                             threadnum, errno);
 status = pthread_setspecific(key, (void *) value);
 if ( status <  0) {
    printf("pthread_setspecific failed, thread %d, errno %d",
                                             threadnum, errno);
    pthread_exit((void *)12);
 }
 printf("Thread %d setspecific value: %d\n", threadnum, value);

 getvalue = 0;
 status = pthread_getspecific(key, &getvalue);
 if ( status <  0) {
    printf("pthread_getspecific failed, thread %d, errno %d",
                                             threadnum, errno);
    pthread_exit((void *)13);
 }

 if (getvalue != value) {
   printf("getvalue not valid, getvalue=%d", (int)getvalue);
   pthread_exit((void *)68);
 }

 pthread_exit((void *)0);
}

void  destr_fn(void *parm)
{

   printf("Destructor function invoked\n");
   free(parm);
}


main() {
 int          getvalue;
 int          status;
 int          i;
 int          threadparm[threads];
 pthread_t    threadid[threads];
 int          thread_stat[threads];

 if ((status = pthread_key_create(&key, destr_fn )) < 0) {
    printf("pthread_key_create failed, errno=%d", errno);
    exit(1);
 }

 /* create 3 threads, pass each its number */
 for (i=0; i<threads; i++) {
    threadparm[i] = i+1;
    status = pthread_create( &threadid[i],
                             NULL,
                             threadfunc,
                             (void *)&threadparm[i]);
    if ( status <  0) {
       printf("pthread_create failed, errno=%d", errno);
       exit(2);
    }
  }

 for ( i=0; i<threads; i++) {
    status = pthread_join( threadid[i], (void *)&thread_stat[i]);
    if ( status <  0) {
       printf("pthread_join failed, thread %d, errno=%d\n", i+1, errno);
    }

    if (thread_stat[i] != 0)   {
        printf("bad thread status, thread %d, status=%d\n", i+1,
                                                  thread_stat[i]);
      }
  }

 exit(0);
}
```

Related information

- "pthread.h" on page 58
- "pthread_getspecific() — Get the thread-specific value for a key" on page 1269
- "pthread_getspecific_d8_np() — Get the thread-specific value for a key" on page 1273
- "pthread_setspecific() — Set the thread-specific value for a key" on page 1336

pthread_key_delete() — Delete thread-specific data key

Standards

Standards / Extensions	C or C++	Dependencies
Single UNIX Specification, version 3	both	z/OS V1R7 POSIX(ON)

Format

```
#define _OPEN_THREADS 2
#include <pthread.h>

int pthread_key_delete(pthread_key_t key);
```

General description

pthread_key_delete() deletes thread-specific data keys created with pthread_key_create(). The thread-specific data values associated with *key* do not need to be NULL when the *key* is deleted. The application is responsible for freeing any storage or cleaning up data structures referring to thread-specific data associated with the deleted *key* in any thread. After *key* has been deleted, passing it to any function taking a thread-specific data key results in undefined behavior.

pthread_key_delete() can be called from destructor functions. Calling pthread_key_delete() will not cause any destructor functions to be invoked. Any destructor function associated with *key* when it was created will not be called on thread exit after *key* has been deleted.

Returned value

If successful, pthread_key_delete() returns 0. Upon failure, pthread_key_delete() returns an error number to indicate the error:

Error Code
Description

EINVAL
The key value is invalid.

Related information

- "pthread.h" on page 58
- "pthread_getspecific() — Get the thread-specific value for a key" on page 1269

- "pthread_getspecific_d8_np() — Get the thread-specific value for a key" on page 1273
- "pthread_key_create() — Create thread-specific data key" on page 1279
- "pthread_setspecific() — Set the thread-specific value for a key" on page 1336
- "unsetenv() — Delete an environment variable" on page 1965

pthread_kill() — Send a signal to a thread

Standards

Standards / Extensions	C or C++	Dependencies
POSIX.4a Single UNIX Specification, Version 3	both	POSIX(ON)

Format

```
#define _OPEN_THREADS
#include <pthread.h>
#include <signal.h>

int pthread_kill(pthread_t thread, int sig);
```

SUSV3:

```
#define _UNIX03_THREADS
#include <signal.h>

int pthread_kill(pthread_t thread, int sig);
```

General description

Directs a signal *sig* to the thread *thread*. The value of *sig* must be either 0 or one of the symbols defined in signal.h. (See Table 48 on page 1612 for a list of signals.) If *sig* is 0, pthread_kill() performs error checking but does not send a signal.

pthread_t is the data type used to uniquely identify a thread. It is returned by pthread_create() and used by the application in function calls that require a thread identifier.

Special behavior for C++: If a thread is sent a signal using pthread_kill() and that thread does not handle the signal, then destructors for local objects may not be executed.

Usage notes

1. The SIGTHSTOP and SIGTHCONT signals can be issued by this function. *pthread_kill()* is the only function that can issue SIGTHSTOP or SIGTHCONT.

Returned value

If successful, pthread_kill() returns 0.

If unsuccessful, pthread_kill() returns -1 sends no signal, and sets errno to one of the following values:

Error Code
Description

EINVAL
One of the following error conditions exists:
- The thread ID specified by *thread* is not valid.
- The value of *sig* is incorrect or is not the number of a supported signal.

ESRCH
No thread could be found corresponding to that specified by the given thread ID.

Special behavior for Single UNIX Specification, Version 3: If unsuccessful, pthread_kill() returns an error number to indicate the error.

Example

CELEBP35

```
/* CELEBP35 */
#define _OPEN_THREADS

#include <errno.h>
#include <pthread.h>
#include <signal.h>
#include <stdio.h>
#include <unistd.h>

void            *threadfunc(void *parm)
{
 int        threadnum;
 int        *tnum;
 sigset_t   set;

 tnum = parm;
 threadnum = *tnum;

 printf("Thread %d executing\n", threadnum);
 sigemptyset(&set);
 if(sigaddset(&set, SIGUSR1) == -1) {
    perror("Sigaddset error");
    pthread_exit((void *)1);
 }

 if(sigwait(&set) != SIGUSR1) {
    perror("Sigwait error");
    pthread_exit((void *)2);
 }

 pthread_exit((void *)0);
}

main() {
 int          status;
 int          threadparm = 1;
 pthread_t    threadid;
 int          thread_stat;

 status = pthread_create( &threadid,
                          NULL,
                          threadfunc,
                          (void *)&threadparm);
 if ( status <  0) {
    perror("pthread_create failed");
    exit(1);
 }

 sleep(5);
 status = pthread_kill( threadid, SIGUSR1);
 if ( status <  0)
    perror("pthread_kill failed");

 status = pthread_join( threadid, (void *)&thread_stat);
 if ( status <  0)
    perror("pthread_join failed");

 exit(0);
}
```

Related information

- "pthread.h" on page 58
- "signal.h" on page 63
- "bsd_signal() — BSD version of signal()" on page 219
- "kill() — Send a signal to a process" on page 931
- "killpg() — Send a signal to a process group" on page 934
- "pthread_self() — Get the caller" on page 1327
- "raise() — Raise signal" on page 1371
- "sigaction() — Examine or change a signal action" on page 1611
- "sighold() — Add a signal to a thread" on page 1636
- "sigignore() — Set disposition to ignore a signal" on page 1637
- "signal() — Handle interrupts" on page 1643
- "sigprocmask() — Examine or change a thread" on page 1651
- "sigrelse() — Remove a signal from a thread" on page 1656
- "sigset() — Change a signal action or a thread" on page 1656

pthread_mutex_destroy() — Delete a mutex object

Standards

Standards / Extensions	C or C++	Dependencies
POSIX.4a Single UNIX Specification, Version 3	both	POSIX(ON)

Format

```
#define _OPEN_THREADS
#include <pthread.h>

int pthread_mutex_destroy(pthread_mutex_t *mutex);
```

SUSV3:

```
#define _UNIX03_THREADS
#include <pthread.h>

int pthread_mutex_destroy(pthread_mutex_t *mutex);
```

General description

Deletes a mutex object, which identifies a mutex. Mutexes are used to protect shared resources. *mutex* is set to an invalid value, but can be reinitialized using pthread_mutex_init().

Returned value

If successful, pthread_mutex_destroy() returns 0.

If unsuccessful, pthread_mutex_destroy() returns -1 and sets errno to one of the following values:

Error Code
Description

EBUSY
A request has detected an attempt to destroy the object referenced by *mutex* while it was locked or referenced by another thread (for example, while being used in a pthread_cond_wait() or pthread_cond_timedwait() function).

EINVAL
The value specified by *mutex* is not valid.

Special behavior for Single UNIX Specification, Version 3: If unsuccessful, pthread_mutex_destroy() returns an error number to indicate the error.

Example

CELEBP36

```
/* CELEBP36 */
#define _OPEN_THREADS
#include <pthread.h>
#include <stdio.h>

main() {
  pthread_mutex_t mutex;

  if (pthread_mutex_init(&mutex, NULL) != 0) {
    perror("pthread_mutex_init() error");
    exit(1);
  }

  if (pthread_mutex_destroy(&mutex) != 0) {
    perror("pthread_mutex_destroy() error");
    exit(2);
  }
}
```

Related information

- “pthread.h” on page 58
- “pthread_cond_timedwait() — Wait on a condition variable” on page 1246
- “pthread_cond_wait() — Wait on a condition variable” on page 1249
- “pthread_mutex_init() — Initialize a mutex object”
- “pthread_mutex_lock() — Wait for a lock on a mutex object” on page 1289
- “pthread_mutex_trylock() — Attempt to lock a mutex object” on page 1291
- “pthread_mutex_unlock() — Unlock a mutex object” on page 1293

pthread_mutex_init() — Initialize a mutex object

Standards

Standards / Extensions	C or C++	Dependencies
POSIX.4a Single UNIX Specification, Version 3	both	POSIX(ON)

Format

```
#define _OPEN_THREADS
#include <pthread.h>

int pthread_mutex_init(pthread_mutex_t *mutex, pthread_mutexattr_t *attr);
```

SUSV3:

```
#define _UNIX03_THREADS
#include <pthread.h>
int pthread_mutex_init(pthread_mutex_t * __restrict__mutex,
                       const pthread_mutexattr_t * __restrict__attr);
```

General description

Creates a mutex, referenced by *mutex*, with attributes specified by *attr*. If *attr* is NULL, the default mutex attribute (NONRECURSIVE) is used.

Returned value

If successful, pthread_mutex_init() returns 0, and the state of the mutex becomes initialized and unlocked.

If unsuccessful, pthread_mutex_init() returns -1 and sets errno to one of the following values:

Error Code
    Description

EAGAIN
    The system lacked the necessary resources (other than memory) to initialize another mutex.

EBUSY
    detected an attempt to re-initialize the object referenced by *mutex*, a previously initialized, but not yet destroyed, mutex.

EINVAL
    The value specified by *attr* is not valid.

ENOMEM
    There is not enough memory to acquire a lock. This errno will only occur in the private path.

EPERM
    The caller does not have the privilege to perform the operation.

Special behavior for Single UNIX Specification, Version 3: If unsuccessful, pthread_mutex_init() returns an error number to indicate the error.

Usage notes

The _OPEN_SYS_MUTEX_EXT feature switch can be optionally included. If the feature is set, then significantly larger pthread_mutex_t objects will be defined. The feature is used for the management of mutex and condition variables in shared memory.

If the supplied extended pthread_mutex_t object is not in shared memory, pthread_mutex_init() will treat the object as a non-shared object, since it is not accessible to any other process.

It is recommended that you define and initialize the pthread_mutex_t objects in the same compile unit. If you pass a pthread_mutex_t object around to be initialized, make sure the initialization code has been compiled with the same _OPEN_SYS_MUTEX_EXT feature setting as the code that defines the object.

The following sequence may cause storage overlay with unpredictable results:

1. Declare or define a pthread_mutex_t object (in shared storage) without #define of the _OPEN_SYS_MUTEX_EXT feature. The created pthread_mutex_t object is standard size (i.e. small) without the _OPEN_SYS_MUTEX_EXT feature defined.
2. Pass the pthread_mutex_t object to another code unit, which was compiled with the _OPEN_SYS_MUTEX_EXT feature defined, to be initialized as a shared object. The pthread_mutex_t initialization generally involves the following steps:
   a. pthread_mutexattr_init()
   b. pthread_mutexattr_setpshared(). Shared pthread_mutex_t objects can be small or of extended size. The presence of the _OPEN_SYS_MUTEX_EXT feature declares it to be of extended size.
   c. pthread_mutex_init(). This step initializes the passed-in (small) pthread_mutex_t object as if it is an extended object, causing storage overlay.

Example

CELEBP37

```
/* CELEBP37 */
#ifndef _OPEN_THREADS
#define _OPEN_THREADS
#endif

#include <pthread.h>

main() {
  pthread_mutexattr_t     attr;
  pthread_mutex_t         mut;

  if (pthread_mutexattr_init(&attr) == -1) {
    perror("mutexattr_init error");
    exit(1);
  }

  if (pthread_mutex_init(&mut, &attr) == -1) {
    perror("mutex_init error");
    exit(2);
  }

  exit(0);
}
```

Related information

- “pthread.h” on page 58
- “pthread_cond_init() — Initialize a condition variable” on page 1243
- “pthread_cond_timedwait() — Wait on a condition variable” on page 1246
- “pthread_cond_wait() — Wait on a condition variable” on page 1249
- “pthread_mutexattr_init() — Initialize a mutex attribute object” on page 1301
- “pthread_mutex_lock() — Wait for a lock on a mutex object” on page 1289
- “pthread_mutex_trylock() — Attempt to lock a mutex object” on page 1291
- “pthread_mutex_unlock() — Unlock a mutex object” on page 1293

pthread_mutex_lock() — Wait for a lock on a mutex object

Standards

Standards / Extensions	C or C++	Dependencies
POSIX.4a Single UNIX Specification, Version 3	both	POSIX(ON)

Format

```
#define _OPEN_THREADS
#include <pthread.h>

int pthread_mutex_lock(pthread_mutex_t *mutex);
```

SUSV3:

```
#define _UNIX03_THREADS
#include <pthread.h>

int pthread_mutex_lock(pthread_mutex_t *mutex);
```

General description

Locks a mutex object, which identifies a mutex. Mutexes are used to protect shared resources. If the mutex is already locked by another thread, the thread waits for the mutex to become available. The thread that has locked a mutex becomes its current owner and remains the owner until the same thread has unlocked it.

When the mutex has the attribute of recursive, the use of the lock may be different. When this kind of mutex is locked multiple times by the same thread, then a count is incremented and no waiting thread is posted. The owning thread must call pthread_mutex_unlock() the same number of times to decrement the count to zero.

The mutex types are described below:

PTHREAD_MUTEX_NORMAL
A normal type mutex does not detect deadlock. That is, a thread attempting to relock this mutex without first unlocking it will deadlock. The mutex is either in a locked or unlocked state for a thread.

PTHREAD_MUTEX_ERRORCHECK
An errorcheck type mutex provides error checking. That is, a thread attempting to relock this mutex without first unlocking it will return with an error. The mutex is either in a locked or unlocked state for a thread. If a thread attempts to relock a mutex that it has already locked, it will return with an error. If a thread attempts to unlock a mutex that is unlocked, it will return with an error.

PTHREAD_MUTEX_RECURSIVE
A recursive type mutex permits a thread to lock many times. That is, a thread attempting to relock this mutex without first unlocking will succeed. This type of mutex must be unlocked the same number to times it is locked before the mutex will be returned to an unlocked state. If locked, an error is returned.

PTHREAD_MUTEX_DEFAULT
The default type mutex is mapped to a normal type mutex which does not detect deadlock. That is, a thread attempting to relock this mutex without

first unlocking it will deadlock. The mutex is either in a locked or unlocked state for a thread. The normal mutex is the default type mutex.

Returned value

If successful, pthread_mutex_lock() returns 0.

If unsuccessful, pthread_mutex_lock() returns -1 and sets errno to one of the following values:

Error Code
  Description

EAGAIN
  The mutex could not be acquired because the maximum number of recursive locks for mutex has been exceeded. This errno will only occur in the shared path.

EDEADLK
  The current thread already owns the mutex, and the mutex has a *kind* attribute of __MUTEX_NONRECURSIVE.

EINVAL
  The value specified by *mutex* is not valid.

Special behavior for Single UNIX Specification, Version 3: If unsuccessful, pthread_mutex_lock() returns an error number to indicate the error.

Usage notes

If the _OPEN_SYS_MUTEX_EXT feature switch is set, all shared (extended) mutex locks are released when the thread ends, whether normally or abnormally. If the thread ends normally (i.e. pthread_exit() or pthread_cancel()), the first waiter of the mutex lock will be resumed. If the thread ends abnormally, the processes of the mutex waiters for this mutex lock will be terminated.

Example

CELEBP38

```
/* CELEBP38 */
#ifndef _OPEN_THREADS
#define _OPEN_THREADS
#endif

#include <pthread.h>
#include <stdio.h>

main() {
  pthread_mutex_t mut;

  if (pthread_mutex_init(&mut, NULL) != 0) {
    perror("mutex_lock");
    exit(1);
  }

  if (pthread_mutex_lock(&mut) != 0) {
    perror("mutex_lock");
    exit(2);
  }
  puts("the mutex has been locked");
  exit(0);
}
```

Related information

- "pthread.h" on page 58
- "pthread_cond_init() — Initialize a condition variable" on page 1243
- "pthread_cond_wait() — Wait on a condition variable" on page 1249
- "pthread_mutex_destroy() — Delete a mutex object" on page 1285
- "pthread_mutex_init() — Initialize a mutex object" on page 1286

pthread_mutex_trylock() — Attempt to lock a mutex object

Standards

Standards / Extensions	C or C++	Dependencies
POSIX.4a Single UNIX Specification, Version 3	both	POSIX(ON)

Format

```
#define _OPEN_THREADS
#include <pthread.h>

int pthread_mutex_trylock(pthread_mutex_t *mutex);
```

SUSV3:

```
#define _UNIX03_THREADS
#include <pthread.h>

int pthread_mutex_trylock(pthread_mutex_t *mutex);
```

General description

Locks a mutex object, which identifies a mutex. Mutexes are used to protect shared resources. If pthread_mutex_trylock() is locked, it returns immediately.

For recursive mutexes, pthread_mutex_trylock() will effectively add to the count of the number of times pthread_mutex_unlock() must be called by the thread to release the mutex. (That is, it has the same behavior as a pthread_mutex_lock().)

Returned value

If successful, pthread_mutex_trylock() returns 0.

If unsuccessful, pthread_mutex_trylock() returns -1 and sets errno to one of the following values:

Error Code
  Description

EAGAIN
  The mutex could not be acquired because the maximum number of recursive locks for mutex has been exceeded. This errno will only occur in the shared path.

EBUSY
  mutex could not be acquired because it was already locked.

EINVAL
The value specified by *mutex* is not valid.

Special behavior for Single UNIX Specification, Version 3: If unsuccessful, pthread_mutex_trylock() returns an error number to indicate the error.

Usage notes

1. If the _OPEN_SYS_MUTEX_EXT feature switch is set, all shared (extended) mutex locks are released when the thread ends, whether normally or abnormally. If the thread ends normally (i.e. pthread_exit() or pthread_cancel()), the first waiter of the mutex lock will be resumed. If the thread ends abnormally, the processes of the mutex waiters for this mutex lock will be terminated.

Example

CELEBP40

```
/* CELEBP40 */
#define _OPEN_THREADS
#include <pthread.h>
#include <stdio.h>
#include <errno.h>

pthread_mutex_t mutex;

void *thread(void *arg) {
  if (pthread_mutex_trylock(&mutex) != 0)
    if (errno == EBUSY)
      puts("thread was denied access to the mutex");
    else {
      perror("pthread_mutex_trylock() error");
      exit(1);
    }
  else puts("thread was granted the mutex");
}

main() {
  pthread_t thid;

  if (pthread_mutex_init(&mutex, NULL) != 0) {
    perror("pthread_mutex_init() error");
    exit(2);
  }

  if (pthread_create(&thid, NULL, thread, NULL) != 0) {
    perror("pthread_create() error");
    exit(3);
  }
  if (pthread_mutex_trylock(&mutex) != 0)
    if (errno == EBUSY)
      puts("IPT was denied access to the mutex");
    else {
      perror("pthread_mutex_trylock() error");
      exit(4);
    }
  else puts("IPT was granted the mutex");

  if (pthread_join(thid, NULL) != 0) {
    perror("pthread_mutex_trylock() error");
    exit(5);
  }
}
```

Output:

```
IPT was granted the mutex
thread was denied access to the mutex
```

Related information

- "pthread.h" on page 58
- "pthread_mutex_destroy() — Delete a mutex object" on page 1285
- "pthread_mutex_init() — Initialize a mutex object" on page 1286

pthread_mutex_unlock() — Unlock a mutex object

Standards

Standards / Extensions	C or C++	Dependencies
POSIX.4a Single UNIX Specification, Version 3	both	POSIX(ON)

Format

```
#define _OPEN_THREADS
#include <pthread.h>

int pthread_mutex_unlock(pthread_mutex_t *mutex);
```

SUSV3:

```
#define _UNIX03_THREADS
#include <pthread.h>

int pthread_mutex_unlock(pthread_mutex_t *mutex);
```

General description

Releases a mutex object. If one or more threads are waiting to lock the mutex, pthread_mutex_unlock() causes one of those threads to return from pthread_mutex_lock() with the mutex object acquired. If no threads are waiting for the mutex, the mutex unlocks with no current owner.

When the mutex has the attribute of recursive the use of the lock may be different. When this kind of mutex is locked multiple times by the same thread, then unlock will decrement the count and no waiting thread is posted to continue running with the lock. If the count is decremented to zero, then the mutex is released and if any thread is waiting it is posted.

Returned value

If successful, pthread_mutex_unlock() returns 0.

If unsuccessful, pthread_mutex_unlock() returns -1 and sets errno to one of the following values:

Error Code
  Description

EINVAL
  The value specified by *mutex* is not valid.

EPERM
  The current thread does not own the *mutex*.

Special behavior for Single UNIX Specification, Version 3: If unsuccessful, pthread_mutex_unlock() returns an error number to indicate the error.

Example

CELEBP41

```
/* CELEBP41 */
#define _OPEN_THREADS
#include <pthread.h>
#include <stdio.h>
#include <errno.h>

pthread_mutex_t mutex;

void *thread(void *arg) {
  if (pthread_mutex_lock(&mutex) != 0) {
    perror("pthread_mutex_lock() error");
    exit(1);
  }

  puts("thread was granted the mutex");

  if (pthread_mutex_unlock(&mutex) != 0) {
    perror("pthread_mutex_unlock() error");
    exit(2);
  }
}

main() {
  pthread_t thid;

  if (pthread_mutex_init(&mutex, NULL) != 0) {
    perror("pthread_mutex_init() error");
    exit(3);
  }

  if (pthread_create(&thid, NULL, thread, NULL) != 0) {
    perror("pthread_create() error");
    exit(4);
  }
  if (pthread_mutex_lock(&mutex) != 0) {
    perror("pthread_mutex_lock() error");
    exit(5);
  }

  puts("IPT was granted the mutex");

  if (pthread_mutex_unlock(&mutex) != 0) {
    perror("pthread_mutex_unlock() error");
    exit(6);
  }

  if (pthread_join(thid, NULL) != 0) {
    perror("pthread_mutex_lock() error");
    exit(7);
  }
}
```

Output:

```
IPT was granted the mutex
thread was granted the mutex
```

Related information

- "pthread.h" on page 58
- "pthread_mutex_destroy() — Delete a mutex object" on page 1285
- "pthread_mutex_init() — Initialize a mutex object" on page 1286
- "pthread_mutex_lock() — Wait for a lock on a mutex object" on page 1289

pthread_mutexattr_destroy() — Destroy a mutex attribute object

Standards

Standards / Extensions	C or C++	Dependencies
POSIX.4a Single UNIX Specification, Version 3	both	POSIX(ON)

Format

```
#define _OPEN_THREADS
#define _OPEN_SYS
#include <pthread.h>

int pthread_mutexattr_destroy(pthread_mutexattr_t *attr);
```

SUSV3:

```
#define _UNIX03_THREADS
#define _OPEN_SYS
#include <pthread.h>

int pthread_mutexattr_destroy(pthread_mutexattr_t *attr);
```

General description

Destroys an initialized mutex attribute object. With a mutex attribute object, you can manage the characteristics of mutexes in your application. It defines the set of values to be used for the mutex during its creation. By establishing a mutex attribute object, you can create many mutexes with the same set of characteristics, without defining those characteristics for each mutex. pthread_mutexattr_init() is used to define a mutex attribute object.

Returned value

If successful, pthread_mutexattr_destroy() returns 0.

If unsuccessful, pthread_mutexattr_destroy() returns -1 and sets errno to one of the following values:

Error Code
  Description

EINVAL
  The value specified for *attr* is not valid.

Special behavior for Single UNIX Specification, Version 3: If unsuccessful, pthread_mutexattr_destroy() returns an error number to indicate the error.

Example

CELEBP42

```
/* CELEBP42 */
#define _OPEN_THREADS
#define _OPEN_SYS       /* Needed to identify __MUTEX_RESCURSIVE */
#include <pthread.h>
#include <stdio.h>

main() {
  pthread_mutexattr_t attr;
  pthread_mutex_t mutex;

  if (pthread_mutexattr_init(&attr) != 0) {
    perror("pthread_mutex_attr_init() error");
    exit(1);
  }

  if (pthread_mutexattr_setkind_np(&attr, __MUTEX_RECURSIVE) != 0) {
    perror("pthread_mutex_attr_setkind_np() error");
    exit(2);
  }

  if (pthread_mutex_init(&mutex, &attr) != 0) {
    perror("pthread_mutex_init() error");
    exit(3);
  }

  if (pthread_mutexattr_destroy(&attr) != 0) {
    perror("pthread_mutex_attr_destroy() error");
    exit(4);
  }
}
```

Related information

- "pthread.h" on page 58
- "pthread_cond_init() — Initialize a condition variable" on page 1243
- "pthread_create() — Create a thread" on page 1261
- "pthread_mutexattr_init() — Initialize a mutex attribute object" on page 1301

pthread_mutexattr_getkind_np() — Get kind from a mutex attribute object

Standards

Standards / Extensions	C or C++	Dependencies
z/OS UNIX	both	POSIX(ON)

Format

```
#define _OPEN_THREADS
#define _OPEN_SYS
#include <pthread.h>

int pthread_mutexattr_getkind_np(pthread_mutexattr_t *attr, int *kind);
```

General description

Gets the attribute *kind* from the mutex attribute object *attr*. With a mutex attribute object, you can manage the characteristics of mutexes in your application. It defines the set of values to be used for the mutex during its creation. By establishing a mutex attribute object, you can create many mutexes with the same set of characteristics without defining those characteristics for each and every mutex.

The values for the attribute *kind* are:

__MUTEX_NONRECURSIVE
: A nonrecursive mutex can be locked only once. That is, the mutex is either in a locked or unlocked state for a thread. If a thread attempts to lock a mutex that it has already locked, an error is returned.

_MUTEX_RECURSIVE
: A recursive mutex can be locked more than once by the same thread. A count of the number of times the mutex has been locked is maintained. The mutex is unlocked when pthread_mutex_unlock() is performed an equal number of times.

__MUTEX_NONRECURSIVE + __MUTEX_NODEBUG
: A nonrecursive mutex can be given an additional attribute, NODEBUG. This indicates that state changes to this mutex will *not* be reported to the debug interface, even if present.

__MUTEX_RECURSIVE + __MUTEX_NODEBUG
: A recursive mutex can be given an additional attribute, NODEBUG. This indicates that state changes to this mutex will *not* be reported to the debug interface, even if present.

Returned value

If successful, pthread_mutexattr_getkind_np() returns 0.

If unsuccessful, pthread_mutexattr_getkind_np() returns -1 and sets errno to one of the following values:

Error Code
: **Description**

EINVAL
: The value specified for *attr* is not valid.

Example

CELEBP43

```
/* CELEBP43 */

#pragma runopts(TEST(ALL))

#ifndef _OPEN_THREADS
#define _OPEN_THREADS
#define _OPEN_SYS     /* Needed to identify __MUTEX_RECURSIVE,
                         __MUTEX_NODEBUG, and __MUTEX_NONRECURSIVE  */
#endif

#include <stdio.h>
#include <pthread.h>

pthread_mutexattr_t attr;

int kind;

main() {
    if (pthread_mutexattr_init(&attr) == -1) {
       perror("pthread_mutexattr_init()");
       exit(1);
     }

    if (pthread_mutexattr_setkind_np(&attr, \
             __MUTEX_RECURSIVE + __MUTEX_NODEBUG) == -1 ) {

       perror("pthread_mutexattr_setkind_np()");
       exit(1);
     }

    if (pthread_mutexattr_getkind_np(&attr, &kind) == -1) {
       perror("pthread_mutexattr_getkind_np()");
       exit(1);
     }

     switch(kind) {

       case __MUTEX_NONRECURSIVE:
         printf("\nmutex will be nonrecursive");
         break;

       case __MUTEX_NONRECURSIVE+__MUTEX_NODEBUG:
         printf("\nmutex will be nonrecursive + nodebug");
         break;

       case __MUTEX_RECURSIVE:
         printf("\nmutex will be recursive");
         break;

       case __MUTEX_RECURSIVE+__MUTEX_NODEBUG:
         printf("\nmutex will be recursive + nodebug");
         break;

       default:
         printf("\nattribute kind value returned by \
 pthread_mutexattr_getkind_np() unrecognized");
         exit(1);
     }
    exit(0);
}
```

Output:

```
a default mutex will be nonrecursive
```

Related information

- "pthread.h" on page 58
- "pthread_mutexattr_init() — Initialize a mutex attribute object" on page 1301
- "pthread_mutexattr_setkind_np() — Set kind for a mutex attribute object" on page 1303

pthread_mutexattr_getpshared() — Get the process-shared mutex attribute

Standards

Standards / Extensions	C or C++	Dependencies
z/OS UNIX Single UNIX Specification, Version 3	both	POSIX(ON) OS/390 V2R7

Format

```
#define _OPEN_THREADS
#include <pthread.h>

int pthread_mutexattr_getpshared(const pthread_mutexattr_t *attr, int *pshared);
```

SUSV3:

```
#define _UNIX03_THREADS
#include <pthread.h>
int pthread_mutexattr_getpshared(const pthread_mutexattr_t *__restrict__attr,
                                 int * __restrict__ pshared);
```

General description

The pthread_mutexattr_getpshared() function gets the attribute *pshared* for the mutex attribute object *attr*. By using *attr* with the pthread_mutexattr_getpshared() function you can determine its *process-shared* value for a mutex.

The valid values for the attribute *pshared* are:

PTHREAD_PROCESS_SHARED
Permits a mutex to be operated upon by any thread that has access to the memory where the mutex is allocated, even if the mutex is allocated in memory that is shared by multiple processes.

PTHREAD_PROCESS_PRIVATE
A mutex can only be operated upon by threads created within the same process as the thread that initialized the mutex. When a new process is created by the parent process it will receive a different copy of the private mutex and this new mutex can only be used to serialize between threads in the child process. The default value of the attribute is `PTHREAD_PROCESS_PRIVATE`.

Returned value

If successful, pthread_mutexattr_getpshared() returns 0.

If unsuccessful, pthread_mutexattr_getpshared() returns -1 and sets errno to one of the following values:

Error Code
Description

EINVAL
The value specified for *attr* is not valid.

Special behavior for Single UNIX Specification, Version 3: If unsuccessful, pthread_mutexattr_getpshared() returns an error number to indicate the error.

Related information

- “pthread.h” on page 58
- “pthread_mutexattr_setpshared() — Set the process-shared mutex attribute” on page 1305
- “pthread_rwlockattr_getpshared() — Get the processed-shared read or write lock attribute” on page 1320
- “pthread_rwlockattr_setpshared() — Set the process-shared read or write lock attribute” on page 1322

pthread_mutexattr_gettype() — Get type of mutex attribute object

Standards

Standards / Extensions	C or C++	Dependencies
z/OS UNIX Single UNIX Specification, Version 3	both	POSIX(ON) OS/390 V2R7

Format

```
#define _OPEN_THREADS
#include <pthread.h>

int pthread_mutexattr_gettype(const pthread_mutexattr_t *attr, int *type);
```

SUSV3:

```
#define _UNIX03_THREADS
#include <pthread.h>
int pthread_mutexattr_gettype(const pthread_mutexattr_t * __restrict__attr,
                              int * __restrict__type);
```

General description

The pthread_mutexattr_gettype() function gets the attribute *type* from the mutex attribute object *attr*.

A mutex attribute object allows you to manage the characteristics of mutexes in your application. It defines the set of values to be used for the mutex during its creation. By establishing a mutex attribute object, you can create many mutexes with the same set of characteristics, without needing to define the characteristics for each and every mutex.

The values for the attribute *type* are:

PTHREAD_MUTEX_NORMAL
A normal type mutex does not detect deadlock. That is, a thread attempting to relock this mutex without first unlocking it will deadlock. The mutex is either in a locked or unlocked state for a thread.

PTHREAD_MUTEX_ERRORCHECK
An errorcheck type mutex provides error checking. That is, a thread attempting to relock this mutex without first unlocking it will return with an error. The mutex is either in a locked or unlocked state for a thread. If a thread attempts to relock a mutex that it has already locked, it will return with an error. If a thread attempts to unlock a mutex that is unlocked, it will return with an error.

PTHREAD_MUTEX_RECURSIVE
A recursive type mutex permits a thread to lock many times. That is, a thread attempting to relock this mutex without first unlocking will succeed. This type of mutex must be unlocked the same number to times it is locked before the mutex will be returned to an unlocked state. If locked, an error is returned.

PTHREAD_MUTEX_DEFAULT
The default type mutex is mapped to a normal type mutex which does not detect deadlock. That is, a thread attempting to relock this mutex without first unlocking it will deadlock. The mutex is either in a locked or unlocked state for a thread. The normal mutex is the default type mutex.

__MUTEX_NONRECURSIVE
A nonrecursive mutex can be locked only once. That is, the mutex is either in a locked or unlocked state for a thread. If a thread attempts to lock a mutex that it has already locked, an error is returned.

__MUTEX_RECURSIVE
A recursive mutex can be locked more than once by the same thread. A count of the number of times the mutex has been locked is maintained. The mutex is unlocked when pthread_mutex_unlock() is performed an equal number of times.

__MUTEX_NONRECURSIVE + __MUTEX_NODEBUG
A nonrecursive mutex can be given an additional attribute, NODEBUG. This indicates that state changes to this mutex will *not* be reported to the debug interface, even if present.

__MUTEX_RECURSIVE + __MUTEX_NODEBUG
A recursive mutex can be given an additional attribute, NODEBUG. This indicates that state changes to this mutex will *not* be reported to the debug interface, even if present.

Returned value

If successful, pthread_mutexattr_gettype() returns 0.

If unsuccessful, pthread_mutexattr_gettype() returns -1 and sets errno to one of the following values:

Error Code
Description

EINVAL
The value specified for *attr* is not valid.

Special behavior for Single UNIX Specification, Version 3:

If unsuccessful, pthread_mutexattr_gettype() returns an error number to indicate the error.

Related information

- “pthread.h” on page 58
- “pthread_mutexattr_settype() — Set type of mutex attribute object” on page 1306

pthread_mutexattr_init() — Initialize a mutex attribute object

Standards

Standards / Extensions	C or C++	Dependencies
POSIX.4a Single UNIX Specification, Version 3	both	POSIX(ON)

Format

```
#define _OPEN_THREADS
#define _OPEN_SYS
#include <pthread.h>

int pthread_mutexattr_init(pthread_mutexattr_t *attr);
```

SUSV3:

```
#define _UNIX03_THREADS
#define _OPEN_SYS
#include <pthread.h>

int pthread_mutexattr_init(pthread_mutexattr_t *attr);
```

General description

Initializes a mutex attribute object. With a mutex attribute object, you can manage the characteristics of mutexes in your application. It defines the set of values to be used for the mutex during its creation. By establishing a mutex attribute object, you can create many mutexes with the same set of characteristics, without defining those characteristics for each and every mutex.

For a valid mutex attribute, refer to “pthread_mutexattr_setkind_np() — Set kind for a mutex attribute object” on page 1303.

Note: Before freeing up the storage containing the pthread_mutexattr_t object, be sure to destroy it by calling pthread_mutexattr_destroy(). If the pthread_mutexattr_t object is not destroyed before the storage is reused, the results are undefined.

Returned value

If successful, pthread_mutexattr_init() returns 0.

If unsuccessful, pthread_mutexattr_init() returns -1 and sets errno to one of the following values:

Error Code
  Description

ENOMEM
  There is not enough memory to initialize *attr*.

Special behavior for Single UNIX Specification, Version 3:

If unsuccessful, pthread_mutexattr_init() returns an error number to indicate the error.

Example

CELEBP44

```
/* CELEBP44 */

#define _OPEN_THREADS
#define _OPEN_SYS       /* Needed to identify __MUTEX_RECURSIVE */
#include <pthread.h>
#include <stdio.h>

main() {
  pthread_mutexattr_t attr;
  pthread_mutex_t mutex;

  if (pthread_mutexattr_init(&attr) != 0) {
    perror("pthread_mutex_attr_init() error");
    exit(1);
  }
  if (pthread_mutexattr_setkind_np(&attr, __MUTEX_RECURSIVE) != 0) {
    perror("pthread_mutex_attr_setkind_np() error");
    exit(2);
  }

  if (pthread_mutex_init(&mutex, &attr) != 0) {
    perror("pthread_mutex_init() error");
    exit(3);
  }

  if (pthread_mutexattr_destroy(&attr) != 0) {
    perror("pthread_mutex_attr_destroy() error");
    exit(4);
  }
}
```

Related information

- "pthread.h" on page 58
- "pthread_cond_init() — Initialize a condition variable" on page 1243
- "pthread_create() — Create a thread" on page 1261
- "pthread_mutex_init() — Initialize a mutex object" on page 1286

pthread_mutexattr_setkind_np() — Set kind for a mutex attribute object

Standards

Standards / Extensions	C or C++	Dependencies
z/OS UNIX	both	POSIX(ON)

Format

```
#define _OPEN_THREADS
#define _OPEN_SYS
#include <pthread.h>

int pthread_mutexattr_setkind_np(pthread_mutexattr_t *attr, int kind);
```

General description

Sets the attribute *kind* for the mutex attribute object *attr*. With a mutex attribute object, you can manage the characteristics of mutexes in your application. It defines the set of values to be used for the mutex during its creation. By establishing a mutex attribute object, you can create many mutexes with the same set of characteristics, without defining those characteristics for each and every mutex.

The valid values for the attribute *kind* are:

__MUTEX_NONRECURSIVE
A nonrecursive mutex can be locked only once. That is, the mutex is either in a locked or unlocked state for a thread. If a thread attempts to lock a mutex that it has already locked, an error is returned.

__MUTEX_RECURSIVE
A recursive mutex can be locked more than once by the same thread. A count of the number of times the mutex has been locked is maintained. The mutex is unlocked when an equal number of pthread_mutex_unlock() functions are performed.

__MUTEX_NONRECURSIVE + __MUTEX_NODEBUG
A nonrecursive mutex can be given an additional attribute, NODEBUG. This indicates that state changes to this mutex will *not* be reported to the debug interface, even though it is present.

__MUTEX_RECURSIVE + __MUTEX_NODEBUG
A recursive mutex can be given an additional attribute, NODEBUG. This indicates that state changes to this mutex will *not* be reported to the debug interface, even though it is present.

Returned value

If successful, pthread_mutexattr_setkind_np() returns 0.

If unsuccessful, pthread_mutexattr_setkind_np() returns -1 and sets errno to one of the following values:

Error Code
Description

EINVAL
The value specified for *attr* or *kind* is not valid.

Example

CELEBP45

```
/* CELEBP45 */
#pragma runopts(TEST(ALL))

#ifndef _OPEN_THREADS
#define _OPEN_THREADS
#define _OPEN_SYS      /* Needed to identify __MUTEX_NODEBUG */
#endif

#include <stdio.h>
#include <pthread.h>

pthread_mutexattr_t attr;

int kind;

main() {
    if (pthread_mutexattr_init(&attr) == -1) {
       perror("pthread_mutexattr_init()");
       exit(1);
     }

    if (pthread_mutexattr_setkind_np(&attr, \
              __MUTEX_RECURSIVE + __MUTEX_NODEBUG) == -1 ) {
       perror("pthread_mutexattr_setkind_np()");
       exit(1);
     }

    if (pthread_mutexattr_getkind_np(&attr, &kind) == -1) {
       perror("pthread_mutexattr_getkind_np()");
       exit(1);
     }

     switch(kind) {

       case __MUTEX_NONRECURSIVE:
         printf("\nmutex will be nonrecursive");
         break;
```

```
        case __MUTEX_NONRECURSIVE+__MUTEX_NODEBUG:
          printf("\nmutex will be nonrecursive + nodebug");
          break;

        case __MUTEX_RECURSIVE:
          printf("\nmutex will be recursive");
          break;

        case __MUTEX_RECURSIVE+__MUTEX_NODEBUG:
          printf("\nmutex will be recursive + nodebug");
          break;

        default:
          printf("\nattribute kind value returned by \
 pthread_mutexattr_getkind_np() unrecognized");
          exit(1);
      }

    exit(0);
}
```

Related information

- "pthread.h" on page 58
- "pthread_mutexattr_init() — Initialize a mutex attribute object" on page 1301
- "pthread_mutexattr_getkind_np() — Get kind from a mutex attribute object" on page 1296

pthread_mutexattr_setpshared() — Set the process-shared mutex attribute

Standards

Standards / Extensions	C or C++	Dependencies
z/OS UNIX Single UNIX Specification, Version 3	both	POSIX(ON) OS/390 V2R7

Format

```
#define _OPEN_THREADS
#include <pthread.h>

int pthread_mutexattr_setpshared(pthread_mutexattr_t *attr, int pshared);
```

SUSV3:

```
#define _UNIX03_THREADS
#include <pthread.h>

int pthread_mutexattr_setpshared(pthread_mutexattr_t *attr, int pshared);
```

General description

The pthread_mutexattr_setpshared() function sets the attribute *pshared* for the mutex attribute object *attr*.

A mutex attribute object allows you to manage the characteristics of mutexes in your application. It defines the set of values to be used for a mutex during its creation. By establishing a mutex attribute object, you can create many mutexes with the same set of characteristics, without needing to define the characteristics for each and every mutex. By using *attr* with the pthread_mutexattr_setpshared() function you can define its *process-shared* value for a mutex.

The valid values for the attribute *pshared* are:

PTHREAD_PROCESS_SHARED
Permits a mutex to be operated upon by any thread that has access to the memory where the mutex is allocated, even if the mutex is allocated in memory that is shared by multiple processes.

PTHREAD_PROCESS_PRIVATE
A mutex can only be operated upon by threads created within the same process as the thread that initialized the mutex; if threads of differing processes attempt to operate on such a mutex, only the process to initialize the mutex will succeed. When a new process is created by the parent process it will receive a different copy of the private mutex and this new mutex can only be used to serialize between threads in the child process. The default value of the attribute is `PTHREAD_PROCESS_PRIVATE`

Returned value

If successful, pthread_mutexattr_setpshared() returns 0.

If unsuccessful, pthread_mutexattr_setpshared() returns -1 and sets errno to one of the following values:

Error Code
Description

EINVAL
The value specified for *attr* or *pshared* is not valid.

Special behavior for Single UNIX Specification, Version 3: If unsuccessful, pthread_mutexattr_setpshared() returns an error number to indicate the error.

Related information

- “pthread.h” on page 58
- “pthread_mutexattr_getpshared() — Get the process-shared mutex attribute” on page 1298
- “pthread_rwlockattr_getpshared() — Get the processed-shared read or write lock attribute” on page 1320
- “pthread_rwlockattr_setpshared() — Set the process-shared read or write lock attribute” on page 1322

pthread_mutexattr_settype() — Set type of mutex attribute object

Standards

Standards / Extensions	C or C++	Dependencies
z/OS UNIX Single UNIX Specification, Version 3	both	POSIX(ON) OS/390 V2R7

Format

```
#define _OPEN_THREADS
#include <pthread.h>

int pthread_mutexattr_settype(pthread_mutexattr_t *attr, int type);
```

SUSV3:

```
#define _UNIX03_THREADS
#include <pthread.h>

int pthread_mutexattr_settype(pthread_mutexattr_t *attr, int type);
```

General description

The pthread_mutexattr_settype() function sets the attribute *type* from the mutex attribute object *attr*.

A mutex attribute object allows you to manage the characteristics of mutexes in your application. It defines the set of values to be used for the mutex during its creation. By establishing a mutex attribute object, you can create many mutexes with the same set of characteristics, without needing to define the characteristics for each and every mutex.

The values for the attribute *type* are:

PTHREAD_MUTEX_NORMAL
A normal type mutex does not detect deadlock. That is, a thread attempting to relock this mutex without first unlocking it will deadlock. The mutex is either in a locked or unlocked state for a thread.

PTHREAD_MUTEX_ERRORCHECK
An errorcheck type mutex provides error checking. That is, a thread attempting to relock this mutex without first unlocking it will return with an error. The mutex is either in a locked or unlocked state for a thread. If a thread attempts to relock a mutex that it has already locked, it will return with an error. If a thread attempts to unlock a mutex that is unlocked, it will return with an error.

PTHREAD_MUTEX_RECURSIVE
A recursive type mutex permits a thread to lock many times. That is, a thread attempting to relock this mutex without first unlocking will succeed. This type of mutex must be unlocked the same number of times it is locked before the mutex will be returned to an unlocked state. If locked, an error is returned.

PTHREAD_MUTEX_DEFAULT
The default type mutex is mapped to a normal type mutex which does not detect deadlock. That is, a thread attempting to relock this mutex without first unlocking it will deadlock. The mutex is either in a locked or unlocked state for a thread. The normal mutex is the default type mutex.

__MUTEX_NONRECURSIVE
A nonrecursive mutex can be locked only once. That is, the mutex is either in a locked or unlocked state for a thread. If a thread attempts to lock a mutex that it has already locked, an error is returned.

__MUTEX_RECURSIVE
A recursive mutex can be locked more than once by the same thread. A count of the number of times the mutex has been locked is maintained. The mutex is unlocked when pthread_mutex_unlock() is performed an equal number of times.

__MUTEX_NONRECURSIVE + __MUTEX_NODEBUG
A nonrecursive mutex can be given an additional attribute, NODEBUG. This indicates that state changes to this mutex will *not* be reported to the debug interface, even if present.

__MUTEX_RECURSIVE + __MUTEX_NODEBUG
A recursive mutex can be given an additional attribute, NODEBUG. This indicates that state changes to this mutex will *not* be reported to the debug interface, even if present.

Returned value

If successful, pthread_mutexattr_settype() returns 0.

If unsuccessful, pthread_mutexattr_settype() returns -1 and sets errno to one of the following values:

Error Code
Description

EINVAL
Either the value type or the value specified for attr is not valid.

Special behavior for Single UNIX Specification, Version 3: If unsuccessful, pthread_mutexattr_settype() returns an error number to indicate the error.

Related information

- "pthread.h" on page 58
- "pthread_mutexattr_gettype() — Get type of mutex attribute object" on page 1300

pthread_once() — Invoke a function once

Standards

Standards / Extensions	C or C++	Dependencies
POSIX.4a Single UNIX Specification, Version 3	both	POSIX(ON)

Format

```
#define _OPEN_THREADS
#include <pthread.h>
pthread_once_t once_control = PTHREAD_ONCE_INIT;

int pthread_once(pthread_once_t *once_control, void(*init_routine)());
```

SUSV3:

```
#define _UNIX03_THREADS
#include <pthread.h>
pthread_once_t once_control = PTHREAD_ONCE_INIT;

int pthread_once(pthread_once_t *once_control, void(*init_routine)());
```

General description

Establishes a function that will be executed only once in a given process. You may have each thread call the function, but only the first call causes the function to run. This is true even if called simultaneously by multiple threads. For example, a mutex or a thread-specific data key must be created exactly once. Calling pthread_once() prevents the code that creates a mutex or thread-specific data from being called by multiple threads. Without this routine, the execution must be serialized so that only one thread performs the initialization. Other threads that reach the same point in the code are delayed until the first thread is finished.

pthread_once() is used in conjunction with a *once control* variable of the type pthread_once_t. This variable is a data type that you initialize to the PTHREAD_ONCE_INIT constant. It is then passed as a parameter on the pthread_once() function call.

init_routine is a normal function. It can be invoked directly outside of pthread_once(). In addition, it is the *once_control* variable that determines if the *init_routine* has been invoked. Calling pthread_once() with the same routine but with different *once_control* variables, will result in the routine being called twice, once for each *once_control* variable.

Returned value

If successful, pthread_once() returns 0.

If unsuccessful, pthread_once() returns -1.

There are no documented errno values. Use perror() or strerror() to determine the cause of the error.

Special behavior for Single UNIX Specification, Version 3: If unsuccessful, pthread_once() returns an error number to indicate the error.

Example

CELEBP46

```
/* CELEBP46 */
#ifndef _OPEN_THREADS
#define _OPEN_THREADS
#endif

#include <stdio.h>
#include <errno.h>
#include <pthread.h>

#define threads 3

int             once_counter=0;
pthread_once_t  once_control = PTHREAD_ONCE_INIT;

void  once_fn(void)
{
 puts("in once_fn");
 once_counter++;
}

void            *threadfunc(void *parm)
{
 int        status;
 int        threadnum;
 int        *tnum;

 tnum = parm;
 threadnum = *tnum;

 printf("Thread %d executing\n", threadnum);

 status = pthread_once(&once_control, once_fn);
 if ( status <  0)
    printf("pthread_once failed, thread %d, errno=%d\n", threadnum,
                                                            errno);

 pthread_exit((void *)0);
}

main() {
 int         status;
 int         i;
 int         threadparm[threads];
 pthread_t   threadid[threads];
 int         thread_stat[threads];

 for (i=0; i<threads; i++) {
    threadparm[i] = i+1;
    status = pthread_create( &threadid[i],
                             NULL,
                             threadfunc,
                             (void *)&threadparm[i]);
    if ( status <  0) {
       printf("pthread_create failed, errno=%d", errno);
       exit(2);
    }
  }

 for ( i=0; i<threads; i++) {
    status = pthread_join( threadid[i], (void *)&thread_stat[i]);
    if ( status <  0)
       printf("pthread_join failed, thread %d, errno=%d\n", i+1, errno);

    if (thread_stat[i] != 0)
        printf("bad thread status, thread %d, status=%d\n", i+1,
                                                 thread_stat[i]);
  }

 if (once_counter != 1)
   printf("once_fn did not get control once, counter=%d",once_counter);
 exit(0);
}
```

Output:

```
Thread 1 executing
in once_fn
Thread 2 executing
Thread 3 executing
```

Related information

- "pthread.h" on page 58

pthread_rwlock_destroy() — Destroy a read or write lock object

Standards

Standards / Extensions	C or C++	Dependencies
z/OS UNIX Single UNIX Specification, Version 3	both	POSIX(ON) OS/390 V2R7

Format

```
#define _OPEN_THREADS
#include <pthread.h>

int pthread_rwlock_destroy(pthread_rwlock_t *rwlock);
```

SUSV3:

```
#define _UNIX03_THREADS
#include <pthread.h>

int pthread_rwlock_destroy(pthread_rwlock_t *rwlock);
```

General description

The pthread_rwlock_destroy() function deletes a read or write lock object, which is identified by *rwlock* and releases any resources used by this read or write lock object. Read/write locks are used to protect shared resources.

Note: *rwlock* is set to an invalid value by pthread_rwlock_destroy() but can be reinitialized using pthread_rwlock_init().

Returned value

If successful, pthread_rwlock_destroy() returns 0.

If unsuccessful, pthread_rwlock_destroy() returns -1 and sets errno to one of the following values:

Error Code
Description

EBUSY
An attempt was made to destroy the object referenced by *rwlock* while it is locked or referenced as part of a wait on a condition variable.

EINVAL
The value specified by *rwlock* is not valid.

Special behavior for Single UNIX Specification, Version 3: If unsuccessful, pthread_rwlock_destroy() returns an error number to indicate the error.

Related information

- "pthread.h" on page 58
- "pthread_rwlock_init() — Initialize a read or write lock object" on page 1312
- "pthread_rwlock_rdlock() — Wait for a lock on a read or write lock object" on page 1313
- "pthread_rwlock_tryrdlock() — Attempt to lock a read or write lock object for reading" on page 1314
- "pthread_rwlock_trywrlock() — Attempt to lock a read or write lock object for writing" on page 1316
- "pthread_rwlock_unlock() — Unlock a read or write lock object" on page 1317
- "pthread_rwlock_wrlock() — Wait for a lock on a read or write lock object for writing" on page 1318

pthread_rwlock_init() — Initialize a read or write lock object

Standards

Standards / Extensions	C or C++	Dependencies
z/OS UNIX Single UNIX Specification, Version 3	both	POSIX(ON) OS/390 V2R7

Format

```
#define _OPEN_THREADS
#include <pthread.h>

int pthread_rwlock_init(pthread_rwlock_t *rwlock, pthread_rwlockattr_t *attr);

pthread_rwlock_t rwlock=PTHREAD_RWLOCK_INITIALIZER;
```

SUSV3:

```
#define _UNIX03_THREADS
#include <pthread.h>
int pthread_rwlock_init(pthread_rwlock_t * __restrict__rwlock,
                        const pthread_rwlockattr_t * __restrict__attr);
```

General description

The pthread_rwlock_init() function creates a read or write lock, referenced by *rwlock*, with attributes specified by *attr*. If *attr* is NULL, the default read or write lock attribute (PTHREAD_PROCESS_PRIVATE) is used. Once initialized, the lock can be used any number of times without being reinitialized. Upon successful initialization, the state of the read or write lock becomes initialized and unlocked.

In cases where default read or write lock attributes are appropriate, the macro PTHREAD_RWLOCK_INITIALIZER can be used to initialize read or write locks that are statically allocated. The effect is equivalent to dynamic initialization by a call to pthread_rwlock_init() with parameter *attr* specified as NULL, except that no error checking is done.

Note: Although the SUSv3 standard does not specify a static initializer for read or write locks, the implementation-defined macro PTHREAD_RWLOCK_INITIALIZER_NP may be used for that purpose. It is functionally equivalent in the SUSv3 context to the PTHREAD_RWLOCK_INITIALIZER macro.

Returned value

If successful, pthread_rwlock_init() returns 0, and the state of the read or write lock becomes initialized and unlocked.

If unsuccessful, pthread_rwlock_init() returns -1 and sets errno to one of the following values:

Error Code
Description

EAGAIN
The system lacked necessary resources (other than memory) to initialize another read or write lock.

EINVAL
The value specified by *attr* is not valid.

ENOMEM
There is not enough memory to initialize the read or write lock.

EPERM
The caller does not have the privilege to perform the operation.

Special behavior for Single UNIX Specification, Version 3: If unsuccessful, pthread_rwlock_init() returns an error number to indicate the error.

Related information

- "pthread.h" on page 58
- "pthread_rwlock_destroy() — Destroy a read or write lock object" on page 1310
- "pthread_rwlock_rdlock() — Wait for a lock on a read or write lock object"
- "pthread_rwlock_tryrdlock() — Attempt to lock a read or write lock object for reading" on page 1314
- "pthread_rwlock_trywrlock() — Attempt to lock a read or write lock object for writing" on page 1316
- "pthread_rwlock_unlock() — Unlock a read or write lock object" on page 1317
- "pthread_rwlock_wrlock() — Wait for a lock on a read or write lock object for writing" on page 1318

pthread_rwlock_rdlock() — Wait for a lock on a read or write lock object

Standards

Standards / Extensions	C or C++	Dependencies
z/OS UNIX Single UNIX Specification, Version 3	both	POSIX(ON) OS/390 V2R7

Format

```
#define _OPEN_THREADS
#include <pthread.h>

int pthread_rwlock_rdlock(pthread_rwlock_t *rwlock);
```

SUSV3:

```
#define _UNIX03_THREADS
#include <pthread.h>

int pthread_rwlock_rdlock(pthread_rwlock_t *rwlock);
```

General description

The pthread_rwlock_rdlock() function applies a read lock to the read or write lock referenced by *rwlock*. The calling thread acquires the read lock if a writer does not hold the lock and there are no writers blocked on the lock. In z/OS UNIX, the calling thread does not acquire the lock when a writer does not hold the lock and there are writers waiting for the lock unless the thread already held *rwlock* for read. It will block and wait until there are no writers holding or waiting for the read or write lock. If a writer holds the lock, the calling thread will not acquire the read lock. If the read lock is not acquired, the calling thread blocks (that is, it does not return from the pthread_rwlock_rdlock() call) until it can acquire the lock.

A thread may hold multiple concurrent read locks on *rwlock* (that is successfully call the pthread_rwlock_rdlock() function n times). If so, the thread must perform matching unlocks (that is, it must call the pthread_rwlock_unlock() function n times). Read/write locks are used to protect shared resources.

Note: If a thread owns locks at the time it is terminated then z/OS UNIX will release those locks.

Returned value

If successful, pthread_rwlock_rdlock() returns 0.

If unsuccessful, pthread_rwlock_rdlock() returns -1 and sets errno to one of the following values:

Error Code
 Description

EAGAIN
 The read lock could not be acquired because the maximum number of read locks for *rwlock* has been exceeded. This errno will only occur in the shared path.

EDEADLK
 The current thread already owns the read or write lock for writing.

EINVAL
 The value specified by *rwlock* is not valid.

ENOMEM
 There is not enough memory to acquire a lock. This errno will only occur in the private path.

Special behavior for Single UNIX Specification, Version 3: If unsuccessful, pthread_rwlock_rdlock() returns an error number to indicate the error.

Related information

- "pthread.h" on page 58
- "pthread_rwlock_destroy() — Destroy a read or write lock object" on page 1310
- "pthread_rwlock_init() — Initialize a read or write lock object" on page 1312

pthread_rwlock_tryrdlock() — Attempt to lock a read or write lock object for reading

Standards

Standards / Extensions	C or C++	Dependencies
z/OS UNIX Single UNIX Specification, Version 3	both	POSIX(ON) OS/390 V2R7

Format

```
#define _OPEN_THREADS
#include <pthread.h>

int pthread_rwlock_tryrdlock(pthread_rwlock_t *rwlock);
```

SUSV3:

```
#define _UNIX03_THREADS
#include <pthread.h>

int pthread_rwlock_tryrdlock(pthread_rwlock_t *rwlock);
```

General description

The pthread_rwlock_tryrdlock() function applies a read lock as in the pthread_rwlock_rdlock() function with the exception that the function fails if any thread holds a write lock on *rwlock* or there are writers blocked on *rwlock* unless the thread already held *rwlock* for read. Read/write locks are used to protect shared resources.

If the read or write lock identified by *rwlock* is locked, pthread_rwlock_tryrdlock() returns immediately.

When there are only read locks on the read or write lock, pthread_rwlock_tryrdlock() will effectively add to the count of the number of times pthread_rwlock_unlock() must be called by the thread to release the mutex (that is, it has the same behavior as a pthread_rwlock_rdlock() function).

Returned value

If successful, pthread_rwlock_tryrdlock() returns 0.

If unsuccessful, pthread_rwlock_tryrdlock() returns -1 and sets errno to one of the following values:

Error Code
    Description

EAGAIN
    The read lock could not be acquired because the maximum number of read locks for *rwlock* has been exceeded. This errno will only occur in the shared path.

EBUSY
    rwlock could not be acquired because it was already locked.

EINVAL
    The value specified by *rwlock* is not valid.

ENOMEM
    There is not enough memory to acquire a lock. This errno will only occur in the private path.

Special behavior for Single UNIX Specification, Version 3: If unsuccessful, pthread_rwlock_tryrdlock() returns an error number to indicate the error.

Related information

- “pthread.h” on page 58
- “pthread_rwlock_destroy() — Destroy a read or write lock object” on page 1310
- “pthread_rwlock_init() — Initialize a read or write lock object” on page 1312

pthread_rwlock_trywrlock() — Attempt to lock a read or write lock object for writing

Standards

Standards / Extensions	C or C++	Dependencies
z/OS UNIX Single UNIX Specification, Version 3	both	POSIX(ON) OS/390 V2R7

Format

```
#define _OPEN_THREADS
#include <pthread.h>

int pthread_rwlock_trywrlock(pthread_rwlock_t *rwlock);
```

SUSV3:

```
#define _UNIX03_THREADS
#include <pthread.h>

int pthread_rwlock_trywrlock(pthread_rwlock_t *rwlock);
```

General description

The pthread_rwlock_trywrlock() function applies a write lock as in the pthread_rwlock_wrlock() function with the exception that the function fails if any thread holds either a read lock or a write lock on *rwlock*. Read/write locks are used to protect shared resources.

If the read or write lock identified by *rwlock* is locked, pthread_rwlock_trywrlock() returns immediately.

Returned value

If successful, pthread_rwlock_trywrlock() returns 0.

If unsuccessful, pthread_rwlock_trywrlock() returns -1 and sets errno to one of the following values:

Error Code
  Description

EBUSY
  rwlock could not be acquired because it was already locked.

EINVAL
  The value specified by *rwlock* is not valid.

ENOMEM
  There is not enough memory to acquire a lock. This errno will only occur in the private path.

Special behavior for Single UNIX Specification, Version 3: If unsuccessful, pthread_rwlock_trywrlock() returns an error number to indicate the error.

Related information

- “pthread.h” on page 58
- “pthread_rwlock_destroy() — Destroy a read or write lock object” on page 1310

- “pthread_rwlock_init() — Initialize a read or write lock object” on page 1312

pthread_rwlock_unlock() — Unlock a read or write lock object

Standards

Standards / Extensions	C or C++	Dependencies
z/OS UNIX Single UNIX Specification, Version 3	both	POSIX(ON) OS/390 V2R7

Format

```
#define _OPEN_THREADS
#include <pthread.h>

int pthread_rwlock_unlock(pthread_rwlock_t *rwlock);
```

SUSV3:

```
#define _UNIX03_THREADS
#include <pthread.h>

int pthread_rwlock_unlock(pthread_rwlock_t *rwlock);
```

General description

The pthread_rwlock_unlock() function releases a read or write lock object. If one or more threads are waiting to lock the rwlock, pthread_rwlock_unlock() causes one or more of these threads to return from the pthread_rwlock_rdlock() or the pthread_rwlock_wrlock() call with the read or write lock object acquired. If there are multiple threads blocked on *rwlock* for both read locks and write locks, z/OS UNIX will give the read or write lock to the next waiting call whether it is a read or a write request even when there is a writer blocked waiting for the lock. If no threads are waiting for the rwlock, the rwlock unlocks with no current owner.

Returned value

If successful, pthread_rwlock_unlock() returns 0.

If unsuccessful, pthread_rwlock_unlock() returns -1 and sets errno to one of the following values:

Error Code
 Description

EINVAL
 The value specified for *rwlock* is not valid.

ENOMEM
 There is not enough memory during the unlock process. This errno will only occur in the private path.

EPERM
 The current thread does not own the read_write lock object.

Special behavior for Single UNIX Specification, Version 3: If unsuccessful, pthread_rwlock_unlock() returns an error number to indicate the error.

Related information

- “pthread.h” on page 58
- “pthread_rwlock_destroy() — Destroy a read or write lock object” on page 1310
- “pthread_rwlock_init() — Initialize a read or write lock object” on page 1312
- “pthread_rwlock_rdlock() — Wait for a lock on a read or write lock object” on page 1313
- “pthread_rwlock_tryrdlock() — Attempt to lock a read or write lock object for reading” on page 1314
- “pthread_rwlock_trywrlock() — Attempt to lock a read or write lock object for writing” on page 1316
- “pthread_rwlock_wrlock() — Wait for a lock on a read or write lock object for writing”

pthread_rwlock_wrlock() — Wait for a lock on a read or write lock object for writing

Standards

Standards / Extensions	C or C++	Dependencies
z/OS UNIX Single UNIX Specification, Version 3	both	POSIX(ON) OS/390 V2R7

Format

```
#define _OPEN_THREADS
#include <pthread.h>

int pthread_rwlock_wrlock(pthread_rwlock_t *rwlock);
```

SUSV3:

```
#define _UNIX03_THREADS
#include <pthread.h>

int pthread_rwlock_wrlock(pthread_rwlock_t *rwlock);
```

General description

The pthread_rwlock_wrlock() function applies a write lock to the read or write lock referenced by *rwlock*. The calling thread acquires the write lock if no other thread (reader or writer) holds the read or write lock *rwlock*. Otherwise, the thread blocks (that is, does not return from the pthread_rwlock_wrlock() call) until it can acquire the lock. In z/OS UNIX the calling thread does not acquire the lock when a writer does not hold the lock and there are writers waiting for the lock. It will block and wait until there are no writers holding or waiting for the read or write lock. If the thread already holds read or write lock for either read or write then a deadlock errno will be returned.

Note: If a thread owns locks at the time it is terminated then z/OS UNIX will release those locks.

Returned value

If successful, pthread_rwlock_wrlock() returns 0.

If unsuccessful, pthread_rwlock_wrlock() returns -1 and sets errno to one of the following values:

Error Code
Description

EDEADLK
The current thread already owns the read or write lock for writing or reading.

EINVAL
The value specified by *rwlock* is not valid.

ENOMEM
There is not enough memory to acquire a lock. This errno will only occur in the private path.

Special behavior for Single UNIX Specification, Version 3:

If unsuccessful, pthread_rwlock_wrlock() returns an error number to indicate the error.

Related information

- “pthread.h” on page 58
- “pthread_rwlock_destroy() — Destroy a read or write lock object” on page 1310
- “pthread_rwlock_init() — Initialize a read or write lock object” on page 1312

pthread_rwlockattr_destroy() — Destroy a read or write lock attribute object

Standards

Standards / Extensions	C or C++	Dependencies
z/OS UNIX Single UNIX Specification, Version 3	both	POSIX(ON) OS/390 V2R7

Format

```
#define _OPEN_THREADS
#include <pthread.h>

int pthread_rwlockattr_destroy(pthread_rwlockattr_t *attr);
```

SUSV3:

```
#define _UNIX03_THREADS
#include <pthread.h>

int pthread_rwlockattr_destroy(pthread_rwlockattr_t *attr);
```

General description

The pthread_rwlockattr_destroy() function destroys an initialized rwlock attribute object.

After a read or write lock attributes object has been used to initialize one or more read or write locks any function affecting the attributes object (including destruction) does not affect any previously initialized read or write locks.

The pthread_rwlockattr_destroy() function destroys a read or write lock attributes object. Subsequent use of the object will cause an error until the object is reinitialized by another call to pthread_rwlockattr_init().

Returned value

If successful, pthread_rwlockattr_destroy() returns 0.

If unsuccessful, pthread_rwlockattr_destroy() returns -1 and sets errno to one of the following values:

Error Code
Description

EINVAL
The value specified for *attr* is not valid.

Special behavior for Single UNIX Specification, Version 3: If unsuccessful, pthread_rwlockattr_destroy() returns an error number to indicate the error.

Related information

- "pthread.h" on page 58
- "pthread_rwlockattr_init() — Initialize a read or write lock attribute object" on page 1321

pthread_rwlockattr_getpshared() — Get the processed-shared read or write lock attribute

Standards

Standards / Extensions	C or C++	Dependencies
z/OS UNIX Single UNIX Specification, Version 3	both	POSIX(ON) OS/390 V2R7

Format

```
#define _OPEN_THREADS
#include <pthread.h>

int pthread_rwlockattr_getpshared(const pthread_rwlockattr_t *attr, int *pshared);
```

SUSV3:

```
#define _UNIX03_THREADS
#include <pthread.h>
int pthread_rwlockattr_getpshared(const pthread_rwlockattr_t *
                                   __restrict__attr,
                                  int * __restrict__pshared);
```

General description

The pthread_rwlockattr_getpshared() function gets the attribute *pshared* for the read or write lock attribute object *attr*. By using *attr* with the pthread_rwlockattr_getpshared() function you can determine its *process-shared* value for a read or write lock.

The valid values for the attribute *pshared* are:

PTHREAD_PROCESS_SHARED
Permits a read or write lock to be operated upon by any thread that has access to the memory where the read or write lock is allocated, even if the read or write lock is allocated in memory that is shared by multiple processes.

PTHREAD_PROCESS_PRIVATE
A read or write lock can only be operated upon by threads created within the same process as the thread that initialized the read or write lock. When a new process is created by the parent process it will receive a different copy of the private read or write lock and this new read or write lock can only be used to serialize between threads in the child process. The default value of the attributed is `PTHREAD_PROCESS_PRIVATE`.

Returned value

If successful, pthread_rwlockattr_getpshared() returns 0.

If unsuccessful, pthread_rwlockattr_getpshared() returns -1 and sets errno to one of the following values:

Error Code
Description

EINVAL
The value specified for *attr* is not valid.

Special behavior for Single UNIX Specification, Version 3: If unsuccessful, pthread_rwlockattr_getpshared() returns an error number to indicate the error.

Related information

- "pthread.h" on page 58
- "pthread_rwlock_init() — Initialize a read or write lock object" on page 1312
- "pthread_rwlockattr_setpshared() — Set the process-shared read or write lock attribute" on page 1322

pthread_rwlockattr_init() — Initialize a read or write lock attribute object

Standards

Standards / Extensions	C or C++	Dependencies
z/OS UNIX Single UNIX Specification, Version 3	both	POSIX(ON) OS/390 V2R7

Format

```
#define _OPEN_THREADS
#include <pthread.h>

int pthread_rwlockattr_init(pthread_rwlockattr_t *attr);
```

SUSV3:

```
#define _UNIX03_THREADS
#include <pthread.h>

int pthread_rwlockattr_init(pthread_rwlockattr_t *attr);
```

General description

The pthread_rwlockattr_init() function initializes a read or write lock attribute object. A read or write lock attribute object allows you to manage the characteristics of read or write locks in your application. It defines the set of values to be used for the read or write lock during its creation. By establishing a read or write lock attribute object, you can create many read or write locks with the same set of characteristics, without needing to define the characteristics for each and every read or write lock.

For a valid read or write lock attribute, refer to "pthread_rwlockattr_setpshared() — Set the process-shared read or write lock attribute."

If pthread_rwlockattr_init() is called specifying an already initialized read or write lock attributes object the request is rejected and the current lock attributes object is unchanged.

Returned value

If successful, pthread_rwlockattr_init() returns 0.

If unsuccessful, pthread_rwlockattr_init() returns -1 and sets errno to one of the following values:

Error Code
Description

ENOMEM
There is not enough memory to initialize *attr*.

Special behavior for Single UNIX Specification, Version 3: If unsuccessful, pthread_rwlockattr_init() returns an error number to indicate the error.

Related information

- "pthread.h" on page 58
- "pthread_rwlock_init() — Initialize a read or write lock object" on page 1312

pthread_rwlockattr_setpshared() — Set the process-shared read or write lock attribute

Standards

Standards / Extensions	C or C++	Dependencies
z/OS UNIX Single UNIX Specification, Version 3	both	POSIX(ON) OS/390 V2R7

Format

```
#define _OPEN_THREADS
#include <pthread.h>

int pthread_rwlockattr_setpshared(pthread_rwlockattr_t *attr, int pshared);
```

SUSV3:

```
#define _UNIX03_THREADS
#include <pthread.h>

int pthread_rwlockattr_setpshared(pthread_rwlockattr_t *attr, int pshared);
```

General description

The pthread_rwlockattr_setpshared() function sets the attribute *pshared* for the read or write lock attribute object *attr*.

A read or write lock attribute object allows you to manage the characteristics of read or write locks in your application. It defines the set of values to be used for a read or write lock during its creation. By establishing a read or write lock attribute object, you can create many read or write locks with the same set of characteristics, without needing to define those characteristics for each and every read or write lock. By using *attr* with the pthread_rwlockattr_setpshared() function you can define its *process-shared* value for a read or write lock.

The valid values for the attribute *pshared* are:

PTHREAD_PROCESS_SHARED
Permits a read or write lock to be operated upon by any thread that has access to the memory where the read or write lock is allocated, even if the read or write lock is allocated in memory that is shared by multiple processes.

PTHREAD_PROCESS_PRIVATE
A read or write lock can only be operated upon by threads created within the same process as the thread that initialized the read or write lock. When a new process is created by the parent process it will receive a different copy of the private read or write lock and this new read or write lock can only be used to serialize between threads in the child process. The default value of the attributed is `PTHREAD_PROCESS_PRIVATE`.

Returned value

If successful, pthread_rwlockattr_setpshared() returns 0.

If unsuccessful, pthread_rwlockattr_setpshared() returns -1 and sets errno to one of the following values:

Error Code
Description

EINVAL
The value specified for *attr* or *pshared* is not valid.

Special behavior for Single UNIX Specification, Version 3: If unsuccessful, pthread_rwlockattr_setpshared() returns an error number to indicate the error.

Related information

- “pthread.h” on page 58
- “pthread_rwlock_init() — Initialize a read or write lock object” on page 1312
- “pthread_rwlockattr_getpshared() — Get the processed-shared read or write lock attribute” on page 1320

pthread_security_np(), pthread_security_applid_np() — Create or delete thread-level security

Standards

Standards / Extensions	C or C++	Dependencies
z/OS UNIX	both	POSIX(ON)

Format

```
#define  _OPEN_SYS  1
#include <pthread.h>

int pthread_security_np(int function_code,
                        int identity_type,
                        size_t identity_length,
                        void *identity,
                        char *password,
                        int options);

int pthread_security_applid_np(int function_code,
                        int identity_type,
                        size_t identity_length,
                        void *identity,
                        char *password,
                        int options
         const char *applid);
```

General description

pthread_security_np() creates or deletes a thread-level security environment for the calling thread.

The __pthread_security_applid_np() function is equivalent to pthread_security_np() with the added feature that it also allows the application identifier (APPLID) to be supplied that will be passed on to the security product to assist with authentication. This is useful, for example, in situations where a pass ticket is provided and the pass ticket was created with a USERID/APPLID combination. When applid is NULL or a pointer to NULL, no application identifier will be passed on to the security product.

The function supports the following parameters:

Parameter
Description

function_code
Specify one of the following:

__CREATE_SECURITY_ENV
Create a thread-level security environment for the calling thread. If a thread-level security environment already exists, it is deleted before a new one is created.

__DAEMON_SECURITY_ENV

Creates a thread-level security environment for the caller's thread without the need for a password if the caller is a superuser and has permission to BPX.DAEMON facility class profile if BPX.DAEMON facility class profile is defined. If a thread-level security environment already exists, it is deleted before the new environment is created. Using the _DAEMON_SECURITY_ENV function code and not specifying a password is similar to using the current BPX.SRV.userid surrogate support. The difference is that the installation does not have to setup individual surrogate profiles for each of the clients that desire a thread level identity in the target server process.

The server will be allowed to create any identity without authentication if it is given permission to the BPX.DAEMON facility class profile.

__DELETE_SECURITY_ENV
Delete the thread-level security environment for the calling thread, if one exists. If the security environment was created using the `__TLS_TASK_ACEE` option, then only the z/OS UNIX security data is deleted (the task-level ACEE is unchanged).

__TLS_TASK_ACEE
Initializes the z/OS UNIX security data for a task that has an existing task-level security environment (task-level ACEE). If the z/OS UNIX security data already exists for the calling task, the existing z/OS UNIX security data is deleted and a new set of z/OS UNIX security data is established.

__TLS_TASK_ACEE_USP
Takes a pre-existing user security packet (USP) from a task-level ACEE and extracts the UID and GID information. This information is then used to build a complete z/OS UNIX security environment for the calling thread. If the calling thread does not have a USP associated with the task-level ACEE, this call is treated as if the __TLS_TASK_ACEE function was specified.

identity_type
Specifies the format of the user identity in the argument *identity*. It can have one of the following values:

__USERID_IDENTITY
User identity in the form of a character string (1 to 8 bytes in length).

__CERTIFICATE_IDENTITY
User identity in the form of a `__certificate_t`.

A `__certificate_t` is a structure containing the following elements:

__cert_type
The type of security certificate. Setting value __CERT_X509, for example, indicates the certificate is an X.509 security certificate.

__userid
An output field in the *__certificate* structure that will be filled with the user ID associated with the certificate. This output will be up to 8 characters long and NULL-terminated.

__cert_length
The length in bytes of the security certificate.

__cert_ptr
A pointer to the start of the security certificate.

identity_length
Specifies the length of the *identity* parameter. If *identity_type* is __USERID_IDENTITY, *identity_length* is the length of the user identity character string. If *identity_type* is __CERTIFICATE_IDENTITY, *identity_length* is the length of the *__certificate* structure.

identity
Specifies the user identity according to the *identity_type* parameter.

password
Specifies a user password or PassTicket, or a password phrase.

options Specifies options used to tailor the request. *options* must be set to 0.

applid Specifies the application identifier that will be used for authentication with the security product.

This function is intended to be used by servers which process requests from multiple clients. By creating and building a thread-level security environment for the client, a server can process many client requests without the overhead of issuing *fork*/*setuid*/*exec*. See usage notes in *z/OS UNIX System Services Programming: Assembler Callable Services Reference* for additional information.

Returned value

If successful, pthread_security_np() returns 0.

If unsuccessful, pthread_security_np() returns -1 and sets errno to one the following values:

Error Code
Description

EACCES
The password or PassTicket, or the password phrase provided is not valid for the passed userid.

EINVAL
A parameter is not valid.

EMVSERR
An MVS environmental error or internal error occurred.

EMVSEXPIRE
The password or PassTicket, or the password phrase provided has expired.

EMVSSAF2ERR
The SAF call to the security product incurred an error.

EMVSSAFEXTRERR
The SAF or RACF RACROUTE EXTRACT service incurred an error. One possible reason is that the user's access was revoked.

ENOSYS
The function is not implemented.

EPERM
One or more of the following conditions were detected:

- The process does not have appropriate privileges to set a thread-level security environment. The caller is not permitted to the BPX.SERVER FACILITY class profile or BPX.SERVER is not defined and the caller is not a superuser.

- No password, PassTicket or password phrase is provided and the BPX.SRV SURROGAT class has not been activated; or no BPX.SRV SURROGAT class profile has been defined for the specified user identity.
- No password, PassTicket or password phrase is provided and the caller is not permitted to the specified user identity's BPX.SRV SURROGAT class profile.
- For function_code __DAEMON_SECURITY_ENV, the caller is not a superuser; or the BPX.DAEMON FACILITY class is defined and the caller is not permitted to the BPX.DAEMON FACILITY class profile.
- A load from an unauthorized (not Program Control protected) library was done in the address space.

ESRCH

The user ID provided as input is not defined to the security product or does not have an OMVS segment defined.

Related information

- "pthread.h" on page 58
- "getlogin() — Get the user login name" on page 731

pthread_self() — Get the caller

Standards

Standards / Extensions	C or C++	Dependencies
POSIX.4a Single UNIX Specification, Version 3	both	POSIX(ON)

Format

```
#define _OPEN_THREADS
#include <pthread.h>

pthread_t pthread_self(void);
```

General description

Returns the thread ID of the calling thread.

Returned value

There are no documented errno values. Use perror() or strerror() to determine the cause of the error.

Example

CELEBP47

```
/* CELEBP47 */
#define _OPEN_THREADS
#include <pthread.h>
#include <stdio.h>

pthread_t thid, IPT;

void *thread(void *arg) {
  if (pthread_equal(IPT, pthread_self()))
    puts("the thread is the IPT...?");
  else
```

```
    puts("the thread is not the IPT");

  if (pthread_equal(thid, pthread_self()))
    puts("the thread is the one created by the IPT");
  else
    puts("the thread is not the one created by the IPT...?");
}

main() {
  IPT = pthread_self();
  if (pthread_create(&thid, NULL, thread, NULL) != 0) {
    perror("pthread_create() error");
    exit(1);
  }

  if (pthread_join(thid, NULL) != 0) {
    perror("pthread_create() error");
    exit(3);
  }
}
```

Output:

```
the thread is not the IPT
the thread is the one created by the IPT
```

Related information

- "pthread.h" on page 58
- "pthread_create() — Create a thread" on page 1261
- "pthread_equal() — Compare thread IDs" on page 1265

pthread_setcancelstate() — Set a thread cancelability state format

Standards

Standards / Extensions	C or C++	Dependencies
Single UNIX Specification, version 3	both	z/OS V1R7 POSIX(ON)

Format

```
#define _OPEN_THREADS 2
#include <pthread.h>

int pthread_setcancelstate(int state, int *oldstate);
```

General description

pthread_setcancelstate() controls whether the thread acts on a cancelation request caused by a call to pthread_cancel(). The old *state* is stored into the location pointed to by *oldstate*. The cancelability states can be:

PTHREAD_CANCEL_ENABLE
The thread can be canceled, but is subject to type. The cancelability types can be found in "pthread_setcanceltype() — Set a thread cancelability type format" on page 1329.

PTHREAD_CANCEL_DISABLE
The thread cannot be canceled.

Returned value

If successful, pthread_setcancelstate() returns 0. Upon failure, returns the following **EINVAL** error code:
- *state* is an invalid value.

Related information

- "Thread Cancellation" in the *z/OS XL C/C++ Programming Guide*
- "pthread.h" on page 58
- "pthread_cancel() — Cancel a thread" on page 1234
- "pthread_setcanceltype() — Set a thread cancelability type format"
- "pthread_testcancel() — Establish a cancelation point" on page 1342

pthread_setcanceltype() — Set a thread cancelability type format

Standards

Standards / Extensions	C or C++	Dependencies
Single UNIX Specification, version 3	both	z/OS V1R7 POSIX(ON)

Format

```
#define _OPEN_THREADS 2
#include <pthread.h>

int pthread_setcanceltype(int type, int *oldtype);
```

General description

pthread_setcanceltype() controls when a cancel request is acted on. The old *type* is stored into the location pointed to by *oldtype*. The cancelability types can be:

PTHREAD_CANCEL_ASYNCHRONOUS
The thread can be canceled at any time.

PTHREAD_CANCEL_DEFERRED
The thread can be canceled, but only at cancelation points introduced by invocation of particular functions. For more information, see the *z/OS XL C/C++ Programming Guide*.

Returned value

If successful, pthread_setcanceltype() returns 0. Upon failure, returns the following **EINVAL** error code:
- *type* is an invalid value.

Related information

- "Thread Cancelation" in the *z/OS XL C/C++ Programming Guide*
- "pthread.h" on page 58
- "pthread_cancel() — Cancel a thread" on page 1234
- "pthread_setcancelstate() — Set a thread cancelability state format" on page 1328
- "pthread_testcancel() — Establish a cancelation point" on page 1342

pthread_setconcurrency() — Set the level of concurrency

Standards

Standards / Extensions	C or C++	Dependencies
Single UNIX Specification, version 3	both	z/OS V1R7 POSIX(ON)

Format

```
#define _OPEN_THREADS 2
#include <pthread.h>

int pthread_setconcurrency(int new_level);
```

General description

pthread_setconcurrency() sets the desired thread concurrency level to *new_level*.

Usage notes

1. z/OS UNIX does not support multiplexing POSIX threads onto TCBs. If successful, pthread_setconcurrency() saves *new_level* for subsequent calls to pthread_getconcurrency() but takes no other action. For related information on the relationship between pthreads and TCBs, see "pthread_attr_setweight_np() — Set weight of thread attribute object" on page 1232 and "pthread_attr_setsynctype_np() — Set thread sync type" on page 1231.

Returned value

If successful, pthread_setconcurrency() returns 0. Upon failure, returns one of the following error values:

- EINVAL – The value specified by *new_level* is negative.
- EAGAIN – The value specific by *new_level* would cause a system resource to be exceeded.

Related information

- "Thread Cancelation" in the *z/OS XL C/C++ Programming Guide*
- "pthread.h" on page 58
- "pthread_getconcurrency() — Get the level of concurrency" on page 1268

pthread_setintr() — Set a thread cancelability state

Standards

Standards / Extensions	C or C++	Dependencies
POSIX.4a	both	POSIX(ON)

Format

```
#define _OPEN_THREADS
#include <pthread.h>

int pthread_setintr(int state);
```

General description

Controls whether the thread accepts a cancel request that was produced by a call to pthread_cancel(). The cancelability states can be:

PTHREAD_INTR_DISABLE
The thread cannot be canceled.

PTHREAD_INTR_ENABLE
The thread can be canceled, but it is subject to type. The cancelability types can be found in "pthread_setintrtype() — Set a thread cancelability type" on page 1333.

Usage notes

1. If you are writing to the Single UNIX Specification, Version 3 standard, use pthread_setcancelstate() in place of pthread_setintr().

Returned value

If successful, pthread_setintr() returns the previous state.

If unsuccessful, pthread_setintr() returns -1 and sets errno to one of the following values:

Error Code
Description

EINVAL
state is an invalid value.

Example

CELEBP48

```
/* CELEBP48 */
#define  _OPEN_THREADS
#include <stdio.h>
#include <string.h>
#include <pthread.h>
#include <errno.h>
#include <unistd.h>

int  thstatus;
char state[60] = "enable/controlled - initial default";

void * thfunc(void *voidptr)
{
 int  rc;
 char *parmptr;

 parmptr = voidptr;
 printf("parm = %s.\n", parmptr);

 strcpy(state, "disable/controlled");
 if ( pthread_setintrtype(PTHREAD_INTR_CONTROLLED ) == -1 ) {
   printf("set controlled failed. %s\n", strerror(errno));
   thstatus = 103;
   pthread_exit(&thstatus);
 }

 if ( pthread_setintr(PTHREAD_INTR_ENABLE) == -1 ) {
   printf("set enable failed. %s\n", strerror(errno));
   thstatus = 104;
   pthread_exit(&thstatus);
 }

 strcpy(state, "enable/controlled");

 strcat(state, " - pthread_testintr");

 while (1) {
   pthread_testintr();
   sleep(1);
 }

 thstatus = 100;
 pthread_exit(&thstatus);
}

 main(int argc, char *argv[]) {
  int            rc;
  pthread_attr_t attrarea;
  pthread_t      thid;
  char           parm[] = "abcdefghijklmnopqrstuvwxyz";
  int            *statptr;

  if ( pthread_attr_init(&attrarea) == -1 ) {
    printf("pthread_attr_init failed. %s\n", strerror(errno));
    exit(1);
  }

  if ( pthread_create(&thid, &attrarea, thfunc, (void *)&parm) == -1) {
    printf("pthread_create failed. %s\n", strerror(errno));
    exit(2);
  }

  sleep(5);

  if ( pthread_cancel(thid) == -1 ) {
    printf("pthread_cancel failed. %s\n", strerror(errno));
    exit(3);
  }

   if ( pthread_join(thid, (void **)&statptr)== -1 ) {
    printf("pthread_join failed. %s\n", strerror(errno));
    exit(4);
  }

  if ( statptr == (int *)-1 )
    printf("thread was cancelled. state = %s.\n", state);
  else
    printf("thread was not cancelled. thstatus = %d.\n", *statptr);

  exit(0);
 }
```

Output:

```
parm = abcdefghijklmnopqrstuvwxyz.
thread was canceled. state = enable/controlled - pthread_testintr.
```

Related information

- "pthread.h" on page 58
- "pthread_cancel() — Cancel a thread" on page 1234
- "pthread_setintrtype() — Set a thread cancelability type" on page 1333
- "pthread_testintr() — Establish a cancelability point" on page 1342

pthread_setintrtype() — Set a thread cancelability type

Standards

Standards / Extensions	C or C++	Dependencies
POSIX.4a	both	POSIX(ON)

Format

```
#define _OPEN_THREADS
#include <pthread.h>

int pthread_setintrtype(int type);
```

General description

Controls when a cancel request is acted on. The cancelability types can be:

PTHREAD_INTR_ASYNCHRONOUS
The thread can be canceled at any time.

PTHREAD_INTR_CONTROLLED
The thread can be canceled, but only at specific points of execution. These are:
- When waiting on a condition variable, which is pthread_cond_wait() or pthread_cond_timedwait()
- When waiting for the end of another thread, which is pthread_join()
- While waiting for an asynchronous signal, which is sigwait()
- When setting the calling thread's cancelability state, which is pthread_setintr()
- Testing specifically for a cancel request, which is pthread_testintr()
- When suspended because of POSIX functions or one of the following C standard functions: close(), fcntl(), open(), pause(), read(), tcdrain(), tcsetattr(), sigsuspend(), sigwait(), sleep(), wait(), or write()

Usage notes

1. If you are writing to the Single UNIX Specification, Version 3 standard, use pthread_setcanceltype() in place of pthread_setintrtype().

Returned value

If successful, pthread_setintrtype() returns the previous type.

If unsuccessful, pthread_setintrtype() returns -1 and sets errno to one of the following values:

Error Code
Description

EINVAL
type is an invalid value.

Example

CELEBP50

```
/* CELEBP50 */
#define _OPEN_THREADS
#include <stdio.h>
```

```
#include <string.h>
#include <pthread.h>
#include <errno.h>
#include <unistd.h>

int  thstatus;
char state[60] = "enable/controlled - initial default";

void * thfunc(void *voidptr)
{
 int  rc;
 char *parmptr;

 parmptr = voidptr;
 printf("parm = %s.\n", parmptr);
 if ( pthread_setintrtype(PTHREAD_INTR_CONTROLLED ) == -1 ) {
   printf("set controlled failed. %s\n", strerror(errno));
   thstatus = 103;
   pthread_exit(&thstatus);
 }
 strcpy(state, "disable/controlled");

 if ( pthread_setintr(PTHREAD_INTR_ENABLE) == -1 ) {
   printf("set enable failed. %s\n", strerror(errno));
   thstatus = 104;
   pthread_exit(&thstatus);
 }
 strcpy(state, "enable/controlled");

 strcat(state, " - pthread_testintr");

 while(1) {
   pthread_testintr();
   sleep(1);
 }

 thstatus = 100;
 pthread_exit(&thstatus);
}

main(int argc, char *argv[]) {
  int            rc;
  pthread_attr_t attrarea;
  pthread_t      thid;
  char           parm[] = "abcdefghijklmnopqrstuvwxyz";
  int            *statptr;

  if ( pthread_attr_init(&attrarea) == -1 ) {
    printf("pthread_attr_init failed. %s\n", strerror(errno));
    exit(1);
  }

  if ( pthread_create(&thid, &attrarea, thfunc, (void *)&parm) == -1) {
    printf("pthread_create failed. %s\n", strerror(errno));
    exit(2);
  }

  sleep(5);

  if ( pthread_cancel(thid) == -1 ) {
    printf("pthread_cancel failed. %s\n", strerror(errno));
    exit(3);
  }

   if ( pthread_join(thid, (void **)&statptr)== -1 ) {
    printf("pthread_join failed. %s\n", strerror(errno));
    exit(4);
  }

  if ( statptr == (int *)-1 )
    printf("thread was cancelled. state = %s.\n", state);
  else
    printf("thread was not cancelled. thstatus = %d.\n", *statptr);

  exit(0);

}
```

Output:

```
parm = abcdefghijklmnopqrstuvwxyz.
thread was canceled. state = enable/controlled - pthread_testintr.
```

Related information

- “pthread.h” on page 58
- “pthread_cancel() — Cancel a thread” on page 1234
- “pthread_setintr() — Set a thread cancelability state” on page 1330
- “pthread_testintr() — Establish a cancelability point” on page 1342

pthread_set_limit_np() — Set task and thread limits

Standards

Standards / Extensions	C or C++	Dependencies
z/OS UNIX	both	POSIX(ON)

Format

```
#define  _OPEN_SYS
#include <pthread.h>

int pthread_set_limit_np(int action, int maxthreadtasks, int maxthreads);
```

General description

The pthread_set_limit_np() function allows you to control how many tasks and threads can be created for a process. On a single call, you can specify that you want to update either the maximum number of tasks, the maximum number of threads, or both. The maximum number of tasks and threads is dependent upon the size of the private area below 16M. A realistic limit is 200 to 400 tasks and threads.

The *action* can be set to one of the following symbolics, as defined in the pthread.h header file:

__STL_MAX_TASKS
Specify this action when only updating the maximum number of tasks.

__STL_MAX_THREADS
Specify this action when only updating the maximum number of threads.

__STL_SET_BOTH
Specify this action when updating both the maximum number of tasks and the maximum number of threads at the same time.

For more information on the allowable values for maxthreadtasks and maxthreads, see the BPX1STL function in *z/OS UNIX System Services Programming: Assembler Callable Services Reference, SA23-2281*.

Returned value

If successful, pthread_set_limit_np() returns 0.

If unsuccessful, pthread_set_limit_np() returns -1.

For more information regarding return values and reason codes, see the BPX1STL function in *z/OS UNIX System Services Programming: Assembler Callable Services Reference, SA23-2281*.

Related information

- "pthread.h" on page 58
- "pthread_attr_setsynctype_np() — Set thread sync type" on page 1231
- "pthread_attr_setweight_np() — Set weight of thread attribute object" on page 1232

pthread_setspecific() — Set the thread-specific value for a key

Standards

Standards / Extensions	C or C++	Dependencies
POSIX.4a Single UNIX Specification, Version 3	both	POSIX(ON)

Format

```
#define _OPEN_THREADS
#include <pthread.h>

int pthread_setspecific(pthread_key_t key, void *value);
```

SUSV3:

```
#define _UNIX03_THREADS
#include <pthread.h>
int pthread_setspecific(pthread_key_t key, const void *value);
```

General description

Associates a thread-specific value, *value*, with a key identifier, *key*.

Many multithreaded applications require storage shared among threads but a unique value for each thread. A thread-specific data key is an identifier, created by a thread, for which each thread in the process can set a unique key *value*.

pthread_key_t is a storage area where the system places the key identifier. To create a key, a thread uses pthread_key_create(). This returns the key identifier into the storage area of type *pthread_key_t*. At this point, each of the threads in the application has the use of that key, and can set its own unique value by use of pthread_setspecific(). A thread can get its own unique value using pthread_getspecific().

Returned value

If successful, pthread_setspecific() returns 0.

If unsuccessful, pthread_setspecific() returns -1 and sets errno to one of the following values:

Error Code
Description

EINVAL
The key identifier *key* is not valid.

ENOMEM
Insufficient memory exists to associate the non-NULL value with the key.

Special behavior for Single UNIX Specification, Version 3: If unsuccessful, pthread_setspecific() returns an error number to indicate the error.

Example

CELEBP51

```
/* CELEBP51 */
#define _OPEN_THREADS

#include <stdio.h>
#include <stdlib.h>
#include <errno.h>
#include <pthread.h>

#define threads 3
#define BUFFSZ  48
pthread_key_t   key;

void            *threadfunc(void *parm)
{
 int        status;
 void      *value;
 int        threadnum;
 int       *tnum;
 void      *getvalue;
 char       Buffer[BUFFSZ];

 tnum = parm;
 threadnum = *tnum;

 printf("Thread %d executing\n", threadnum);

 if (!(value = malloc(sizeof(Buffer))))
     printf("Thread %d could not allocate storage, errno = %d\n",
                                              threadnum, errno);
 status = pthread_setspecific(key, (void *) value);
 if ( status <  0) {
    printf("pthread_setspecific failed, thread %d, errno %d",
                                              threadnum, errno);
    pthread_exit((void *)12);
 }
 printf("Thread %d setspecific value: %d\n", threadnum, value);

 getvalue = 0;
 status = pthread_getspecific(key, &getvalue);
 if ( status <  0) {
    printf("pthread_getspecific failed, thread %d, errno %d",
                                              threadnum, errno);
    pthread_exit((void *)13);
 }

 if (getvalue != value) {
   printf("getvalue not valid, getvalue=%d", (int)getvalue);
   pthread_exit((void *)68);
 }

 pthread_exit((void *)0);
}

void  destr_fn(void *parm)
{
   printf("Destructor function invoked\n");
   free(parm);
}

main() {
 int          getvalue;
 int          status;
 int          i;
 int          threadparm[threads];
 pthread_t    threadid[threads];
 int          thread_stat[threads];


 if ((status = pthread_key_create(&key, destr_fn )) < 0) {
    printf("pthread_key_create failed, errno=%d", errno);
    exit(1);
 }

 for (i=0; i<threads; i++) {
    threadparm[i] = i+1;
    status = pthread_create( &threadid[i],
                             NULL,
                             threadfunc,
                             (void *)&threadparm[i]);
    if ( status <  0) {
       printf("pthread_create failed, errno=%d", errno);
       exit(2);
    }
  }

 for ( i=0; i<threads; i++) {
    status = pthread_join( threadid[i],
    (void *)&thread_stat[i]);
    if ( status <  0) {
       printf("pthread_join failed, thread %d, errno=%d\n", i+1, errno);
    }

    if (thread_stat[i] != 0)   {
        printf("bad thread status, thread %d, status=%d\n", i+1,
                                            thread_stat[i]);
      }
  }
 exit(0);
}
```

Related information

- "pthread.h" on page 58
- "pthread_getspecific() — Get the thread-specific value for a key" on page 1269
- "pthread_getspecific_d8_np() — Get the thread-specific value for a key" on page 1273
- "pthread_key_create() — Create thread-specific data key" on page 1279

pthread_sigmask() — Examine or change a thread blocked signals format

Standards

Standards / Extensions	C or C++	Dependencies
Single UNIX Specification, version 3	both	z/OS V1R7 POSIX(ON)

Format

```
#define _OPEN_THREADS 2
#include <signal.h>

int pthread_sigmask(int option, const sigset_t *__restrict__ new_set,
                    sigset_t *__restrict__ old_set);
```

General description

pthread_sigmask() examines, changes, or examines and changes the signal mask of the calling thread. If there is only one thread, it does the same for the calling process.

Typically, pthread_sigmask(SIG_BLOCK, ..., ...) is used to block signals during a critical section of code. At the end of the critical section of code, pthread_sigmask(SIG_SETMASK, ..., ...) is used to restore the mask to the previous value returned by pthread_sigmask(SIG_BLOCK, ..., ...).

option indicates the way in which the existing set of blocked signals should be changed. The following are the possible values for *option*, defined in the signal.h header file:

- SIG_BLOCK – Indicates that the set of signals given by *new_set* should be blocked, in addition to the set currently being blocked.
- SIG_UNBLOCK – Indicates that the set of signals given by *new_set* should not be blocked. These signals are removed from the current set of signals being blocked.
- SIG_SETMASK – Indicates that the set of signals given by *new_set* should replace the old set of signals being blocked.

new_set points to a signal set giving the new signals that should be blocked or unblocked (depending on the value of *option*) or it points to the new signal mask if the option was SIG_SETMASK. Signal sets are described in "sigemptyset() — Initialize a Signal Mask to Exclude All Signals" in topic 3.727. If *new_set* is a NULL pointer, the set of blocked signals is not changed. pthread_sigmask() determines the current set and returns this information in **old_set*. If *new_set* is NULL, the value of option is not significant. The signal set manipulation functions: sigemptyset(), sigfillset(), sigaddset(), and sigdelset() must be used to establish the new signal set pointed to by *new_set*.

old_set points to a memory location where pthread_sigmask() can store a signal set. If *new_set* is NULL, *old_set* returns the current set of signals being blocked. When *new_set* is not NULL, the set of signals pointed to by *old_set* is the previous set.

If there are any pending unblocked signals, either at the process level or at the current thread's level after pthread_sigmask() has changed the signal mask, then at least one of those signals is delivered to the thread before pthread_sigmask() returns.

The signals SIGKILL, SIGSTOP, or SIGTRACE cannot be blocked. If you attempt to use pthread_sigmask() to block these signals, the attempt is ignored. pthread_sigmask() does not return an error status.

SIGFPE, SIGILL, and SIGSEGV signals that are not artificially generated by kill(), killpg(), raise(), or pthread_kill() (that is, were generated by the system as a result of a hardware or software exception) will not be blocked.

If an artificially raised SIGFPE, SIGILL, or SIGSEGV signal is pending and blocked when an exception causes another SIGFPE, SIGILL, or SIGSEGV signal, both the artificial and exception-caused signals may be delivered to the application.

If pthread_sigmask() fails, the signal mask of the thread is not changed.

Usage Notes

1. The use of the SIGTHSTOP and SIGTHCONT signal is not supported with this function.

Returned value

If successful, pthread_sigmask() returns 0. Otherwise, pthread_sigmask() returns one of the following error numbers:

EINVAL
option does not have one of the recognized values.

Related information

- "pthread.h" on page 58
- "signal.h" on page 63
- "kill() — Send a signal to a process" on page 931
- "killpg() — Send a signal to a process group" on page 934
- "pthread_kill() — Send a signal to a thread" on page 1283
- "raise() — Raise signal" on page 1371
- "sigaction() — Examine or change a signal action" on page 1611
- "sigaddset() — Add a signal to the signal mask" on page 1628
- "sigdelset() — Delete a signal from the signal mask" on page 1632
- "sigemptyset() — Initialize a signal mask to exclude all signals" on page 1633
- "sigfillset() — Initialize a signal mask to include all signals" on page 1635
- "sighold() — Add a signal to a thread" on page 1636
- "sigismember() — Test if a signal is in a signal mask" on page 1638
- "signal() — Handle interrupts" on page 1643
- "sigpause() — Unblock a signal and wait for a signal" on page 1649
- "sigpending() — Examine pending signals" on page 1650
- "sigprocmask() — Examine or change a thread" on page 1651
- "sigrelse() — Remove a signal from a thread" on page 1656
- "sigset() — Change a signal action or a thread" on page 1656

- “sigsuspend() — Change mask and suspend the thread” on page 1663

pthread_tag_np() — Set and query thread tag data

Standards

Standards / Extensions	C or C++	Dependencies
z/OS UNIX	both	

Format

```
#define _OPEN_THREADS
#include <pthread.h>

int pthread_tag_np(const char *newtag, char *oldtag);
```

General description

The pthread_tag_np() function is used to set and query the contents of the calling thread's tag data.

The parameters supported are:

newtag
Specifies the new tag data to be set for the callers thread. The length of the new tag data must be in the range of 0-65 bytes. If the length is zero (NULL string) the caller's thread tag data will be cleared.

oldtag Specifies the string where pthread_tag_np() returns the old (current) tag data for the caller's thread. Tag data can be up to 66 bytes (including the trailing NULL).

Returned value

If successful, pthread_tag_np() returns 0.

If unsuccessful, pthread_tag_np() returns -1 and sets errno to one of the following values:

Error Code
Description

EFAULT
One of the following errors was detected:
- All or part of the newtag string is not addressable by the caller.
- All or part of the oldtag string is not addressable by the caller.

EINVAL
The length of the newtag string is not within allowable range (0 to 65 bytes).

EMVSERR
An MVS environmental or internal error has occurred.

Related information

- “pthread.h” on page 58
- “pthread_create() — Create a thread” on page 1261

pthread_testcancel() — Establish a cancelation point

Standards

Standards / Extensions	C or C++	Dependencies
Single UNIX Specification, version 3	both	z/OS V1R7 POSIX(ON)

Format

```
#define _OPEN_THREADS 2
#include <pthread.h>

void pthread_testcancel(void);
```

General description

pthread_testcancel() allows the thread to solicit cancel requests at specific points within the current thread. You must have the cancelability state set to enabled (PTHREAD_CANCEL_ENABLE) for this function to have any effect.

Returned value

pthread_testcancel() returns no values.

Related information

- "Thread Cancelation" in the *z/OS XL C/C++ Programming Guide*
- “pthread.h” on page 58
- “pthread_cancel() — Cancel a thread” on page 1234
- “pthread_setcancelstate() — Set a thread cancelability state format” on page 1328
- “pthread_setcanceltype() — Set a thread cancelability type format” on page 1329

pthread_testintr() — Establish a cancelability point

Standards

Standards / Extensions	C or C++	Dependencies
POSIX.4a	both	POSIX(ON)

Format

```
#define _OPEN_THREADS
#include <pthread.h>

void pthread_testintr(void);
```

General description

Allows the thread to solicit cancel requests at specific points within the current thread. You must have the cancelability state set to enabled (PTHREAD_INTR_ENABLE) for this function to have any effect.

Usage notes

1. If you are writing to the Single UNIX Specification, Version 3 standard, use pthread_testcancel() in place of pthread_testintr().

Returned value

pthread_testintr() returns no values.

There are no documented errno values. Use perror() or strerror() to determine the cause of the error.

Example

CELEBP52

```
/* CELEBP52 */
#define _OPEN_THREADS
#include <stdio.h>
#include <string.h>
#include <pthread.h>
#include <errno.h>
#include <unistd.h>

int  thstatus;
char state[60] = "enable/controlled - initial default";

void * thfunc(void *voidptr)
{
 int  rc;
 char *parmptr;

 parmptr = voidptr;
 printf("parm = %s.\n", parmptr);
 if ( pthread_setintrtype(PTHREAD_INTR_CONTROLLED ) == -1 ) {
   printf("set controlled failed. %s\n", strerror(errno));
   thstatus = 103;
   pthread_exit(&thstatus);
 }
 strcpy(state, "disable/controlled");

 if ( pthread_setintr(PTHREAD_INTR_ENABLE) == -1 ) {
   printf("set enable failed. %s\n", strerror(errno));
   thstatus = 104;
   pthread_exit(&thstatus);
 }
 strcpy(state, "enable/controlled");

 strcat(state, " - pthread_testintr");

 while(1) {
   pthread_testintr();
   sleep(1);
 }

 thstatus = 100;
 pthread_exit(&thstatus);
}

main(int argc, char *argv[]) {
  int            rc;
  pthread_attr_t attrarea;
  pthread_t      thid;
  char           parm[] = "abcdefghijklmnopqrstuvwxyz";
  int            *statptr;

  if ( pthread_attr_init(&attrarea) == -1 ) {
    printf("pthread_attr_init failed. %s\n", strerror(errno));
    exit(1);
  }
  if ( pthread_create(&thid, &attrarea, thfunc, (void *)&parm) == -1) {
    printf("pthread_create failed. %s\n", strerror(errno));
    exit(2);
  }

  sleep(5);

  if ( pthread_cancel(thid) == -1 ) {
    printf("pthread_cancel failed. %s\n", strerror(errno));
    exit(3);
  }

   if ( pthread_join(thid, (void **)&statptr)== -1 ) {
    printf("pthread_join failed. %s\n", strerror(errno));
    exit(4);
  }

  if ( statptr == (int *)-1 )
    printf("thread was cancelled. state = %s.\n", state);
  else
    printf("thread was not cancelled. thstatus = %d.\n", *statptr);

  exit(0);
}
```

Output:

```
parm = abcdefghijklmnopqrstuvwxyz.
thread was canceled. state = enable/controlled - pthread_testintr.
```

Related information

- "pthread.h" on page 58
- "pthread_cancel() — Cancel a thread" on page 1234
- "pthread_setintr() — Set a thread cancelability state" on page 1330
- "pthread_setintrtype() — Set a thread cancelability type" on page 1333

pthread_yield() — Release the processor to other threads

Standards

Standards / Extensions	C or C++	Dependencies
POSIX.4a	both	POSIX(ON)

Format

```
#define _OPEN_THREADS
#include <pthread.h>

void pthread_yield(NULL);
```

General description

The pthread_yield() function allows a thread to give up control of a processor so that another thread can have the opportunity to run.

The parameter to the function must be NULL, because non-NULL values are reserved.

The speed at which the pthread_yield() function releases a processor can be configured by using the _EDC_PTHREAD_YIELD and _EDC_PTHREAD_YIELD_MAX environment variables. The

_EDC_PTHREAD_YIELD environment variable is used to configure the pthread_yield() function to release the processor immediately, or to release the processor after a delay. The _EDC_PTHREAD_YIELD_MAX environment variable is used to change the maximum delay to a value less than the default (32 milliseconds).

For more information about the _EDC_PTHREAD_YIELD and _EDC_PTHREAD_YIELD_MAX environment variables, see “Using Environment Variables” in *z/OS XL C/C++ Programming Guide*.

Returned value

pthread_yield() returns no values.

There are no documented errno values.

Example

CELEBP53

```
/* CELEBP53 */
#define _OPEN_THREADS
#include <pthread.h>
#include <stdio.h>
#include <unistd.h>

void *thread(void *arg) {

  /* A simple loop with only puts() would allow a thread to write several
  lines in a row.
  With pthread_yield(), each thread gives another thread a chance before
  it writes its next line */

  while (1) {
    puts((char*) arg);
    pthread_yield(NULL);
  }
}

main() {
  pthread_t t1, t2, t3;

  if (pthread_create(&t1, NULL, thread, "thread 1") != 0) {
    perror("pthread_create() error");
    exit(1);
  }

  if (pthread_create(&t2, NULL, thread, "thread 2") != 0) {
    perror("pthread_create() error");
    exit(2);
  }

  if (pthread_create(&t3, NULL, thread, "thread 3") != 0) {
    perror("pthread_create() error");
    exit(3);
  }

  sleep(1);

  exit(0); /* this will tear all threads down */
}
```

Output:

```
thread 1
thread 3
thread 2
thread 1
thread 3
thread 2
thread 1
thread 3
thread 2
thread 1
thread 3
```

Related information

- “pthread.h” on page 58
- “sched_yield() — Release the processor to other threads” on page 1467

ptsname() — Get name of the slave pseudoterminal device

Standards

Standards / Extensions	C or C++	Dependencies
XPG4.2 Single UNIX Specification, Version 3	both	

Format

```
#include <stdlib.h>

char *ptsname(int fildes);
```

General description

The ptsname() function returns the name of the slave pseudoterminal device associated with a master pseudoterminal device. The *fildes* argument is a file descriptor that refers to the master device. ptsname() returns a pointer to a string containing the path name of the corresponding slave device.

Returned value

If successful, ptsname() returns a pointer to a string which is the name of the pseudoterminal slave device.

If unsuccessful, ptsname() returns a NULL pointer. This could occur if *fildes* is an invalid file descriptor or if the slave device name does not exist in the file system.

No errors are defined.

Related information

- “stdlib.h” on page 70
- “grantpt() — Grant access to the slave pseudoterminal device” on page 814
- “open() — Open a file” on page 1152
- “ttyname() — Get the name of a terminal” on page 1931
- “unlockpt() — Unlock a pseudoterminal master and slave pair” on page 1964

putc(), putchar() — Write a character

Standards

Standards / Extensions	C or C++	Dependencies
ISO C POSIX.1 XPG4 XPG4.2 C99 Single UNIX Specification, Version 3	both	

Format

```
#include <stdio.h>

int putc(int c, FILE *stream);
int putchar(int c);
```

General description

Converts *c* to `unsigned char` and then writes *c* to the output *stream* at the current position. The putchar() function is identical to:

```
putc(c, stdout);
```

These functions are also available as macros in the z/OS XL C/C++ product. For performance purposes, it is recommended that the macro forms rather than the functional forms be used.

By default, if the stdio.h header file is included, the macro is invoked. Therefore, the stream argument expression should never be an expression with side effects.

The actual function can be accessed using one of the following methods:
- For C only: do *not* include stdio.h.
- Specify `#undef`, for example, `#undef putc`.
- Surround the function name by parentheses, for example: `(putchar)('a')`.

In a multithread application, in the presence of the feature test macro, _OPEN_THREADS, these macros are in an #undef status because they are not thread-safe.

putc() and putchar() are not supported for files opened with `type=record` or `type=blocked`.

putc() and putchar() have the same restriction as any write operation for a read immediately following a write or a write immediately following a read. Between a write and a subsequent read, there must be an intervening flush or reposition. Between a read and a subsequent write, there must also be an intervening flush or reposition unless an EOF has been reached.

If the application is not multithreaded, then setting the _ALL_SOURCE_NO_THREADS feature test macro may improve performance of the application, because it allows use of the inline version of this function.

Returned value

If successful, putc() and putchar() return the character written.

If unsuccessful, putc() and putchar() return EOF.

Example

CELEBP54

```
/* CELEBP54

   This example writes the contents of a buffer to a data
   stream.
   The body of the "for" statement is null because the
   example carries out the writing operation in the test
   expression.

 */
#include <stdio.h>
#include <string.h>
#define  LENGTH 80

int main(void)
{
   FILE *stream = stdout;
   int i, ch;
   char buffer[LENGTH + 1] = "Hello world\n";

   /* This could be replaced by using the fwrite routine */
   for ( i = 0;
        (i < strlen(buffer)) && ((ch = putc(buffer[i], stream)) != EOF);
         ++i);
}
```

Output:

```
Hello world
```

Related information

- "stdio.h" on page 68
- "getc(), getchar() — Read a character" on page 688
- "fputc() — Write a character" on page 603
- "fwrite() — Write items" on page 676

putenv() — Change or add an environment variable

Standards

Standards / Extensions	C or C++	Dependencies
XPG4 XPG4.2 Single UNIX Specification, Version 3	both	

Format

```
#define _XOPEN_SOURCE
#include <stdlib.h>

int putenv(char *envvar);
```

General description

Adds a new environment variable or changes the value of an existing one. The argument *envvar* is a pointer to a NULL-terminated character string that should be of the form:

name=value

Where:

1. The first part, *name*, is a character string that represents the name of the environment variable. It is this part of the environment variable that putenv() will use when it searches the array of environment variable to determine whether to add or change this environment variable.
2. The second part, =, is a separator character (since the equal sign is used as a separator character it cannot appear in the *name*).
3. The third part, *value*, is a NULL-terminated character string that represents the value that the environment variable, *name*, will be set to.

putenv() is a simplified form of setenv() and is equivalent to

setenv(*name*, *value*, 1)

Note: Starting with, z/OS V1R2, the storage used to define the environment variable pointed to by *envvar* is added to the array of environment variables. Previously, the system copied the string into system allocated storage. A new environment variable, _EDC_PUTENV_COPY, will allow the previous behavior to continue if set to YES. If _EDC_PUTENV_COPY is not set or is set to any other value the new behavior will take place.

Special behavior for POSIX C: You can use the external variable ***environ* (defined as `extern char **environ`) to access the array of pointers to environment variables.

Returned value

If successful, putenv() returns 0.

If unsuccessful, putenv() returns -1 and sets errno to one of the following values:

Error Code
Description

ENOMEM
Insufficient memory was available.

Special behavior for z/OS UNIX Services:

EINVAL
The environment variable pointed to by the argument *envvar* does not follow the prescribed format. The equal sign (=) separating the environment variable name from the value was not found.

Related information

- See the topic about C/370 environmental variables in *z/OS XL C/C++ Programming Guide*
- “stdlib.h” on page 70
- “clearenv() — Clear environment variables” on page 281
- “getenv() — Get value of environment variables” on page 704

- "__getenv() — Get an environment variable" on page 705
- "setenv() — Add, delete, and change environment variables" on page 1528

putmsg(), putpmsg() — Send a message on a STREAM

Standards

Standards / Extensions	C or C++	Dependencies
XPG4.2 Single UNIX Specification, Version 3	both	

Format

```
#define _XOPEN_SOURCE_EXTENDED 1
#include <stropts.h>

int putmsg(int fildes, const struct strbuf *ctlptr,
      const struct strbuf *dataptr, int flags);

int putpmsg(int fildes, const struct strbuf *ctlptr,
       const struct strbuf *dataptr, int band, int flags);
```

General description

The putmsg() function creates a message from a process buffer(s) and sends the message to a STREAMS file. The message may contain either a data part, a control part, or both. The data and control parts are distinguished by placement in separate buffers, as described below. The semantics of each part is defined by the STREAMS module that receives the message.

The putpmsg() function does the same thing as putmsg(), but the process can send messages in different priority bands. Except where noted, all requirements on putmsg() also pertain to putpmsg().

The *fildes* argument specifies a file descriptor referencing an open STREAM. The *ctlptr* and *dataptr* arguments each point to a **strbuf** structure.

The *ctlptr* argument points to the structure describing the control part, if any, to be included in the message. The **buf** member in the **strbuf** structure points to the buffer where the control information resides, and the **len** member indicates the number of bytes to be sent. The **maxlen** member is not used by putmsg(). In a similar manner, the argument *dataptr* specifies the data, if any, to be included in the message. The *flags* argument indicates what type of message should be sent and is described further below.

To send the data part of a message, *dataptr* must not be a NULL pointer and the **len** member of *dataptr* must be 0 or greater. To send the control part of a message, the corresponding values must be set for *ctlptr*. No data (control) part will be sent if either *dataptr* (*ctlptr*) is a NULL pointer or the **len** member of *dataptr* (*ctlptr*) is set to -1.

For putmsg(), if a control part is specified and *flags* is set to `RS_HIPRI`, a high priority message is sent. If no control part is specified, and *flags* is set to `RS_HIPRI`, putmsg() fails and sets errno to EINVAL. If *flags* is set to 0, a normal message (priority band equal to 0) is sent. If a control part and data part are not specified and *flags* is set to 0, no message is sent and 0 is returned.

The STREAM head guarantees that the control part of a message generated by putmsg() is at least 64 bytes in length.

For putpmsg(), the flags are different. The *flags* argument is a bitmask with the following mutually-exclusive flags defined: `MSG_HIPRI` and `MSG_BAND`. If *flags* is set to 0, putpmsg() fails and sets errno to EINVAL. If a control part is specified and *flags* is set to `MSG_HIPRI` and *band* is set to 0, a high-priority message is sent. If *flags* is set to `MSG_HIPRI` and either no control part is specified or *band* is set to a nonzero value, putpmsg() fails and sets errno to EINVAL. If *flags* is set to `MSG_BAND`, then a message is sent in the priority band specified by *band*. If a control part and data part are not specified and *flags* is set to `MSG_BAND`, no message is sent and 0 is returned.

The putmsg() function blocks if the STREAM write queue is full due to internal flow control conditions, with the following exceptions:

- For high-priority messages, putmsg() does not block on this condition and continues processing the message.
- For other messages, putmsg() does not block but fails when the write queue is full and `O_NONBLOCK` is set.

The putmsg() function also blocks, unless prevented by lack of internal resources, while waiting for the availability of message blocks in the STREAM, regardless of priority or whether `O_NONBLOCK` has been specified. No partial message is sent.

The following symbolic constants are defined under _XOPEN_SOURCE_EXTENDED 1 in <stropts.h>.

MSG_ANY
Receive any message.

MSG_BAND
Receive message from specified band.

MSG_HIPRI
Send/Receive high priority message.

MORECTL
More control information is left in message.

MOREDATA
More data is left in message.

Returned value

If successful, putmsg() and putpmsg() return 0.

If unsuccessful, putmsg() and putpmsg() return -1 and set errno to one of the following values.

Note: z/OS UNIX services do not supply any STREAMS devices or pseudodevices. It is impossible for putmsg() and putpmsg() to send a message on a STREAM. It will always return -1 with errno set to indicate the failure. See “open() — Open a file” on page 1152

Error Code
Description

EAGAIN
A non-priority message was specified, the `O_NONBLOCK` flag is set, and the

STREAM write queue is full due to internal flow control conditions; or buffers could not be allocated for the message that was to be created.

EBADF
fildes is not a valid file descriptor open for writing.

EINTR
A signal was caught during putmsg().

EINVAL
An undefined value is specified in *flags*, or *flags* is set to `RS_HIPRI` or `MSG_HIPRI` and no control part is supplied, or the STREAM or multiplexer referenced by *fildes* is linked (directly or indirectly) downstream from a multiplexer, or *flags* is set to `MSG_HIPRI` and *band* is nonzero (for putpmsg() only).

ENOSR
Buffers could not be allocated for the message that was to be created due to insufficient STREAMS memory resources.

ENOSTR
A STREAM is not associated with *fildes*.

ENXIO
A hang-up condition was generated downstream for the specified STREAM.

EPIPE or EIO
The *fildes* argument refers to a STREAMS-based pipe and the other end of the pipe is closed. A SIGPIPE signal is generated for the calling process.

ERANGE
The size of the data part of the message does not fall within the range specified by the maximum and minimum packet sizes of the topmost STREAM module. This value is also returned if the control part of the message is larger than the maximum configured size of the control part of a message, or if the data part of a message is larger than the maximum configured size of the data part of a message.

In addition, putmsg() and putpmsg() will fail if the STREAM head had processed an asynchronous error before the call. In this case, the value of errno does not reflect the result of putmsg() or putpmsg() but reflects the prior error.

Related information

- “stropts.h” on page 72
- “getmsg(), getpmsg() — Receive next message from a STREAMS file” on page 735
- “poll() — Monitor activity on file descriptors and message queues” on page 1185
- “read() — Read from a file or socket” on page 1376
- “readv() — Read data on a file or socket and store in a set of buffers” on page 1389
- “write() — Write data on a file or socket” on page 2085
- “writev() — Write data on a file or socket from an array” on page 2092

puts() — Write a string

Standards

Standards / Extensions	C or C++	Dependencies
XPG4 XPG4.2 Single UNIX Specification, Version 3 Language Environment	both	

Format

```
#include <stdio.h>

int puts(const char *string);

#define _OPEN_SYS_UNLOCKED_EXT 1
#include <stdio.h>

int puts_unlocked(const char *string);
```

General description

Writes the string pointed to by *string* to the stream pointed to by `stdout`, and appends the newline character to the output. The terminating NULL character is not written.

If `stdout` points to the text stream, and the output string is longer than the length of the stream's record, the output is *wrapped*. That is, the record is filled with the output characters, the last character of the record is set to a newline character, and the remaining output characters are written to the next record. Such wrapping is repeated until the remaining output characters fit into the record. Please note that the newline character is appended to the last portion of the output string. If the output string is shorter than the record, the remaining characters of the record are filled with blanks—if `stdout` is opened in a text mode—or with NULL characters if the `stdout` is opened in binary mode.

The puts() function is not supported for files opened with `type=record` or `type=blocked`.

puts() has the same restriction as any write operation for a read immediately following a write or a write immediately following a read. Between a write and a subsequent read, there must be an intervening flush or reposition. Between a read and a subsequent write, there must also be an intervening flush or reposition unless an EOF has been reached.

puts_unlocked() is functionally equivalent to puts() with the exception that it is not thread-safe. This function can safely be used in a multithreaded application if and only if it is called while the invoking thread owns the (FILE*) object, as is the case after a successful call to either the flockfile() or ftrylockfile() function.

Returned value

If successful, puts() returns the number of bytes written. However, newline characters used to wrap the data are not counted.

If unsuccessful, puts() returns EOF.

If a system write-error occurs, the write stops at the point of failure.

After truncation, puts() does not count the truncated characters, but returns the actual number of bytes written.

Example

CELEBP55

```
/* cCELEBP55

   This example writes "Hello World" to stdout.

 */
#include <stdio.h>

int main(void)
{
  if ( puts("Hello World") == EOF )
    printf( "Error in puts\n" );
}
```

Output:

```
Hello World
```

Related information

- “stdio.h” on page 68
- “fputs() — Write a string” on page 605
- “gets() — Read a string” on page 769

pututxline() — Write entry to utmpx database

Standards

Standards / Extensions	C or C++	Dependencies
XPG4.2 Single UNIX Specification, Version 3	both	

Format

```
#define _XOPEN_SOURCE_EXTENDED 1
#include <utmpx.h>

struct utmpx *pututxline(const struct utmpx *utmpx);
```

General description

The pututxline() function writes out the structure into the utmpx database, when the calling process has appropriate privileges. The pututxline() function uses getutxid() to search for a record that satisfies the request. If the getutxid() search succeeds, then the entry is replaced. Otherwise, a new entry is made at the end of the database. If the utmpx database does not already exist, then pututxline() creates the utmpx database with file permissions 0644. (See the __utmpxname() function for information on the utmpx structure.)

If the ut_type field in the entry being added is EMPTY, it is always placed at the start of the utmpx database. For this reason, pututxline() should not be used to place EMPTY entries in the utmpx database.

The pututxline() function obtains an exclusive lock in the utmpx database on the byte range of the record which is ready to write and releases the lock before returning to its caller. The functions getutxent(), getutxid(), and getutxline() might continue to read and are not affected by pututxline().

Because the pututxline() function processes thread-specific data the pututxline() function can be used safely from a multithreaded application. If multiple threads in the same process open the database, then each thread opens the database with a different file descriptor. The thread's database file descriptor is closed when the calling thread terminates or the endutxent() function is called by the calling thread.

The name of the database file defaults to /etc/utmpx. To process a different database file name use the __utmpxname() function.

pututxline() is not supported when all of the following conditions are true:

- The security environment for the current address space has the trusted attribute.
- Either the effective UID is different than the real UID, or the effective GID is different than the real GID.
- The effective UID is not 0.
- The utmpx file is not writable by normal (non-trusted) processes with the current effective UID and GID.
- pututxline() is called after getutxline(), getutxid(), or getutxent(), with no intervening calls to endutxent() or __utmpxname().

For all entries that match a request, the ut_type member indicates the type of the entry. Other members of the entry will contain meaningful data based on the value of the `ut_type` member as follows:

EMPTY
No other members have meaningful data.

BOOT_TIME
ut_tv is meaningful.

__RUN_LVL
ut_tv and ut_line are meaningful

OLD_TIME
ut_tv is meaningful.

NEW_TIME
ut_tv is meaningful.

USER_PROCESS
ut_id, ut_user (login name of the user), ut_line, ut_pid, and ut_tv are meaningful.

INIT_PROCESS
ut_id, ut_pid, and ut_tv are meaningful.

LOGIN_PROCESS
ut_id, ut_user (implementation-specific name of the login process), ut_pid, and ut_tv are meaningful.

DEAD_PROCESS
ut_id, ut_pid, and ut_tv are meaningful.

Returned value

If successful, pututxline() returns a pointer to a utmpx structure containing a copy of the entry written to the database.

If unsuccessful, pututxline() returns a NULL pointer.

pututxline() may fail if the process does not have appropriate privileges.

Related information

- "utmpx.h" on page 84
- "endutxent() — Close the utmpx database" on page 425
- "getutxent() — Read next entry in utmpx database" on page 793
- "getutxid() — Search by ID utmpx database" on page 795
- "getutxline() — Search by line utmpx database" on page 796
- "setutxent() — Reset to start of utmpx database" on page 1595
- "__utmpxname() — Change the utmpx database name" on page 1971

putw() — Put a machine word on a stream

Standards

Standards / Extensions	C or C++	Dependencies
XPG4 XPG4.2	both	

Format

```
#define _XOPEN_SOURCE
#include <stdio.h>

int putw(int w, FILE *stream);
```

General description

The putw() function writes the word *w* to the output *stream* (at the position at which the file offset, if defined, is pointing). The size of the word is the size of a type int, and varies from machine to machine. The putw() function neither assumes nor causes special alignment in the file. The *st_ctime* and *st_mtime* fields of the file will be marked for update between the successful execution of putw() and the next successful call to fflush() or fclose() on the same stream or a call to exit() or abort().

Note:

This function is kept for historical reasons. It was part of the Legacy Feature in Single UNIX Specification, Version 2, but has been withdrawn and is not supported as part of Single UNIX Specification, Version 3. New applications should use character-based output functions to replace putw() for portability.

If it is necessary to continue using this function in an application written for Single UNIX Specification, Version 3, define the feature test macro _UNIX03_WITHDRAWN before including any standard system headers. The macro exposes all interfaces and symbols removed in Single UNIX Specification, Version 3.

Returned value

If successful, putw() returns 0.

If unsuccessful, putw() returns a nonzero value, sets the error indicators for *stream*, and sets errno to indicate the error. Refer to “fread() — Read items” on page 610 for errno values.

Related information

- “stdio.h” on page 68
- “fopen() — Open a file” on page 565
- “fwrite() — Write items” on page 676
- “getw() — Get a machine word from a stream” on page 798

putwc() — Output a wide character

Standards

Standards / Extensions	C or C++	Dependencies
XPG4 XPG4.2 Single UNIX Specification, Version 3 Language Environment	both	

Format

Non-XPG4:

```
#include <stdio.h>
#include <wchar.h>

wint_t putwc(wchar_t wc, FILE *stream);

#define _OPEN_SYS_UNLOCKED_EXT 1
#include <wchar.h>

wint_t putwc_unlocked(wchar_t wc, FILE *stream);
```

XPG4:

```
#define _XOPEN_SOURCE
#include <stdio.h>
#include <wchar.h>

wint_t putwc(wint_t wc, FILE *stream);

#define _OPEN_SYS_UNLOCKED_EXT 1
#include <wchar.h>

wint_t putwc_unlocked(wchar_t wc, FILE *stream);
```

XPG4 and MSE:

```
#define _XOPEN_SOURCE
#define _MSE_PROTOS
#include <stdio.h>
#include <wchar.h>

wint_t putwc(wchar_t wc, FILE *stream);

#define _OPEN_SYS_UNLOCKED_EXT 1
```

```
#include <wchar.h>

wint_t putwc_unlocked(wchar_t wc, FILE *stream);
```

General description

The putwc() function is equivalent to the fputwc() function, except that if it is implemented as a macro, it may evaluate *stream* more than once. Therefore, the argument should never be an expression with side effects. The behavior of this wide-character function is affected by the LC_CTYPE category of the current locale. If you use a non-wide-oriented function with putwc(), undefined results can occur.

putwc() has the same restriction as any write operation for a read immediately following a write or a write immediately following a read. Between a write and a subsequent read, there must be an intervening flush or reposition. Between a read and a subsequent write, there must also be an intervening flush or reposition unless an EOF has been reached.

putwc_unlocked() is functionally equivalent to putwc() with the exception that it is not thread-safe. This function can safely be used in a multithreaded application if and only if it is called while the invoking thread owns the (FILE*) object, as is the case after a successful call to either the flockfile() or ftrylockfile() function.

Special behavior for XPG4: If you define any feature test macro specifying XPG4 behavior before the statement in your program source file to include the wchar header, then the compiler assumes that your program is using the XPG4 variety of the putwc() function, unless you also define the _MSE_PROTOS feature test macro. Please see Table 2 on page 4 for a list of XPG4 and other feature test macros.

The prototype for the XPG4 variety of the putwc() function is:

```
wint_t putwc(wint_t wc, FILE *stream);
```

The difference between this variety and the MSE variety of the putwc() function is that the first parameter has type wint_t rather than type wchar_t.

Returned value

If successful, putwc() returns the wide character written.

If a write error occurs, the error indicator for the stream is set and WEOF is returned. If an encoding error occurs when converting from a wide character to a multibyte character, the value of the macro EILSEQ is stored in errno and WEOF is returned.

Example

CELEBP56

```
/* CELEBP56 */
#include <stdio.h>
#include <stdlib.h>
#include <wchar.h>
#include <errno.h>

int main(void)
{
   FILE    *stream;
   wchar_t *wcs = L"This test string should not cause a WEOF condition";
   int     i;
   int     rc;

   if ((stream = fopen("myfile.dat", "w")) == NULL) {
      printf("Unable to open file\n");
      exit(1);
   }

   for (i=0; wcs[i] != L'\0'; i++) {
      errno = 0;
      if ((rc = putwc(wcs[i], stream)) == WEOF) {
         printf("Unable to putwc() the wide character.\n");
         printf("wcs[%d] = 0x%X\n", i, wcs[i]);
         if (errno == EILSEQ)
            printf("An invalid wide character was encountered.\n");
         exit(1);
      }
   }

   fclose(stream);
}
```

Related information

- “stdio.h” on page 68
- “wchar.h” on page 85
- “fputwc() — Output a wide-character” on page 607

putwchar() — Output a wide character to standard output

Standards

Standards / Extensions	C or C++	Dependencies
ISO C Amendment XPG4 XPG4.2 C99 Single UNIX Specification, Version 3 Language Environment	both	

Format

Non-XPG4:

```
#include <stdio.h>
#include <wchar.h>

wint_t putwchar(wchar_t wc);

#define _OPEN_SYS_UNLOCKED_EXT 1
#include <wchar.h>

wint_t putwchar_unlocked(wchar_t wc);
```

XPG4:

```
#define _XOPEN_SOURCE
#include <stdio.h>
#include <wchar.h>

wint_t putwchar(wint_t wc);
```

```
#define _OPEN_SYS_UNLOCKED_EXT 1
#include <wchar.h>

wint_t putwchar_unlocked(wchar_t wc);
```

XPG4 and MSE:

```
#define _XOPEN_SOURCE
#define _MSE_PROTOS
#include <stdio.h>
#include <wchar.h>

wint_t putwchar(wchar_t wc);

#define _OPEN_SYS_UNLOCKED_EXT 1
#include <wchar.h>

wint_t putwchar_unlocked(wchar_t wc);
```

General description

The putwchar() function is equivalent to: putwc()(*wc* `stdout`).

The behavior of this wide-character function is affected by the LC_CTYPE category of the current locale. If you use a non-wide-oriented function with putwchar(), undefined results can occur.

putwchar() has the same restriction as any write operation for a read immediately following a write or a write immediately following a read. Between a write and a subsequent read, there must be an intervening flush or reposition. Between a read and a subsequent write, there must also be an intervening flush or reposition unless an EOF has been reached.

You may not use putwc() or putwchar() with files opened as `type=record` or `type=blocked`.

putwchar_unlocked() is functionally equivalent to putwchar() with the exception that it is not thread-safe. This function can safely be used in a multithreaded application if and only if it is called while the invoking thread owns the (FILE*) object, as is the case after a successful call to either the flockfile() or ftrylockfile() function.

Special behavior for XPG4: If you define any feature test macro specifying XPG4 behavior before the statement in your program source file to include the wchar header, then the compiler assumes that your program is using the XPG4 variety of the putwchar() function, unless you also define the _MSE_PROTOS feature test macro. Please see Table 2 on page 4 for a list of XPG4 and other feature test macros.

The prototype for the XPG4 variety of the putwchar() function is:

```
wint_t putwchar(wint_t wc);
```

The difference between this variety and the MSE variety of the putwchar() function is that its parameter has type wint_t rather than type wchar_t.

Returned value

If successful, putwchar() returns the wide character written.

If a write error occurs, the error indicator for the stream is set and WEOF is returned. If an encoding error occurs when converting from a wide character to a multibyte character, the value of the macro EILSEQ is stored in errno and WEOF is returned.

Example

CELEBP57

```
/* CELEBP57 */
#include <stdio.h>
#include <stdlib.h>
#include <wchar.h>
#include <errno.h>

int main(void)
{
   wchar_t *wcs = L"This test string should not cause a WEOF condition";
   int     i;
   int     rc;

   for (i=0; wcs[i] != L'\0'; i++) {
      errno = 0;
      if ((rc = putwchar(wcs[i])) == WEOF) {
         printf("Unable to putwchar() the wide character.\n");
         printf("wcs[%d] = 0x%X\n", i, wcs[i]);
         if (errno == EILSEQ)
            printf("An invalid wide character was encountered.\n");
         exit(1);
      }
   }
}
```

Related information

- “stdio.h” on page 68
- “wchar.h” on page 85
- “fputwc() — Output a wide-character” on page 607

pwrite() — Write data on a file or socket without file pointer change

Standards

Standards / Extensions	C or C++	Dependencies
Single UNIX Specification, Version 2 Single UNIX Specification, Version 3	both	OS/390 V2R10

Format

```
#define _XOPEN_SOURCE 500
#include <unistd.h>

ssize_t pwrite(int fildes, const void *buf, size_t nbyte, off_t offset);
```

General description

The pwrite() function performs the same action as write(), except that it writes into a given position without changing the file pointer.

The first three arguments to pwrite() are the same as write() with the addition of a fourth argument *offset* for the desired position inside the file.

Returned value

If successful, pwrite() returns the number of bytes actually written to the file associated with *fildes*. This number will never be greater than *nbyte*.

If unsuccessful, pwrite() returns -1 and sets errno to one of the following values:

Error Code
Description

EAGAIN
Resources temporarily unavailable. Subsequent requests may complete normally.

EBADF
fildes is not a valid file or socket descriptor.

ECONNRESET
A connection was forcibly closed by a peer.

EDESTADDRREQ
The socket is not connection-oriented and no peer address is set.

EFAULT
Using the *buf* and *nbyte* parameters would result in an attempt to access storage outside the caller's address space.

EFBIG
An attempt was made to write a file that exceeds the system-established maximum file size or the process's file size limit supported by the implementation.

The file is a regular file, *nbyte* is greater than 0 and the starting position is greater than or equal to the offset maximum established in the open file description associated with fields.

EINTR
pwrite() was interrupted by a signal before it had written any output.

EINVAL
The request is invalid or not supported. The STREAM or multiplexer referenced by *fildes* is linked (directly or indirectly) downstream from a multiplexer.

The *offset* argument is invalid. The value is negative.

EIO
The process is in a background process group and is attempting to write to its controlling terminal, but TOSTOP (defined in the termios.h header file) is set, the process is neither ignoring nor blocking SIGTTOU signals, and the process group of the process is orphaned. An I/O error occurred.

EMSGSIZE
The message was too big to be sent as a single datagram.

ENOBUFS
Buffer space is not available to send the message.

ENOSPC
There is no available space left on the output device.

ENOTCONN
The socket is not connected.

ENXIO
A hang-up occurred on the STREAM being written to.

EPIPE pwrite() is trying to write to a pipe that is not open for reading by any other process. This error also generates a SIGPIPE signal. For a connected stream socket the connection to the peer socket has been lost.

ERANGE
The transfer request size was outside the range supported by the STREAMS file associated with *fildes*.

ESPIPE
fildes is associated with a pipe or FIFO.

EWOULDBLOCK
socket is in nonblocking mode and no data buffers are available or the SO_SNDTIMEO timeout value was reached before buffers became available.

Related information

- "unistd.h" on page 82
- "pread() — Read from a file or socket without file pointer change" on page 1198
- "write() — Write data on a file or socket" on page 2085

qsort() — Sort array

Standards

Standards / Extensions	C or C++	Dependencies
ISO C XPG4 XPG4.2 C99 Single UNIX Specification, Version 3	both	

Format

```
#include <stdlib.h>

void qsort(void *base, size_t num, size_t width,
          int(*compare)(const void *element1, const void *element2));
```

General description

The qsort() function sorts an array of *num* elements, each of *width* bytes in size, where the first element of the array is pointed to by *base*.

The *compare* pointer points to a function, which you supply, that compares two array elements and returns an integer value specifying their relationship. The qsort() function calls the comparison function one or more times during the sort, passing pointers to two array elements on each call. The comparison function must compare the elements and return one of the following values:

Value	Meaning
< 0	*element1* less than *element2*
0	*element1* equal to *element2*
> 0	*element1* greater than *element2*

The sorted array elements are stored in increasing order, as returned by the comparison function. You can sort in reverse order by reversing the “greater than” and “less than” logic in the comparison function. If two elements are equal, their order in the sorted array is unspecified. The qsort() function overwrites the contents of the array with the sorted elements.

Special behavior for C++: C++ and C linkage conventions are incompatible, and therefore the qsort() function cannot receive C++ function pointers. If you attempt to pass a C++ function pointer to the qsort() function, the compiler will flag it as an error. To use the C++ qsort() function, you must ensure that the comparison function has C linkage, by declaring it as extern "C".

Returned value

The qsort() function returns no values.

Example

CELEBQ01

```
/* CELEBQ01

   This example sorts the arguments (argv) in ascending sequence, based on
   the ASCII value of each character and string, and using the comparison
   function compare() supplied in the example.

 */
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

               /* Declaration of compare() as a function */
#ifdef __cplusplus
extern "C" int compare(const void *, const void *);
#else
int compare(const void *, const void *);  /* macro is automatically */
#endif                                    /* defined by the C++/MVS compiler  */


int main (int argc, char *argv[ ])
{
   int i;
   argv++;
   argc--;
   qsort((char *)argv, argc, sizeof(char *), compare);
   for (i = 0; i < argc; ++i)
      printf("%s\n", argv[i]);
   return 0;
}

int compare (const void *arg1, const void *arg2)
{
               /* Compare all of both strings */
   return(strcmp(*(char **)arg1, *(char **)arg2));
}
```

Output

If the program is passed the arguments:

```
Does, this, really, sort, the, arguments, correctly?
```

then expect the following output.

```
arguments
correctly?
really
sort
the
this
Does
```

Related information

- "stdlib.h" on page 70
- "bsearch() — Search arrays" on page 220

quantexpd32(), quantexpd64(), quantexpd128() - Compute the quantum exponent

Standards

Standards / Extensions	C or C++	Dependencies
C/C++ DFP	both	z/OS V1.11

Format

```
#define __STDC_WANT_DEC_FP__
#include <math.h>

int quantexpd32(_Decimal32 x);
int quantexpd64(_Decimal64 x);
int quantexpd128(_Decimal128 x);

int quantexp(_Decimal32 x);     /* C++ only */
int quantexp(_Decimal64 x);     /* C++ only */
int qunatexp(_Decimal128 x);    /* C++ only */
```

General description

The quantexp() functions compute the quantum exponent of a finite argument.

Notes:

1. These functions work in IEEE decimal floating-point format. See "IEEE decimal floating-point" on page 95IEEE Decimal Floating-Point for more information.
2. To use IEEE decimal floating-point, the hardware must have the Decimal Floating-Point Facility installed.
3. The "quantum" when referring to a finite decimal floating-point number is defined as the "magnitude of a value of one in the rightmost digit position of the significand". For example, pennies and dollars can be represented respectively as 1 * 10 with a quantum of -2 and 0, or, 1*10^-2 and 1*10^0. For more information on the term quantum, see *z/Architecture Principles of Operation*.

Returned value

The quantexp() functions return the quantum exponent of *x*.

If x is infinite or NaN, they compute INT_MIN and a domain error occurs.

Example

```
/* CELEBQ03

   This example illustrates the quantexpd128() function.

*/

#define  __STDC_WANT_DEC_FP__
#include <stdio.h>
#include <math.h>

void main(void)
{
   _Decimal128 x, y;

   x = 4.56DL;
   y = quantexpd128(x);

   printf("quantexpd128( %DDf ) = %DDf\n", x, y);
}
```

Related information

- “math.h” on page 44
- “quantized32(), quantized64(), quantized128() — Set the exponent of X to the exponent of Y”
- “samequantumd32(), samequantumd64(), samequantumd128() — Determine if exponents X and Y are the same” on page 1460

quantized32(), quantized64(), quantized128() — Set the exponent of X to the exponent of Y

Standards

Standards / Extensions	C or C++	Dependencies
C/C++ DFP	both	z/OS V1.8

Format

```
#define __STDC_WANT_DEC_FP__
#include <math.h>

_Decimal32 quantized32(_Decimal32 x, _Decimal32 y);
_Decimal64 quantized64(_Decimal64 x, _Decimal64 y);
_Decimal128 quantized128(_Decimal128 x, _Decimal128 y);
```

General description

The quantize functions set the exponent of argument x to the exponent of argument y, while trying to keep the value the same. If the exponent is being increased, the value is correctly rounded according to the current rounding mode. If the result does not have the same value as x, the "inexact" (FP_INEXACT) floating-point exception is raised. If the exponent is being decreased, and the significand of the result has more digits than the type would allow, the result is NaN and the "invald" (FP_INVALID) floating-point exception is raised.

If one of both operands are NaN, the result is NaN, and the "invalid" floating-point exception may be raised. Otherwise, if only one operand is infinity, the result is NaN, and the "invalid" floating-point exception is raised. If both operands are infinity, the result is DEC_INFINITY, and the sign is the same as x.

The quantize functions do not signal underflow (FP_UNDERFLOW) or overflow (FP_OVERFLOW).

Argument
Description

x Input value to be converted and perhaps rounded using the exponent of y.

y Input value whose exponent is used for the output value

Usage notes

1. To use IEEE decimal floating-point, the hardware must have the Decimal Floating-Point Facility installed.
2. These functions work in IEEE decimal floating-point format. See "IEEE Decimal Floating-Point" for more information.

Returned value

The quantize functions return the number which is equal in value (except for any rounding) and sign to x, and which has been set to be equal to the exponent of y.

Example

```
/* CELEBQ02

   This example illustrates the quantized128() function.

*/

#define  __STDC_WANT_DEC_FP__
#include <stdio.h>
#include <math.h>

int main(void)
{
  _Decimal128 price = 64999.99DL;
  _Decimal128 rate  = 0.09875DL;
  _Decimal128 tax   = quantized128(price * rate, 0.01DL);
  _Decimal128 total = price + tax;

  printf( "price = %22.16DDF\n"
          "  tax = %22.16DDF (price * rate = %-.16DDF)\n"
          "total = %22.16DDF\n"
        ,  price
        ,    tax ,           price * rate
        ,  total
        );

  return 0;
}
```

Related information

- “math.h” on page 44
- “samequantumd32(), samequantumd64(), samequantumd128() — Determine if exponents X and Y are the same” on page 1460
- “quantexpd32(), quantexpd64(), quantexpd128() - Compute the quantum exponent” on page 1365

QueryMetrics() — Query WLM system information

Standards

Standards / Extensions	C or C++	Dependencies
z/OS UNIX	both	

Format

```
#include <sys/__wlm.h>

int QueryMetrics(struct sysi *sysi_ptr, int *anslen);
```

General description

The QueryMetrics() function provides the ability for an application to query WLM system information.

***sysi_ptr**
Points to a buffer that the service is to return the WLM system information. The data returned is in the format of the structure sysi.

***anslen**
Points to an integer data field that contains the length of the buffer to return the WLM system information into.

Returned value

If successful, QueryMetrics() returns 0.

If unsuccessful, QueryMetrics() returns -1 and sets errno to one of the following values. If the returned errno and __errno2() indicate the supplied buffer is too small, the *anslen* argument is updated to contain the length required to hold WLM system information.

Error Code
Description

EFAULT
An argument of this function contained an address that was not accessible to the caller.

EINVAL
An argument of this function contained an incorrect value.

EMVSSAF2ERR
An error occurred in the security product.

EMVSWLMERROR
The WLM query system information failed. Use __errno2() to obtain the WLM service reason code for the failure.

EPERM
The calling thread's address space is not permitted to the BPX.WLMSERVER Facility class. The caller's address space must be permitted to the BPX.WLMSERVER Facility class if it is defined. If BPX.WLMSERVER is not defined, the calling process is not defined as a superuser (UID=0).

Related information

- “sys/__wlm.h” on page 77
- “CheckSchEnv() — Check WLM scheduling environment” on page 271
- “ConnectServer() — Connect to WLM as a server manager” on page 313
- “ConnectWorkMgr() — Connect to WLM as a work manager” on page 315
- “ContinueWorkUnit() — Continue WLM work unit” on page 323
- “CreateWorkUnit() — Create WLM work unit” on page 344
- “DeleteWorkUnit() — Delete a WLM work unit” on page 378
- “DisconnectServer() — Disconnect from WLM server” on page 383
- “JoinWorkUnit() — Join a WLM work unit” on page 927
- “LeaveWorkUnit() — Leave a WLM work unit” on page 945
- “QuerySchEnv() — Query WLM scheduling environment”

QuerySchEnv() — Query WLM scheduling environment

Standards

Standards / Extensions	C or C++	Dependencies
z/OS UNIX	both	

Format

```
#include <sys/__wlm.h>

int QuerySchEnv(struct sethdr *sysi_ptr, int *anslen);
```

General description

The QuerySchEnv() function provides the ability for an application to query WLM scheduling environment.

***sysi_ptr**
Points to a buffer that the service is to return the WLM system information. The data returned is in the format of the structure sysi.

***anslen**
Points to an integer data field that contains the length of the buffer to return the WLM system information into.

Returned value

If successful, QuerySchEnv() returns 0.

If unsuccessful, QuerySchEnv() returns -1 and sets errno to one of the following values. If the returned errno and __errno2() indicate the supplied buffer is too small, the *anslen* argument is updated to contain the length required to hold WLM system information.

Error Code
Description

EFAULT
An argument of this function contained an address that was not accessible to the caller.

EINVAL
An argument of this function contained an incorrect value.

EMVSSAF2ERR
An error occurred in the security product.

EMVSWLMERROR
The WLM query scheduling environment failed. Use __errno2() to obtain the WLM service reason code for the failure.

EPERM
The calling thread's address space is not permitted to the BPX.WLMSERVER Facility class. The caller's address space must be permitted to the BPX.WLMSERVER Facility class if it is defined. If BPX.WLMSERVER is not defined, the calling process is not defined as a superuser (UID=0).

Related information

- “sys/__wlm.h” on page 77
- “CheckSchEnv() — Check WLM scheduling environment” on page 271
- “ConnectServer() — Connect to WLM as a server manager” on page 313
- “ConnectWorkMgr() — Connect to WLM as a work manager” on page 315
- “ContinueWorkUnit() — Continue WLM work unit” on page 323
- “CreateWorkUnit() — Create WLM work unit” on page 344
- “DeleteWorkUnit() — Delete a WLM work unit” on page 378
- “DisconnectServer() — Disconnect from WLM server” on page 383
- “JoinWorkUnit() — Join a WLM work unit” on page 927
- “LeaveWorkUnit() — Leave a WLM work unit” on page 945
- “QueryMetrics() — Query WLM system information” on page 1368

QueryWorkUnitClassification() — WLM query enclave classification service

Standards

Standards / Extensions	C or C++	Dependencies
z/OS UNIX	both	OS/390 V2R9

Format

```
#include <sys/__wlm.h>

int QueryWorkunitClassification(wlmetok_t *e_token,
                                struct sysec *sysec_ptr,
                                int *ans_len);
```

General description

Returns the classification attributes of an enclave, using the following parameters:

**e_token*
Points to a work unit enclave token.

**sysec_ptr*
Points to the enclave classification data (mapped by *sysec*) returned by the QueryWorkunitClassification function.

**anslen*
The length of the data area provided by the caller to receive the information generated by the service. WLM will update this value with the size of the area needed for the service to work. The minimum area should

hold the entire *sysec* structure through version 3. The existing area will be populated with as much of the information as can be returned.

Returned value

If successful, QueryWorkunitClassification() returns 0.

If unsuccessful, QueryWorkunitClassification() returns -1 and sets errno to one of the following values:

Error Code
Description

EFAULT
An argument of this function contained an address that was not accessible to the caller.

EINVAL
An argument of this function contained a value that is not correct.

EMVSSAF2ERR
An error occurred in the security product.

EMVSWLMERROR
A WLM service failed. Use __errno2() to obtain the WLM service reason code for the failure.

EPERM
The calling thread's address space is not permitted to the BPX.WLMSERVER Facility class. The caller's address space must be permitted to the BPX.WLMSERVER Facility class, if the BPX.WLMSERVER class is defined. If BPX.WLMSERVER is not defined, the calling process is not defined as a superuser (UID=0).

Related information

- “sys/__wlm.h” on page 77
- “ConnectExportImport() — WLM connect for export or import use” on page 312
- “ExportWorkUnit() — WLM export service” on page 452
- “ExtractWorkUnit() — Extract enclave service” on page 457
- “ImportWorkUnit() — WLM import service” on page 840
- “UnDoExportWorkUnit() — WLM undo export service” on page 1955
- “UnDoImportWorkUnit() — WLM undo import service” on page 1956
- For more information, see *z/OS MVS Programming: Workload Management Services, SC34-2663*

raise() — Raise signal

Standards

Standards / Extensions	C or C++	Dependencies
ISO C XPG4 XPG4.2 C99 Single UNIX Specification, Version 3	both	

Format

```
#include <signal.h>

int raise(int sig);
```

General description

Sends the signal *sig* to the program that issued raise(). See Table 50 on page 1645 for the list of signals supported.

You can use signal() to specify how a signal will be handled when raise() is invoked.

In C++ only, the use of signal() and raise() with try(), catch(), or throw() is undefined. The use of signal() and raise() with destructors is also undefined.

Special behavior for POSIX: To obtain access to the special POSIX behavior for raise(), the POSIX runtime option must be set ON, and the version of MVS must be 4.3 or higher.

The raise() function sends the signal, sig, to the process that issued the raise(). If the signal is not blocked, it is delivered to the sender before raise() returns. See Table 48 on page 1612 in the description of the sigaction() function for the list of signals supported.

You can use signal() or sigaction() to specify how a signal will be handled when raise() is invoked.

Special behavior for XPG4.2: To obtain access to the special POSIX behavior for raise(), the POSIX runtime option must be set ON, and the version of MVS must be 4.3 or higher.

Several other functions are available to the XPG4.2 application for affecting the behavior of a signal:
- bsd_signal()
- sigignore()
- sigset()

Special behavior for C++: The behavior when mixing signal-handling with C++ exception handling is undefined. Also, the use of signal-handling with constructors and destructors is undefined.

Usage note

The use of the `SIGTHSTOP` and `SIGTHCONT` signal is not supported with this function.

Returned value

If successful, raise() returns 0.

If unsuccessful, raise() returns nonzero.

Special behavior for XPG4: The raise() function sets errno to one of the following values:

Error Code
Description

EINVAL
The value of the *sig* argument is an invalid signal number.

Example

CELEBR01

```
/* CELEBR01

   This example establishes a signal handler called sig_hand for the
   signal SIGUSR1.
   The signal handler is called whenever the SIGUSR1 signal is raised and
   will ignore the first nine occurrences of the signal.
   On the tenth raised signal, it exits the program with an error code of 10.
   Note that the signal handler must be reestablished each time it is called.

 */
#include <signal.h>
#include <stdio.h>
#include <stdlib.h>

#ifdef __cplusplus
extern "C" {
#endif
 void sig_hand(int);
#ifdef __cplusplus
}
#endif
int i;

int main(void)
{
   signal(SIGUSR1, sig_hand);  /* set up handler for SIGUSR1 */
   for (i=0; i<10; ++i)
      raise(SIGUSR1);            /* signal SIGUSR1 is raised */
}                            /* sig_hand() is called */

void sig_hand(int dummy)
{
   static int count = 0;          /* initialized only once */

   count++;
   if (count == 10)   /* ignore the first 9 occurrences of this signal */
   {
      printf("reached 10th signal\n");
      exit(10);
   }
   else
      signal(SIGUSR1, sig_hand);  /* set up the handler again */
}
```

Related information

- “signal.h” on page 63
- “bsd_signal() — BSD version of signal()” on page 219
- “kill() — Send a signal to a process” on page 931
- “killpg() — Send a signal to a process group” on page 934
- “pthread_kill() — Send a signal to a thread” on page 1283
- “sigaction() — Examine or change a signal action” on page 1611
- “sighold() — Add a signal to a thread” on page 1636
- “sigignore() — Set disposition to ignore a signal” on page 1637
- “signal() — Handle interrupts” on page 1643
- “sigprocmask() — Examine or change a thread” on page 1651
- “sigset() — Change a signal action or a thread” on page 1656

rand() — Generate random number

Standards

Standards / Extensions	C or C++	Dependencies
ISO C POSIX.1 XPG4 XPG4.2 C99 Single UNIX Specification, Version 3	both	

Format

```
#include <stdlib.h>

int rand(void);
```

General description

Generates a pseudo-random integer in the range 0 to RAND_MAX. Use the srand() function before calling rand() to set a seed for the random number generator. If you do not make a call to srand(), the default seed is 1.

Returned value

Returns the calculated value.

Example

CELEBR02

```
/* CELEBR02

   This example prints the first 10 random numbers generated.

 */
#include <stdlib.h>
#include <stdio.h>

int main(void)
{
   int x;

   for (x = 1; x <= 10; x++)
      printf("iteration %d, rand=%d\n", x, rand());
}
```

Output

```
iteration 1, rand=16838
iteration 2, rand=5758
iteration 3, rand=10113
iteration 4, rand=17515
iteration 5, rand=31051
iteration 6, rand=5627
iteration 7, rand=23010
iteration 8, rand=7419
iteration 9, rand=16212
iteration 10, rand=4086
```

Related information

- "stdlib.h" on page 70
- "rand_r() — Pseudo-random number generator"
- "srand() — Set seed for rand() function" on page 1716

rand_r() — Pseudo-random number generator

Standards

Standards / Extensions	C or C++	Dependencies
Single UNIX Specification, Version 2 Single UNIX Specification, Version 3	both	OS/390 V2R8

Format

```
#define _XOPEN_SOURCE 500
#include <stdlib.h>

int rand_r(unsigned int *seed);
```

General description

The rand_r() function generates a sequence of pseudo-random integers in the range 0 to **RAND_MAX**. (The value of the **RAND_MAX** macro will be at least 32767.)

If rand_r() is called with the same initial value for the object pointed to by *seed* and that object is not modified between successive returns and calls to rand_r(), the same sequence shall be generated.

Returned value

rand_r() returns a pseudo-random integer.

There are no documented errno values.

Related information

- "stdlib.h" on page 70
- "rand() — Generate random number" on page 1374
- "srand() — Set seed for rand() function" on page 1716

random() — A better random-number generator

Standards

Standards / Extensions	C or C++	Dependencies
XPG4.2 Single UNIX Specification, Version 3	both	

Format

```
#define _XOPEN_SOURCE_EXTENDED 1
#include <stdlib.h>

long random(void);
```

General description

The random() function uses a nonlinear additive feedback random-number generator employing a default state array size of 31 long integers to return successive pseudo-random numbers in the range from 0 to 2^{31}-1. The period of this random-number generator is approximately 16x(2^{31}-1). The size of the state array determines the period of the random-number generator. Increasing the state array size increases the period.

With 256 bytes of state information, the period of the random-number generator is greater than 2^{69}.

Like rand(), random() produces by default a sequence of numbers that can be duplicated by calling srandom() with 1 as the seed. The state information for the random functions is maintained on a per-thread basis. For example, calls to srandom() in one thread will have no effect on the numbers generated by calls to random() in another thread.

Returned value

random() returns the generated pseudo-random number.

Related information

- "stdlib.h" on page 70
- "drand48() — Pseudo-random number generator" on page 402
- "initstate() — Initialize generator for random()" on page 867
- "setstate() — Change generator for random()" on page 1590
- "srandom() — Use seed to initialize generator for random()" on page 1718

read() — Read from a file or socket

Standards

Standards / Extensions	C or C++	Dependencies
POSIX.1 XPG4 XPG4.2 Single UNIX Specification, Version 3	both	

Format

```
#define_POSIX_SOURCE
#include <unistd.h>

ssize_t read(int fs, void *buf, size_t N);
```

X/Open

```
#define _XOPEN_SOURCE_EXTENDED 1
#include <unistd.h>

ssize_t read(int fs, void *buf, ssize_t N);
```

Berkeley sockets

```
#define _OE_SOCKETS
#include <unistd.h>

ssize_t read(int socket, void *buf, ssize_t N);
```

General description

From the file indicated by the file descriptor *fs*, the read() function reads *N* bytes of input into the memory area indicated by *buf*. A successful read() updates the access time for the file.

If *fs* refers to a regular file or any other type of file on which the process can seek, read() begins reading at the file offset associated with *fs*. If successful, read() changes the file offset by the number of bytes read. *N* should not be greater than INT_MAX (defined in the limits.h header file).

If *fs* refers to a file on which the process cannot seek, read() begins reading at the current position. There is no file offset associated with such a file.

If *fs* refers to a socket, read() is equivalent to recv() with no flags set.

Parameter	**Description**
fs	The file or socket descriptor.
buf	The pointer to the buffer that receives the data.
N	The length in bytes of the buffer pointed to by the *buf* parameter.

Behavior for sockets: The read() call reads data on a socket with descriptor *fs* and stores it in a buffer. The read() all applies only to connected sockets. This call returns up to *N* bytes of data. If there are fewer bytes available than requested, the call returns the number currently available. If data is not available for the socket *fs*, and the socket is in blocking mode, the read() call blocks the caller until data arrives. If data is not available, and the socket is in nonblocking mode, read() returns a -1 and sets the error code to EWOULDBLOCK. See “ioctl() — Control device” on page 869 or “fcntl() — Control open file descriptors” on page 474 for a description of how to set nonblocking mode.

For datagram sockets, this call returns the entire datagram that was sent, provided that the datagram fits into the specified buffer. Excess datagram data is discarded. Stream sockets act like streams of information with no boundaries separating data. For example, if applications A and B are connected with a stream socket and application A sends 1000 bytes, each call to this function can return 1 byte, or 10 bytes, or the entire 1000 bytes. Therefore, applications using stream sockets should place this call in a loop, calling this function until all data has been received.

Behavior for streams: A read() from a STREAMS file can read data in three different modes: byte-stream mode, message-nondiscard mode, and message-discard mode. The default is byte-stream mode. This can be changed using the `I_SRDOPT` ioctl() request, and can be tested with the `I_GRDOPT` ioctl(). In byte-stream mode, read() retrieves data from the STREAM until as many bytes as were requested are transferred, or until there is no more data to be retrieved. Byte-stream mode ignores message boundaries.

In STREAMS message-nondiscard mode, read() retrieves data until as many bytes as were requested are transferred, or until a message boundary is reached. If read() does not retrieve all the data in a message, the remaining data is left on the STREAM, and can be retrieved by the next read() call. Message-discard mode also retrieves data until as many bytes as were requested are transferred, or a message boundary is reached. However, unread data remaining in a message after the read() returns is discarded, and is not available for a subsequent read(), readv() or getmsg() call.

How read() handles zero-byte STREAMS messages is determined by the current read mode setting. In byte-stream mode, read() accepts data until it has read *N* bytes, or until there is no more data to read, or until a zero-byte message block is encountered. The read() function then returns the number of bytes read, and places the zero-byte message back on the STREAM to be retrieved by the next read(), readv() or getmsg(). In message-nondiscard mode or message-discard mode, a zero-byte message returns 0 and the message is removed from the STREAM. When a zero-byte message is read as the first message on a STREAM, the message is removed from the STREAM and 0 is returned, regardless of the read mode.

A read() from a STREAMS file returns the data in the message at the front of the STREAM head read queue, regardless of the priority band of the message.

By default, STREAMs are in control-normal mode, in which a read() from a STREAMS file can only process messages that contain a data part but do not contain a control part. The read() fails if a message containing a control part is encountered at the STREAM head. This default action can be changed by placing the STREAM in either control-data mode or control-discard mode with the `I_SRDOPT` ioctl() command. In control-data mode, read() converts any control part to data and passes it to the application before passing any data part originally present in the same message. In control-discard mode, read() discards message control parts but returns to the process any data part in the message.

In addition, read() and readv() will fail if the STREAM head had processed an asynchronous error before the call. In this case, the value of errno does not reflect the result of read() or readv() but reflects the prior error. If a hang-up occurs on the STREAM being read, read() continues to operate normally until the STREAM head read queue is empty. Thereafter, it returns 0.

Large file support for z/OS UNIX files: Large z/OS UNIX files are supported automatically for AMODE 64 C/C++ applications. AMODE 31 C/C++ applications must be compiled with the option LANGLVL(LONGLONG) and define the _LARGE_FILES feature test macro before any headers are included to enable this function to operate on z/OS UNIX files that are larger than 2 GB in size. File size and offset fields are enlarged to 63 bits in width. Therefore, any other function operating on the file is required to define the _LARGE_FILES feature test macro as well.

Returned value

If successful, read() returns the number of bytes actually read and placed in *buf*. This number is less than or equal to *N*. It is less than *N* only if:

- read() reached the end of the file before reading the requested number of bytes.
- read() was interrupted by a signal.
- In POSIX C programs only, the file is a pipe, FIFO special file, or a character special file that has fewer than *N* bytes immediately available for reading.
- If the Physical File System does not support simple reads from directories, read() will return 0 if it is used for a directory. Users should use Opendir() and readdir() instead.

In POSIX C programs only, if read() is interrupted by a signal, the effect is one of the following:

- If read() has not read any data yet, it returns -1 and sets errno to EINTR.
- If read() has successfully read some data, it returns the number of bytes it read before it was interrupted.

If the starting position for the read operation is at the end of the file or beyond, read() returns 0.

In POSIX C programs, if read() attempts to read from an empty pipe or a FIFO special file, it has one of the following results:

- If no process has the pipe open for writing, read() returns 0 to indicate the end of the file.
- If some process has the pipe open for writing and O_NONBLOCK is set to 1, read() returns -1 and sets errno to EAGAIN.
- If some process has the pipe open for writing and O_NONBLOCK is set to 0, read() blocks (that is, does not return) until some data is written, or the pipe is closed by all other processes that have the pipe open for writing.

With other files that support nonblocking read operations (for example, character special files), a similar principle applies:

- If data is available, read() reads the data immediately.
- If no data is available and O_NONBLOCK is set to 1, read() returns -1 and sets errno to EAGAIN.
- If no data is available and O_NONBLOCK is set to 0, read() blocks until some data becomes available.

read() causes the signal SIGTTIN to be sent when all these conditions exist:

- The process is attempting to read from its controlling terminal.
- The process is running in a background process group.
- The SIGTTIN signal is not blocked or ignored.
- The process group of the process is not orphaned.

If read() is reading a regular file and encounters a part of the file that has not been written (but before the end of the file), read() places 0 bytes into *buf* in place of the unwritten bytes.

If the number of bytes of input that you want to read is 0, read() simply returns 0 without attempting any other action.

If the connection is broken on a stream socket, but data is available, then the read() function reads the data and gives no error. If the connection is broken on a stream socket, but no data is available, then the read() function returns 0 bytes as EOF.

Note: z/OS UNIX services do not supply any STREAMS devices or pseudodevices. It is impossible for read() to read any data from a STREAMS-based file indicated by *fs*. It will always return -1 with errno set to EBADF. EINVAL will never be set because there are no multiplexing STREAMS drivers. See "open() — Open a file" on page 1152 for more information.

If unsuccessful, read() returns -1 and sets errno to one of the following:

Error Code
Description

EAGAIN
O_NONBLOCK is set to 1, but data was not available for reading.

EBADF
fs is not a valid file or socket descriptor.

ECONNRESET
A connection was forcibly closed by a peer.

EFAULT
Using the *buf* and *N* parameters would result in an attempt to access memory outside the caller's address space.

EINTR
read() was interrupted by a signal that was caught before any data was available.

EINVAL
N contains a value that is less than 0, or the request is invalid or not supported, or the STREAM or multiplexer referenced by *fs* is linked (directly or indirectly) downstream from a multiplexer.

EIO The process is in a background process group and is attempting to read from its controlling terminal, and either the process is ignoring or blocking the SIGTTIN signal or the process group of the process is orphaned. For sockets, an I/O error occurred.

ENOBUFS
Insufficient system resources are available to complete the call.

ENOTCONN
A receive was attempted on a connection-oriented socket that is not connected.

EOVERFLOW
The file is a regular file and an attempt was made to read or write at or beyond the offset maximum associated with the file.

ETIMEDOUT
The connection timed out during connection establishment, or due to a transmission timeout on active connection.

EWOULDBLOCK
socket is in nonblocking mode and data is not available to read. or the SO_RCVTIMEO timeout value was been reached before data was available.

Example

CELEBR03

```
/* CELEBR03

   This example opens a file and reads input.

 */
#define _POSIX_SOURCE
#include <fcntl.h>
#include <unistd.h>
#undef _POSIX_SOURCE
#include <stdio.h>

main() {
  int ret, fd;
  char buf[1024];
  system("ls -l / >| ls.output");

  if ((fd = open("ls.output", O_RDONLY)) < 0)
    perror("open() error");
  else {
    while ((ret = read(fd, buf, sizeof(buf)-1)) > 0) {
      buf[ret] = 0x00;
      printf("block read: \n<%s>\n", buf);
    }
    close(fd);
  }

  unlink("ls.output");
}
```

Output

```
block read:
<total 0
drwxr-xr-x   3 USER1    SYS1           0 Apr 16 07:59 bin
drwxr-xr-x   2 USER1    SYS1           0 Apr  6 10:20 dev
drwxr-xr-x   4 USER1    SYS1           0 Apr 16 07:59 etc
drwxr-xr-x   2 USER1    SYS1           0 Apr  6 10:15 lib
drwxrwxrwx   2 USER1    SYS1           0 Apr 16 07:55 tmp
drwxr-xr-x   2 USER1    SYS1           0 Apr  6 10:15 u
drwxr-xr-x   6 USER1    SYS1           0 Apr  6 10:15 usr
>
```

Related information

- "limits.h" on page 39
- "unistd.h" on page 82
- "close() — Close a file" on page 288
- "connect() — Connect a socket" on page 308
- "creat() — Create a new file or rewrite an existing one" on page 340
- "dup() — Duplicate an open file descriptor" on page 404
- "fcntl() — Control open file descriptors" on page 474
- "fread() — Read items" on page 610
- "getsockopt() — Get the options associated with a socket" on page 777
- "ioctl() — Control device" on page 869
- "lseek() — Change the offset of a file" on page 1020
- "open() — Open a file" on page 1152
- "pipe() — Create an unnamed pipe" on page 1179
- "pread() — Read from a file or socket without file pointer change" on page 1198
- "readv() — Read data on a file or socket and store in a set of buffers" on page 1389
- "recv() — Receive data on a socket" on page 1398
- "recvfrom() — Receive messages on a socket" on page 1401
- "recvmsg() — Receive messages on a socket and store in an array of message headers" on page 1405
- "select(), pselect() — Monitor activity on files or sockets and message queues" on page 1470
- "selectex() — Monitor activity on files or sockets and message queues" on page 1479
- "send() — Send data on a socket" on page 1494
- "sendmsg() — Send messages on a socket" on page 1500
- "sendto() — Send data on a socket" on page 1506
- "setsockopt() — Set options associated with a socket" on page 1578
- "socket() — Create a socket" on page 1688
- "write() — Write data on a file or socket" on page 2085
- "writev() — Write data on a file or socket from an array" on page 2092

readdir() — Read an entry from a directory

Standards

Standards / Extensions	C or C++	Dependencies
POSIX.1 XPG4 XPG4.2 Single UNIX Specification, Version 3	both	

Format

```
#define_POSIX_SOURCE
#include <dirent.h>

struct dirent *readdir(DIR *dir);
```

General description

Returns a pointer to a `dirent` structure describing the next directory entry in the directory stream associated with *dir*.

A call to readdir() overwrites data produced by a previous call to readdir() or __readdir2() on the same directory stream. Calls for different directory streams do not overwrite each other's data.

Each call to readdir() updates the `st_atime` (access time) field for the directory.

A `dirent` structure contains the character pointer *d_name*, which points to a string that gives the name of a file in the directory. This string ends in a terminating NULL, and has a maximum of NAME_MAX characters.

Save the data from readdir(), if required, before calling closedir(), because closedir() frees the data.

If the contents of a directory have changed since the directory was opened (files added or removed); a call should be made to rewinddir() so that subsequent readdir() requests can read the new contents.

Special behavior for XPG4: If entries for dot or dot-dot exist, one entry will be returned for dot and one entry will be returned for dot-dot; otherwise they will not be returned.

After a call to fork(), either the parent or child (but not both) may continue processing the directory stream using __readdir2(), readdir(), rewinddir(), or seekdir(). If both the parent and child processes use these functions, the result is undefined.

Special behavior for XPG4.2: If the entry names a symbolic link, the value of `d_ino` member in *dirent* structure is unspecified.

Returned value

If successful, readdir() returns a pointer to a `dirent` structure describing the next directory entry in the directory stream. When readdir() reaches the end of the directory stream, it returns a NULL pointer.

If unsuccessful, readdir() returns a NULL pointer and sets errno to one of the following values:

Error Code
Description

EBADF
dir does not yield an open directory stream.

EINVAL
The buffer was too small to contain any directories.

ENOENT
The current position of the directory stream is invalid.

EOVERFLOW
One of the values in the structure to be returned cannot be represented correctly.

Note: The environment variable _EDC_SUSV3 can be used to control the behavior of readdir() with respect to detecting an EOVERFLOW condition. By default, readdir() will not detect that values in the structure returned can be represented correctly. When _EDC_SUSV3 is set to 1, readdir() will check for overflow conditions.

Example

CELEBR04

```
/* CELEBR04

   This example reads the contents of a root directory.

 */
#define _POSIX_SOURCE
#include <dirent.h>
#include <errno.h>
#include <sys/types.h>
#undef _POSIX_SOURCE
#include <stdio.h>

main() {
  DIR *dir;
  struct dirent *entry;

  if ((dir = opendir("/")) == NULL)
    perror("opendir() error");
  else {
    puts("contents of root:");
    while ((entry = readdir(dir)) != NULL)
      printf("  %s\n", entry->d_name);
    closedir(dir);
  }
}
```

Output

```
contents of root:
  .
  ..
  bin
  dev
  etc
```

```
lib
tmp
u
usr
```

Related information

- "dirent.h" on page 22
- "stdio.h" on page 68
- "sys/types.h" on page 75
- "closedir() — Close a directory" on page 291
- "opendir() — Open a directory" on page 1158
- "__opendir2() — Open a directory" on page 1160
- "readdir_r() — Read an entry from a directory" on page 1386
- "__readdir2() — Read directory entry and get file information"
- "rewinddir() — Reposition a directory stream to the beginning" on page 1446
- "seekdir() — Set position of directory stream" on page 1469
- "telldir() — Current location of directory stream" on page 1866

__readdir2() — Read directory entry and get file information

Standards

Standards / Extensions	C or C++	Dependencies
z/OS UNIX	both	OS/390 V2R6

Format

```
#define _OPEN_SYS_DIR_EXT
#include <dirent.h>

struct dirent *__readdir2(DIR *dir, struct stat *info);
```

General description

The __readdir2() function returns a pointer to a dirent structure describing the next directory entry in the directory stream associated with *dir*.

A dirent structure contains the character pointer *d_name*, which points to a string that gives the name of a file in the directory. This string ends in a terminating NULL, and has a maximum of NAME_MAX characters.

The *info* argument points to an area of storage that will be filled in with information about the file *d_name*. This information is returned in a stat structure defined in the sys/stat.h header file. The format of this structure is described in the section about the lstat() function. If *info* is NULL, no stat information is passed back.

If entries for dot or dot-dot exist, one entry will be returned for dot and one entry will be returned for dot-dot; otherwise they will not be returned.

A call to __readdir2() overwrites data produced by a previous call to __readdir2() or readdir() on the same directory stream. Calls for different directory streams do not overwrite each other's data.

Save the dirent data from __readdir2(), if required, before calling closedir(), because closedir() frees the dirent data.

The __readdir2() function may buffer several directory entries per actual read operation. __readdir2() updates the st_atime (access time) field of the directory each time the directory is actually read.

After a call to fork(), either the parent or child (but not both) may continue processing the directory stream using __readdir2(), readdir(), rewinddir() or seekdir(). If both the parent and child processes use these functions, the result is undefined.

If the entry names a symbolic link, the value of d_ino member in *dirent* structure is unspecified.

Unpredictable results can occur if closedir() is used to close the the directory stream before __readdir2() is called. If the contents of a directory have changed since the directory was opened (files added or removed), a call should be made to rewinddir() so that subsequent _readdir2() requests can read the new contents.

The output from this function is similar to a combination of readdir() and lstat(). In some cases, certain information in the output stat structure differs from what lstat() would return. Also, the d_extra field in *dir* is always NULL for __readdir2().

Returned value

If successful, __readdir2() returns a pointer to a `dirent` structure describing the next directory entry in the directory stream. When __readdir2() reaches the end of the directory stream, it returns a NULL pointer.

If unsuccessful, __readdir2() returns a NULL pointer and sets errno to one of the following values:

Error Code
Description

EBADF
dir does not yield an open directory stream.

EINVAL
The buffer was too small to contain any directories.

ELOOP
A loop exists in symbolic links. This error occurs if the number of symbolic links in a file name in the directory is greater than POSIX_SYMLOOP.

ENOENT
The current position of the directory stream is invalid.

Related information

- “dirent.h” on page 22
- “stdio.h” on page 68
- “sys/types.h” on page 75
- “closedir() — Close a directory” on page 291
- “opendir() — Open a directory” on page 1158
- “__opendir2() — Open a directory” on page 1160
- “readdir() — Read an entry from a directory” on page 1382
- “readdir_r() — Read an entry from a directory” on page 1386
- “rewinddir() — Reposition a directory stream to the beginning” on page 1446
- “seekdir() — Set position of directory stream” on page 1469
- “telldir() — Current location of directory stream” on page 1866

readdir_r() — Read an entry from a directory

Standards

Standards / Extensions	C or C++	Dependencies
Single UNIX Specification, Version 2 Single UNIX Specification, Version 3	both	OS/390 V2R8

Format

```
#define _XOPEN_SOURCE 500
#include <dirent.h>

int readdir_r(DIR *__restrict__dir, struct dirent *__restrict__entry,
              struct dirent **__restrict__result);
```

General description

The readdir_r() function initializes the dirent structure referenced by *entry* to represent the directory entry at the current position in the directory stream referred to by *dir*, stores a pointer to this structure at the location referenced by *result*, and positions the directory stream at the next entry.

The storage pointed to by *entry* will be large enough for a dirent with an array of char d_name member containing at least **NAME_MAX**+1 elements.

On successful return, the pointer returned at **result* will have the same value as the argument *entry*. Upon reaching the end of the directory stream, this pointer will have the value NULL.

The readdir_r() function will not return directory entries containing empty names. It is unspecified whether entries are returned for dot or dot-dot.

If a file is removed from or added to the directory after the most recent call to opendir() or rewinddir(), whether a subsequent call to readdir_r() returns an entry for that file is unspecified.

The readdir_r() function may buffer several directory entries per actual read operation. The readdir_r() function marks for update the st_atime field of the directory each time the directory is actually read.

Applications wishing to check for error situations should set errno to 0 before calling readdir(). If errno is set to non-zero on return, an error occurred.

Returned value

If successful, readdir_r() returns 0.

If unsuccessful, readdir_r() sets errno to one of the following values:

Error Code
    Description

EBADF
    dir does not refer to an open directory stream.

Related information

- “dirent.h” on page 22
- “stdio.h” on page 68
- “sys/types.h” on page 75
- “closedir() — Close a directory” on page 291
- “opendir() — Open a directory” on page 1158
- “__opendir2() — Open a directory” on page 1160
- “readdir() — Read an entry from a directory” on page 1382
- “__readdir2() — Read directory entry and get file information” on page 1384
- “rewinddir() — Reposition a directory stream to the beginning” on page 1446
- “seekdir() — Set position of directory stream” on page 1469
- “telldir() — Current location of directory stream” on page 1866

readlink() — Read the value of a symbolic link

Standards

Standards / Extensions	C or C++	Dependencies
POSIX.1a XPG4.2 Single UNIX Specification, Version 3	both	

Format

```
#define _POSIX1_SOURCE 2
#include <unistd.h>

int readlink(const char *path,
                 char *buf, size_t bufsiz);

#define _POSIX_C_SOURCE 200112L
#include <unistd.h>

ssize_t readlink(const char *__restrict__path,
                 char *__restrict__buf, size_t bufsiz);
```

General description

Places the contents of the symbolic link *path* in the buffer *buf*. The size of the buffer is set by *bufsiz*. The result stored in *buf* does not include a terminating NULL character.

If the buffer is too small to contain the value of the symbolic link, that value is truncated to the size of the buffer (*bufsiz*). If the value returned is the size of the buffer, use lstat() to determine the actual size of the symbolic link.

Returned value

If successful, when *bufsiz* is greater than 0, readlink() returns the number of bytes placed in the buffer. When *bufsiz* is 0 and readlink() completes successfully, it returns the number of bytes contained in the symbolic link and the buffer is not changed.

If the returned value is equal to *bufsiz*, you can determine the contents of the symbolic link with either lstat() or readlink(), with a 0 value for *bufsiz*.

If unsuccessful, readlink() returns -1 and sets errno to one of the following values:

Error Code
Description

EACCES
Search permission is denied for a component of the path prefix.

EINVAL
The named file is not a symbolic link.

EIO An I/O error occurred while reading from the file system.

ELOOP
A loop exists in symbolic links. This error is issued if more than POSIX_SYMLOOP symbolic links are encountered during resolution of the *path* argument.

ENAMETOOLONG
pathname is longer than **PATH_MAX** characters, or some component of *pathname* is longer than **NAME_MAX** characters while _POSIX_NO_TRUNC is in effect. For symbolic links, the length of the path name string substituted for a symbolic link exceeds **PATH_MAX**. The **PATH_MAX** and **NAME_MAX** values can be determined using pathconf().

ENOENT
The named file does not exist.

ENOTDIR
A component of the path prefix is not a directory.

Example

CELEBR05

```
/* CELEBR05 */
#define _POSIX_SOURCE 1
#include <fcntl.h>
#include <sys/stat.h>
#include <sys/types.h>
#include <unistd.h>
#undef _POSIX_SOURCE

main() {
  char fn[]="readlink.file";
  char sl[]="readlink.symlink";
  char buf[30];
  int  fd;

  if ((fd = creat(fn, S_IWUSR)) < 0)
    perror("creat() error");
  else {
    close(fd);
    if (symlink(fn, sl) != 0)
      perror("symlink() error");
    else {
      if (readlink(sl, buf, sizeof(buf)) < 0)
        perror("readlink() error");
      else printf("readlink() returned '%s' for '%s'\n", buf, sl);

      unlink(sl);
    }
    unlink(fn);
  }
}
```

Output

```
readlink() returned 'readlink.file' for 'readlink.symlink'
```

Related information

- “unistd.h” on page 82
- “lstat() — Get status of file or symbolic link” on page 1022
- “stat() — Get file information” on page 1720
- “symlink() — Create a symbolic link to a path name” on page 1795
- “unlink() — Remove a directory entry” on page 1962

readv() — Read data on a file or socket and store in a set of buffers

Standards

Standards / Extensions	C or C++	Dependencies
XPG4.2 Single UNIX Specification, Version 3	both	

Format

X/Open

```
#define _XOPEN_SOURCE_EXTENDED 1
#include <sys/uio.h>

ssize_t readv(int fs, const struct iovec *iov, int iovcnt);
```

Berkeley sockets

```
#define _OE_SOCKETS
#include <sys/uio.h>

int readv(int fs, struct iovec *iov, int iovcnt);
```

General description

The readv() function reads data from a file or a socket with descriptor *fs* and stores it in a set of buffers. The data is scattered into the buffers specified by iov[0]...iov[iovcnt-1].

Parameter
Description

fs The file or socket descriptor.

iov A pointer to an **iovec** structure.

iovcnt The number of buffers pointed to by the *iov* parameter.

The **iovec** structure is defined in **uio.h** and contains the following fields:

Element
Description

iov_base
The pointer to the buffer.

iov_len The length of the buffer.

If the descriptor refers to a socket, then it must be a connected socket.

This call returns a number of bytes of data equal to but not exceeding the sum of all the *iov_len* fields. If less than the number of bytes requested is available, the call returns the number currently available. If data is not available for the socket *fs*, and the socket is in blocking mode, readv() call blocks the caller until data arrives. If data is not available and *fs* is in nonblocking mode, readv() returns a -1 and sets the error code to EWOULDBLOCK. See "fcntl() — Control open file descriptors" on page 474 or "ioctl() — Control device" on page 869 for a description of how to set nonblocking mode.

For datagram sockets, this call returns the entire datagram that was sent, provided that the datagram fits into the specified buffer. Excess datagram data is discarded. Stream sockets act like streams of information with no boundaries separating data. For example, if applications A and B are connected with a stream socket and application A sends 1000 bytes, each call to this function can return 1 byte, or 10 bytes, or the entire 1000 bytes. Therefore, applications using stream sockets should place this call in a loop, calling this function until all data has been received.

For X/Open sockets, if the total number of bytes to read is 0, readv() returns 0. If readv() is for a file and no data is available, readv() returns 0. If a readv() is interrupted by a signal before it reads any data, it returns -1 with errno set to EINTR. If readv() is interrupted by a signal after it has read data, it returns the number of bytes read. If *fs* refers to a socket, readv() is the equivalent of recv() with no flags set.

Special behavior for C++: To use this function with C++, you must use the _XOPEN_SOURCE_EXTENDED 1 feature test macro.

Returned value

If successful, readv() returns the number of bytes read into the buffer.

If the connection is broken on a stream socket and data is available, then the readv() function reads the data and gives no error. If the connection is broken on a stream socket and no data is available, then the readv() function returns 0 bytes as EOF.

If unsuccessful, readv() returns -1 and sets errno to one of the following values:

Error Code
Description

EAGAIN
The **O_NONBLOCK** flag is set for the file descriptor and the process would be delayed by the readv().

EBADF
fs is not a valid file or socket descriptor.

ECONNRESET
A connection was forcibly closed by a peer.

EFAULT
Using *iov* and *iovcnt* would result in an attempt to access storage outside the caller's address space.

EINTR
readv() was interrupted by a signal that was caught before any data was available.

EINVAL
iovcnt was not valid, or one of the fields in the *iov* array was not valid.

ENOBUFS
Insufficient system resources are available to complete the call.

ENOTCONN
A receive is attempted on a connection-oriented socket that is not connected.

ETIMEDOUT
The connection timed out during connection establishment, or due to a transmission timeout on active connection.

EWOULDBLOCK
socket is in nonblocking mode and data is not available to read. or the SO_RCVTIMEO timeout value was been reached before data was available.

Related information

- “sys/uio.h” on page 76
- “connect() — Connect a socket” on page 308
- “fcntl() — Control open file descriptors” on page 474
- “getsockopt() — Get the options associated with a socket” on page 777
- “ioctl() — Control device” on page 869
- “read() — Read from a file or socket” on page 1376
- “recv() — Receive data on a socket” on page 1398
- “recvfrom() — Receive messages on a socket” on page 1401
- “recvmsg() — Receive messages on a socket and store in an array of message headers” on page 1405
- “select(), pselect() — Monitor activity on files or sockets and message queues” on page 1470
- “selectex() — Monitor activity on files or sockets and message queues” on page 1479
- “send() — Send data on a socket” on page 1494
- “sendmsg() — Send messages on a socket” on page 1500
- “sendto() — Send data on a socket” on page 1506
- “setsockopt() — Set options associated with a socket” on page 1578
- “socket() — Create a socket” on page 1688
- “write() — Write data on a file or socket” on page 2085
- “writev() — Write data on a file or socket from an array” on page 2092

realloc() — Change reserved storage block size

Standards

Standards / Extensions	C or C++	Dependencies
ISO C POSIX.1 XPG4 XPG4.2 C99 Single UNIX Specification, Version 3	both	

Format

```
#include <stdlib.h>

void *realloc(void *ptr, size_t size);
```

General description

Changes the size of a previously reserved storage block. The *ptr* argument points to the beginning of the block. The *size* argument gives the new size of the block in bytes. The contents of the block are unchanged up to the shorter of the new and old sizes.

If the *ptr* is NULL, realloc() reserves a block of storage of *size* bytes. It does not give all bits of each element an initial value of 0.

If *size* is 0 and *ptr* is not NULL, the storage pointed to by *ptr* is freed and NULL is returned.

If you use realloc() with a pointer that does not point to a *ptr* created previously by malloc(), calloc(), or realloc(), or if you pass *ptr* to storage already freed, you get undefined behavior—usually an exception.

If you ask for more storage, the contents of the extension are undefined and are not guaranteed to be 0.

The storage to which the returned value points is aligned for storage of any type of object. Under z/OS XL C only, if 4K alignment is required, the __4kmalc() function should be used. (This function is only available to C applications in stand-alone System Programming C (SPC) Facility applications.) The library functions specific to the System Programming C (SPC) environment are described in *z/OS XL C/C++ Programming Guide*.

To investigate the cause of realloc() running out of heap storage, see *z/OS Language Environment Programming Reference*

Note: The environment variable *_CEE_REALLOC_CONTROL* controls reallocation that can lead to improved application performance. For more information about the *_CEE_REALLOC_CONTROL* environment variable, see *z/OS XL C/C++ Programming Guide*.

Special behavior for C++: The C++ keywords new and delete are not interoperable with calloc(), free(), malloc(), or realloc().

Returned value

If successful, realloc() returns a pointer to the reallocated storage block. The storage location of the block might be moved. Thus, the returned value is not necessarily the same as the *ptr* argument to realloc().

The returned value is NULL if *size* is 0. If there is not enough storage to expand the block to the given size, the original block is unchanged and a NULL pointer is returned. If realloc() returns NULL because there is not enough storage, it will also set errno to one of the following values:

Error Code
Description

ENOMEM
Insufficient memory is available

Example

CELEBR06

```
/* CELEBR06

   This example allocates storage for the prompted size of array
   and then uses &realloc. to reallocate the block to hold the
   new size of the array.
   The contents of the array are printed after each allocation.

 */
#include <stdio.h>
#include <stdlib.h>

int main(void)
{
  long * array;    /* start of the array */
  long * ptr;      /* pointer to array   */
  int    i;        /* index variable     */
  int  num1, num2; /* number of entries of the array */

  void print_array( long *ptr_array, int size);

  printf( "Enter the size of the array\n" );
  scanf( "%i", &num1 );

  /* allocate num1 entries using malloc() */
  if ( (array = (long *)malloc( num1 * sizeof( long ))) != NULL ) {
     for ( ptr = array, i = 0; i < num1 ; ++i ) /* assign values */
        *ptr++ = i;
     print_array( array, num1 );
     printf("\n");
  }
  else { /*  malloc error  */
    printf( "Out of storage\n" );
    abort();
  }

  /* Change the size of the array ... */
  printf( "Enter the size of the new array\n" );
  scanf( "%i", &num2);

  if ( (array = (long *)realloc( array, num2* sizeof( long ))) != NULL )
  {
     for ( ptr = array + num1, i = num1; i <= num2; ++i )
        *ptr++ = i + 2000;  /* assign values to new elements */
     print_array( array, num2 );
  }

  else { /* realloc error */
    printf( "Out of storage\n" );
    abort();
  }
}

void print_array( long  * ptr_array, int size )
{
  int i;
  long * index = ptr_array;

  printf("The array of size %d is:\n", size);
  for ( i = 0; i < size; ++i )           /* print the array out    */
    printf( "  array[ %i ] = %li\n", i, ptr_array[i] );
}
```

Output: If the initial value entered is 2 and the second value entered is 4, then expect the following output:

```
Enter the size of the array
The array of size 2 is:
  array[ 0 ] = 0
  array[ 1 ] = 1

Enter the size of the new array
The array of size 4 is:
  array[ 0 ] = 0
  array[ 1 ] = 1
  array[ 2 ] = 2002
  array[ 3 ] = 2003
```

Related information

- “System Programming C (SPC) Facilities” in *z/OS XL C/C++ Programming Guide*
- “spc.h” on page 64
- “stdlib.h” on page 70
- “calloc() — Reserve and initialize storage” on page 231
- “free() — Free a block of storage” on page 618
- “malloc() — Reserve storage block” on page 1031

realpath() — Resolve path name

Standards

Standards / Extensions	C or C++	Dependencies
XPG4.2 Single UNIX Specification, Version 3	both	

Format

```
#define _XOPEN_SOURCE_EXTENDED 1
#include <stdlib.h>

char *realpath(const char *__restrict__file_name, char *__restrict__resolved_name);
```

General description

The realpath() function derives, from the path name pointed to by *file_name*, an absolute path name that names the same file, whose resolution does not involve ".","..", or symbolic links. The generated path name is stored, up to a maximum of **PATH_MAX** bytes, in the buffer pointed to by *resolved_name*.

Returned value

If successful, realpath() returns a pointer to the resolved name.

If unsuccessful, the contents of the buffer pointed to by *resolved_name* are undefined, realpath() returns a NULL pointer and sets errno to one of the following values:

Error Code
Description

EACCES
Read or search permission was denied for a component of *file_name*.

EINVAL
Either the *file_name* or *resolved_name* argument is a NULL pointer.

EIO An error occurred while reading from the file system.

ELOOP
Too many symbolic links were encountered in resolving *path*

ENAMETOOLONG
Path name is longer that **PATH_MAX** characters, or some component of path name is longer that **NAME_MAX** characters while _POSIX_NO_TRUNC is in effect. For symbolic links, the length of the path name string substituted for a symbolic link exceeds **PATH_MAX**. The **PATH_MAX** and **NAME_MAX** values are determined using pathconf().

ENOENT
A component of *file_name* does not name an existing file or *file_name* points to an empty string.

ENOTDIR
A component of the path prefix is not a directory.

ERANGE
File system will return ERANGE if the result to be stored in 'resolved_name' is larger than **PATH_MAX**.

Related information

- "stdlib.h" on page 70
- "getcwd() — Get path name of the working directory" on page 697
- "sysconf() — Determine system configuration options" on page 1798

re_comp() — Compile regular expression

Standards

Standards / Extensions	C or C++	Dependencies
XPG4.2	both	

Format

```
#define _XOPEN_SOURCE_EXTENDED 1
#include <re_comp.h>

char *re_comp(const char *string);
```

General description

Restriction: This function is not supported in AMODE 64.

The re_comp() function converts a regular expression string into an internal form suitable for pattern matching by re_exec().

The parameter *string* is a pointer to a character string defining a source regular expression to be compiled.

If re_comp() is called with a NULL argument, the current regular expression remains unchanged.

Strings passed to re_comp() must be terminated by a NULL byte, and may include newline characters.

Notes:

1. The re_comp() and re_exec() functions are supported on the thread-level. They must be issued from the *same* thread to work properly.
2. The re_comp() and re_exec() functions are provided for historical reasons. These functions were part of the Legacy Feature in Single UNIX Specification, Version 2. They have been withdrawn and are not supported as part of Single UNIX Specification, Version 3. New applications should use the newer functions fnmatch(), glob(), regcomp() and regexec(), which provide full internationalized regular expression functionality compatible with IEEE Std 1003.1-2001.
3. The z/OS UNIX implementation of the re_comp() function supports only the POSIX locale. Any other locales will yield unpredictable results.

The re_comp() function supports simple regular expressions, which are defined below.

Simple regular expressions: A Simple Regular Expression (SRE) specifies a set of character strings. The simplest form of regular expression is a string of characters with no special meaning. A small set of special characters, known as metacharacters, do have special meaning when encountered in patterns.

The following one-character regular expressions (RE) match a single character:

1. An ordinary character *c* (*not* a special character) is a one character regular expression that matches itself.
2. A backslash (\) followed by any special character (that is, \c where *c* is any special character) is a one character regular expression that matches the special character itself. The special characters are:
   a. **.**, *, [, and \ (period, asterisk, left square bracket, and backslash, respectively) which are always special, except when they appear within square brackets ([]).
   b. ^(caret or circumflex), which is special at the beginning of the entire regular expression, or when it immediately follows the left of a pair of square brackets ([]).
   c. $ (dollar symbol), which is special at the end of the regular expression.
   d. The character used to bound (delimit) an entire regular expression, which is special for that regular expression.

   Note: A backslash (\) followed by an ordinary character is a one character regular expression that matches the ordinary character itself.
3. A period (.) is a one-character RE that matches any character, except newline.
4. A non-empty string within square brackets ([*string*]) is a one-character RE that matches any one character in that *string*. Thus, `[abc]`, if compared to other strings, would match any which contained a, b, or c.

   If the caret symbol (^) is the first character of the string within square brackets (that is, `[^string]`), the one-character RE matches any characters except newline and the remaining characters within the square brackets. Thus, `[^abc]`, if compared to other strings, would *fail* to match any which contains even one a, b, or c.

Ranges may be specified as *c*–*c*. The hyphen symbol, within square brackets, means "through". It may be used to indicate a range of consecutive ASCII characters. For example, [0–9] is equivalent to [0123456789].

The – (hyphen) can be used by itself, but only if it is the first (after an initial ^, if any), or last character in the expression.

The right square bracket (]) can be used as part of the string but only if it is the first character within it (after an initial ^, if any). For example, the expression []a–d] matches either a right square bracket or one of the characters `a` through `d`.

The following rules may be used to construct REs from one character REs:

1. A one-character RE is a RE that matches whatever the one-character RE matches.
2. A one-character RE followed by an asterisk symbol (*) is a RE that matches 0 or more occurrences of the one-character RE. For example, (`a*e`) will match any of the following: `e, ae, aaaaae`. The longest leftmost match is chosen.
3. A one-character RE followed by \{*m*\}, \{*m*,\}, or \{*m*,*u*\} is a RE that matches a range of occurrences of the one-character RE. Nonnegative integer values enclosed in \{\} indicate the number of times to apply the preceding one-character RE. *m* is the minimum number and *u* is the maximum number. *u* must be less than 256. If you specify only *m*, it indicates the exact number of times to apply the regular expression.

   \{*m*,\} is equivalent to \{*m*,*u*\}. They both match *m* or more occurrences of the expression. The * (asterisk) operation is equivalent to \{0,\}.

   The maximum number of occurrences is matched.
4. REs can be concatenated. The concatenation of REs is a RE that matches the concatenation of the strings matched by each component of the RE.
5. A RE enclosed between the character sequences \(*and*\) is a RE that matches whatever the unadorned RE matches. The \(and \) sequences are ignored.
6. The expression *n* (where 1 <= *n* <= 9) matches the same string of characters as was matched by an expression enclosed between \(and \) earlier in the same regular expression. The sub-expression it specified is that beginning with the *n*th occurrence of \(counting from the left. For example, in the expression, \(*a*\)*r*\(*e*\)\1, the \1 is equivalent to *a*, giving *area*.

An entire RE may be constrained to match only an initial segment or final segment of a line (or both).

1. A caret (^) at the beginning of an entire RE constrains that RE to match an initial segment of a line.
2. A dollar symbol ($) at the end of an entire RE constrains that RE to match a final segment of a line. For example, the construct ^*entire RE*$ constrains the entire RE to match the entire line.

Returned value

If the string pointed to by the *string* argument is successfully converted, re_comp() returns a NULL pointer.

If unsuccessful, re_comp() returns a pointer to an error message string (NULL-terminated).

The following re_comp() error messages are defined:

```
EDC7008E No previous regular expression
EDC7009E Regular expression too long
EDC7010E \(\) imbalance
EDC7011E \{\} imbalance
EDC7012E [] imbalance
EDC7013E Too many \(\) pairs.
EDC7014E Incorrect range values in \{\}
EDC7015E Back reference number in \digit incorrect
EDC7016E Incorrect endpoint in range expression
```

Note: The error message string is not to be freed by the application. It will be freed when the thread terminates.

Related information

- "re_comp.h" on page 61
- "fnmatch() — Match file name or path name" on page 564
- "glob() — Generate path names matching a pattern" on page 807
- "re_exec() — Match regular expression" on page 1410
- "regcomp() — Compile regular expression" on page 1415
- "regexec() — Execute compiled regular expression" on page 1421

recv() — Receive data on a socket

Standards

Standards / Extensions	C or C++	Dependencies
XPG4.2 Single UNIX Specification, Version 3	both	

Format

X/Open

```
#define _XOPEN_SOURCE_EXTENDED 1
#include <sys/socket.h>

ssize_t recv(int socket, void *buffer, size_t length, int flags);
```

Berkeley sockets

```
#define _OE_SOCKETS
#include <sys/socket.h>

int recv(int socket, char *buffer, int length, int flags);
```

General description

The recv() function receives data on a socket with descriptor *socket* and stores it in a buffer. The recv() call applies only to connected sockets.

Parameter
Description

socket The socket descriptor.

buf The pointer to the buffer that receives the data.

len The length in bytes of the buffer pointed to by the *buf* parameter. If the MSG_CONNTERM flag is set, the length of the buffer must be zero.

flags The *flags* parameter is set by specifying one or more of the following flags.

If more than one flag is specified, the logical OR operator (|) must be used to separate them. The MSG_CONNTERM flag is mutually exclusive with other flags.

MSG_CONNTERM
Requests that the function completes only when a TCP connection is terminated. It is valid for TCP sockets only. Other normal receive requests are also completed. The application must be able to deal with the fact that a normal receive and this special connection termination receive might be driven in parallel.

AT-TLS considerations: If AT-TLS is being used to provide transparent TLS/SSL support for a TCP socket and a receive request with MSG_CONNTERM is outstanding, AT-TLS will immediately honor any TLS/SSL close notify alerts sent by the peer and initiate TLS/SSL session shutdown. For more information about AT-TLS and determining whether a TCP connection is using AT-TLS, see *z/OS Communications Server: IP Programmer's Guide and Reference*.

MSG_OOB
Reads any out-of-band data on the socket. Out-of-band data is sent when the MSG_OOB flag is on for a send(), sendto(), or sendmsg().

The fcntl() command should be used with F_SETOWN to specify the recipient, either a pid or a gid, of a SIGURG signal that will be sent when out-of-band data is sent. If no recipient is set, no signal will be sent. For more information, see the fcntl() command. The recipient of the data determines whether to receive out-of-band data inline or not inline by the setting of the SO_OOBINLINE option of setsockopt(). If SO_OOBINLINE is set off and the MSG_OOB flag is set on, the out-of-band data byte will be read out-of-line. It is invalid for the MSG_OOB flag to be set on when SO_OOBINLINE is set on. If there is out-of-band data available, and the MSG_OOB flag is not set (SO_OOBINLINE can be on or off), then the data up to, but not including, the out-of-band data will be read. When the read cursor has reached the out-of-band data byte, then only the out-of-band data will be read on the next read. The SIOCATMARK option of ioctl() can be used to determine if the read cursor is currently at the out-of-band data byte. For more information, refer to the setsockopt() and ioctl() commands.

MSG_PEEK
Peeks at the data present on the socket; the data is returned but not consumed, so that a subsequent receive operation sees the same data.

MSG_WAITALL
Requests that the function block until the full amount of data requested can be returned. The function may return a smaller amount of data if a signal is caught, the connection is terminated, an error is pending, or SO_RCVTIMEO is set and the timer is expired for the socket.

This call returns the length of the incoming message or data. If a datagram packet is too long to fit in the supplied buffer, datagram sockets discard excess bytes. If data is not available for the socket *socket*, and *socket* is in blocking mode, the recv() call blocks the caller until data arrives. If data is not available and *socket* is in nonblocking mode, recv() returns a -1 and sets the error code to EWOULDBLOCK. See “fcntl() — Control open file descriptors” on page 474 or “ioctl() — Control device” on page 869 for a description of how to set nonblocking mode.

For datagram sockets, this call returns the entire datagram that was sent, provided that the datagram fits into the specified buffer. Stream sockets act like streams of information with no boundaries separating data. For example, if applications A and B are connected with a stream socket and application A sends 1000 bytes, each call to this function can return 1 byte, or 10 bytes, or the entire 1000 bytes. Therefore, applications using stream sockets should place this call in a loop, calling this function until all data has been received.

Special behavior for C++: To use this function with C++, you must use the _XOPEN_SOURCE_EXTENDED 1 feature test macro.

Returned value

If successful, recv() returns the length of the message or datagram in bytes. The value 0 indicates the connection is closed.

If unsuccessful, recv() returns -1 and sets errno to one of the following values:

Error Code
Description

EBADF
socket is not a valid socket descriptor.

ECONNRESET
A connection was forcibly closed by a peer.

EFAULT
Using the *buf* and *len* parameters would result in an attempt to access storage outside the caller's address space.

EINTR
The recv() call was interrupted by a signal that was caught before any data was available.

EINVAL
The request is invalid or not supported. The MSG_OOB flag is set and no out-of-band data is available.

EIO There has been a network or transport failure.

ENOBUFS
Insufficient system resources are available to complete the call.

ENOTCONN
A receive is attempted on a connection-oriented socket that is not connected.

ENOTSOCK
The descriptor is for a file, not for a socket.

EOPNOTSUPP
The specified flags are not supported for this socket type or protocol.

ETIMEDOUT
The connection timed out during connection establishment, or due to a transmission timeout on active connection.

EWOULDBLOCK
socket is in nonblocking mode and data is not available to read. or the SO_RCVTIMEO timeout value was been reached before data was available.

Related information

- "sys/socket.h" on page 74
- "connect() — Connect a socket" on page 308
- "fcntl() — Control open file descriptors" on page 474
- "getsockopt() — Get the options associated with a socket" on page 777
- "ioctl() — Control device" on page 869
- "read() — Read from a file or socket" on page 1376
- "readv() — Read data on a file or socket and store in a set of buffers" on page 1389
- "recvfrom() — Receive messages on a socket"
- "recvmsg() — Receive messages on a socket and store in an array of message headers" on page 1405
- "select(), pselect() — Monitor activity on files or sockets and message queues" on page 1470
- "selectex() — Monitor activity on files or sockets and message queues" on page 1479
- "send() — Send data on a socket" on page 1494
- "sendmsg() — Send messages on a socket" on page 1500
- "sendto() — Send data on a socket" on page 1506
- "setsockopt() — Set options associated with a socket" on page 1578
- "socket() — Create a socket" on page 1688
- "write() — Write data on a file or socket" on page 2085
- "writev() — Write data on a file or socket from an array" on page 2092

recvfrom() — Receive messages on a socket

Standards

Standards / Extensions	C or C++	Dependencies
XPG4.2 Single UNIX Specification, Version 3	both	

Format

X/Open

```
#define _XOPEN_SOURCE_EXTENDED 1
#include <sys/socket.h>

int recvfrom(int socket, void *__restrict__ buffer,
             size_t length, int flags,
             struct sockaddr *__restrict__ address,
             socklen_t *__restrict__ address_length);
```

Berkeley sockets

```
#define _OE_SOCKETS
#include <sys/socket.h>

int recvfrom(int socket, char *buffer,
             int length, int flags,
             struct sockaddr *address,
             int *address_length);
```

General description

The recvfrom() function receives data on a socket named by descriptor *socket* and stores it in a buffer. The recvfrom() function applies to any datagram socket, whether connected or unconnected.

Parameter
Description

socket The socket descriptor.

buffer The pointer to the buffer that receives the data.

length The length in bytes of the buffer pointed to by the *buffer* parameter. If the MSG_CONNTERM flag is set, the length of the buffer must be zero.

flags A parameter that can be set to 0, MSG_CONNTERM, MSG_PEEK, MSG_OOB, or MSG_WAITALL. The MSG_CONNTERM flag is mutually exclusive with other flags.

MSG_CONNTERM
Requests that the function completes only when a TCP connection is terminated. It is valid for TCP sockets only. Other normal receive requests are also completed. The application must be able to deal with the fact that a normal receive and this special connection termination receive might be driven in parallel.

AT-TLS considerations: If AT-TLS is being used to provide transparent TLS/SSL support for a TCP socket and a receive request with MSG_CONNTERM is outstanding, AT-TLS will immediately honor any TLS/SSL close notify alerts sent by the peer and initiate TLS/SSL session shutdown. For more information about AT-TLS and determining whether a TCP connection is using AT-TLS, see *z/OS Communications Server: IP Programmer's Guide and Reference*.

MSG_OOB
Reads any out-of-band data on the socket. Out-of-band data is sent when the MSG_OOB flag is on for a send(), sendto(), or sendmsg().

The fcntl() command should be used with F_SETOWN to specify the recipient, either a pid or a gid, of a SIGURG signal that will be sent when out-of-band data is sent. If no recipient is set, no signal will be sent. For more information, see the fcntl() command. The recipient of the data determines whether to receive out-of-band data inline or not inline by the setting of the SO_OOBINLINE option of setsockopt(). If SO_OOBINLINE is set off and the MSG_OOB flag is set on, the out-of-band data byte will be read out-of-line. It is invalid for the MSG_OOB flag to be set on when SO_OOBINLINE is set on. If there is out-of-band data available, and the MSG_OOB flag is not set (SO_OOBINLINE can be on or off), then the data up to, but not including, the out-of-band data will be read. When the read cursor has reached the out-of-band data byte, then only the out-of-band data will be read on the next read. The SIOCATMARK option of ioctl() can be used to determine if the read cursor is currently at the out-of-band data byte. For more information, refer to the setsockopt() and ioctl() commands.

MSG_PEEK
Peeks at the data present on the socket; the data is returned but not consumed, so that a subsequent receive operation sees the same data.

MSG_WAITALL
Requests that the function block until the full amount of data requested can be returned. The function may return a smaller amount of data if a signal is caught, the connection is terminated, an error is pending or SO_RCVTIMEO is set and the timer is expired for the socket. For AF_UNIX, the function may also return earlier if out-of-band (OOB) data is inline and there is OOB data to be read. In this case, the data up to the OOB data is returned on the first recvfrom(). The OOB data is returned on the subsequent read request.

address A pointer to a socket address structure from which data is received. If *address* is nonzero, the source address is returned.

address_length
Must initially point to an integer that contains the size in bytes of the storage pointed to by *address*. On return, that integer contains the size required to represent the address of the connecting socket. If this value is larger than the size supplied on input, then the information contained in sockaddr is truncated to the length supplied on input. If *address* is NULL, *address_length* is ignored.

If *address* is nonzero the source address of the message is filled. *address_length* must first be initialized to the size of the buffer associated with *address* and is then modified on return to indicate the actual size of the address stored there.

If either *address* or *address_length* is a NULL pointer, then *address* and *address_length* are unchanged.

If *address* is nonzero, the source address of the message is filled. *address_length* must first be initialized to the size of the buffer associated with *address*, and is then modified on return to indicate the actual size of the address stored there.

This call returns the length of the incoming message or data. If a datagram packet is too long to fit in the supplied buffer, datagram sockets discard excess bytes. If data is not available for the socket *socket*, and *socket* is in blocking mode, the recvfrom() call blocks the caller until data arrives. If data is not available and *socket* is in nonblocking mode, recvfrom() returns a -1 and sets the error code to EWOULDBLOCK. See "fcntl() — Control open file descriptors" on page 474 or "ioctl() — Control device" on page 869 for a description of how to set nonblocking mode.

For datagram sockets, this call returns the entire datagram that was sent, provided that the datagram fits into the specified buffer. Stream sockets act like streams of information with no boundaries separating data. For example, if applications A and B are connected with a stream socket and application A sends 1000 bytes, each call to this function can return 1 byte, or 10 bytes, or the entire 1000 bytes. Therefore, applications using stream sockets should place this call in a loop, calling this function until all data has been received.

Socket address structure for IPv6: For an AF_INET6 socket, the address is returned in a sockaddr_in6 address structure. The sockaddr_in6 structure is defined in the header file **netinet/in.h**.

Special behavior for C++: To use this function with C++, you must use the _XOPEN_SOURCE_EXTENDED 1 feature test macro.

Note: The recvfrom() function has a dependency on the level of the Enhanced ASCII Extensions. See "Enhanced ASCII support" on page 2113 for details.

Returned value

If successful, recvfrom() returns the length of the message or datagram in bytes. The value 0 indicates the connection is closed.

If unsuccessful, recvfrom() returns -1 and sets errno to one of the following values:

Error Code
Description

EBADF
socket is not a valid socket descriptor.

ECONNRESET
The connection was forcibly closed by a peer.

EFAULT
Using the *buffer* and *length* parameters would result in an attempt to access storage outside the caller's address space.

EINTR
A signal interrupted recvfrom() before any data was available.

EINVAL
The request is invalid or not supported. The MSG_OOB flag is set and no out-of-band data is available.

EIO There has been a network or transport failure.

ENOBUFS
Insufficient system resources are available to complete the call.

ENOTCONN
A receive is attempted on a connection-oriented socket that is not connected.

ENOTSOCK
The descriptor is for a file, not for a socket.

EOPNOTSUPP
The specified flags are not supported for this socket type.

ETIMEDOUT
The connection timed out during connection establishment, or due to a transmission timeout on active connection.

EWOULDBLOCK
socket is in nonblocking mode and data is not available to read. or the SO_RCVTIMEO timeout value was been reached before data was available.

Related information

- “sys/socket.h” on page 74
- “fcntl() — Control open file descriptors” on page 474
- “getsockopt() — Get the options associated with a socket” on page 777
- “ioctl() — Control device” on page 869
- “read() — Read from a file or socket” on page 1376
- “readv() — Read data on a file or socket and store in a set of buffers” on page 1389
- “recv() — Receive data on a socket” on page 1398
- “recvmsg() — Receive messages on a socket and store in an array of message headers”
- “select(), pselect() — Monitor activity on files or sockets and message queues” on page 1470
- “selectex() — Monitor activity on files or sockets and message queues” on page 1479
- “send() — Send data on a socket” on page 1494
- “sendmsg() — Send messages on a socket” on page 1500
- “sendto() — Send data on a socket” on page 1506
- “setsockopt() — Set options associated with a socket” on page 1578
- “socket() — Create a socket” on page 1688
- “write() — Write data on a file or socket” on page 2085
- “writev() — Write data on a file or socket from an array” on page 2092

recvmsg() — Receive messages on a socket and store in an array of message headers

Standards

Standards / Extensions	C or C++	Dependencies
XPG4.2 Single UNIX Specification, Version 3	both	

Format

X/Open

```
#define _XOPEN_SOURCE_EXTENDED 1
#include <sys/socket.h>

ssize_t recvmsg(int socket, struct msghdr *msg, int flags);
```

Berkeley sockets

```
#define _OE_SOCKETS
#include <sys/socket.h>

int recvmsg(int socket, struct msghdr *msg, int flags);
```

General description

The recvmsg() function receives messages on a socket with descriptor *socket* and stores them in an array of message headers.

Parameter
Description

socket The socket descriptor.

msg An array of message headers into which messages are received.

flags The *flags* parameter is set by specifying one or more of the following flags. If more than one flag is specified, the logical OR operator (|) must be used to separate them. The MSG_CONNTERM flag is mutually exclusive with other flags.

MSG_CONNTERM
Requests that the function completes only when a TCP connection is terminated. It is valid for TCP sockets only. Other normal receive requests are also completed. The application must be able to deal with the fact that a normal receive and this special connection termination receive might be driven in parallel.

AT-TLS considerations: If AT-TLS is being used to provide transparent TLS/SSL support for a TCP socket and a receive request with MSG_CONNTERM is outstanding, AT-TLS will immediately honor any TLS/SSL close notify alerts sent by the peer and initiate TLS/SSL session shutdown. For more information about AT-TLS and determining whether a TCP connection is using AT-TLS, see *z/OS Communications Server: IP Programmer's Guide and Reference*.

MSG_OOB
Reads any out-of-band data on the socket. Out-of-band data is sent when the MSG_OOB flag is on for a send(), sendto() or sendmsg().

The fcntl command should be used with F_SETOWN to specify the recipient, either a pid or a gid, of a SIGURG signal that will be sent when out-of-band data is sent. If no recipient is set, no signal will be sent. For more information, see the fcntl() command. The recipient of the data determines whether to receive out-of-band data inline or not inline by the setting of the SO_OOBINLINE option of setsockopt(). If SO_OOBINLINE is set off and the MSG_OOB flag is set on, the out-of-band data byte will be read out-of-line. It is invalid for the MSG_OOB flag to be set on when SO_OOBINLINE is set on. If there is out-of-band data available, and the MSG_OOB flag is not set (SO_OOBINLINE can be on or off), then the data up to, but not including, the out-of-band data will be read. When the read cursor has reached the out-of-band data byte, then only the out-of-band data will be read on the next read, and the output MSG_OOB msg_flag in the message header will be set on. The SIOCATMARK option of ioctl() can be used to determine if the read cursor is currently at the out-of-band data byte. For more information, refer to the setsockopt() and ioctl() commands.

MSG_PEEK
Peeks at the data present on the socket; the data is returned but not consumed, so that a subsequent receive operation will see the same data.

MSG_WAITALL
Requests that the function block until the full amount of data requested can be returned. The function may return a smaller amount of data if a signal is caught, the connection is terminated, an error is pending or SO_RCVTIMEO is set and the timer is expired for the socket.

A message header is defined by a msghdr structure. A definition of this structure can be found in the sys/socket.h include file and contains the following elements:

Element
Description

msg_iov
An array of *iovec* buffers into which the message is placed.

msg_iovlen
The number of elements in the *msg_iov* array. If the MSG_CONNTERM flag is set, the number of elements must be zero.

msg_name
An optional pointer to a buffer where the sender's address is stored.

msg_namelen
The size of the address buffer.

caddr_t msg_accrights
Access rights sent/received (ignored if specified by the user).

int msg_accrightslen
Length of access rights data (ignored if specified by the user).

msg_control
Ancillary data, see below.

msg_controllen
Ancillary data buffer length.

msg_flags
Flags on received message.

Ancillary data consists of a sequence of pairs, each consisting of a cmsghdr structure followed by a data array. The data array contains the ancillary data message, and the cmsghdr structure contains descriptive information that allows an application to correctly parse the data.

The sys/socket.h header file defines the cmsghdr structure that includes at least the following members:

Element
Description

cmsg_len
Data byte count, including header.

cmsg_level
Originating protocol.

cmsg_type
Protocol-specific type.

The following ancillary data are available at the IPv4 level:

Ancillary data
Description

IP_PKTINFO
(RAW and UDP) Returns the source IP address for an outgoing packet and the outgoing interface. The data is passed in an in_pktinfo structure as defined in netinet/in.h.

The following ancillary data are available at the IPv6 level:

Ancillary data
: **Description**

IPV6_HOPLIMIT
: (RAW, TCP, and UDP) Returns the maximum hop limit for an incoming packet. The data is passed in a structure as defined in netinet/in.h.

IPV6_PATHMTU
: (RAW and UDP) Returns the path MTU value for the source of a connected socket. The data is passed in a structure as defined in netinet/in.h.

IPV6_PKTINFO
: (RAW and UDP) Returns the source IP address for an outgoing packet and the outgoing interface. The data is passed in an in6_pktinfo structure as defined in netinet/in.h.

The following ancillary data are available at the socket level:

Ancillary data
: **Description**

SCM_RIGHTS
: Returns the data array that contains the access rights to be sent or received. This ancillary data is valid only for the AF_UNIX domain. The structure is defined in sys/socket.h.

The sys/socket.h header file defines the following macros to gain access to the data arrays in the ancillary data associated with a message header:

CMSG_DATA*(cmsg)*
: If the argument is a pointer to a cmsghdr structure, this macro returns an unsigned character pointer to the data array associated with the cmsghdr structure.

CMSG_NXTHDR*(mhdr,cmsg)*
: If the first argument is a pointer to a msghdr structure and the second argument is a pointer to a cmsghdr structure in the ancillary data, pointed to by the msg_control field of that msghdr structure, this macro returns a pointer to the next cmsghdr structure, or a NULL pointer if this structure is the last cmsghdr in the ancillary data.

CMSG_FIRSTHDR*(mhdr)*
: If the argument is a pointer to a msghdr structure, this macro returns a pointer to the first cmsghdr structure in the ancillary data associated with this msghdr structure, or a NULL pointer if there is no ancillary data associated with the msghdr structure.

The recvmsg() function applies to sockets, regardless of whether they are in the connected state.

This call returns the length of the data received. If data is not available for the socket *socket*, and *socket* is in blocking mode, the recvmsg() call blocks the caller until data arrives. If data is not available and *socket* is in nonblocking mode, recvmsg() returns a -1 and sets the error code to EWOULDBLOCK. See “fcntl() — Control open file descriptors” on page 474 or “ioctl() — Control device” on page 869 for a description of how to set nonblocking mode.

For datagram sockets, this call returns the entire datagram that was sent, provided that the datagram fits into the specified buffer. Stream sockets act like streams of information with no boundaries separating data. For example, if applications A and B are connected with a stream socket and application A sends 1000 bytes, each call to this function can return 1 byte, or 10 bytes, or the entire 1000 bytes. Therefore, applications using stream sockets should place this call in a loop, calling this function until all data has been received.

On successful completion, the msg_flags member for the message header is the bitwise inclusive-OR of all of the following flags that indicate conditions detected for the received message:

MSG_OOB
Out-of-band data was received.

MSG_TRUNC
Normal data was truncated.

MSG_CTRUNC
Control data was truncated.

Socket address structure for IPv6: For an AF_INET6 socket, the address is returned in a sockaddr_in6 address structure. The sockaddr_in6 structure is defined in the header file netinet/in.h.

Special behavior for C++: To use this function with C++, you must use the _XOPEN_SOURCE_EXTENDED 1 feature test macro.

Note: The recvmsg() function has a dependency on the level of the Enhanced ASCII Extensions. See "Enhanced ASCII support" on page 2113 for details.

Returned value

If successful, recvmsg() returns the length of the message in bytes. The value 0 indicates the connection is closed.

If unsuccessful, recvmsg() returns -1 and sets errno to one of the following values:

Error Code
Description

EBADF
socket is not a valid socket descriptor.

ECONNRESET
The connection was forcibly closed by a peer.

EFAULT
Using *msg* would result in an attempt to access storage outside the caller's address space.

EINTR
The function was interrupted by a signal before any data was available.

EINVAL
The request is invalid or not supported. The sum of the **iov_len** values overflows a **ssize_t**.

EIO There has been a network or transport failure.

ENOBUFS
Insufficient system resources are available to complete the call.

ENOTCONN
A receive is attempted on a connection-oriented socket that is not connected.

ENOTSOCK
The descriptor is for a file, not for a socket.

EOPNOTSUPP
The specified flags are not supported for this socket type.

ETIMEDOUT
The connection timed out during connection establishment, or due to a transmission timeout on active connection.

EWOULDBLOCK
socket is in nonblocking mode and data is not available to read. or the SO_RCVTIMEO timeout value was been reached before data was available.

Related information

- "sys/socket.h" on page 74
- "connect() — Connect a socket" on page 308
- "fcntl() — Control open file descriptors" on page 474
- "getsockopt() — Get the options associated with a socket" on page 777
- "ioctl() — Control device" on page 869
- "read() — Read from a file or socket" on page 1376
- "readv() — Read data on a file or socket and store in a set of buffers" on page 1389
- "recv() — Receive data on a socket" on page 1398
- "recvfrom() — Receive messages on a socket" on page 1401
- "select(), pselect() — Monitor activity on files or sockets and message queues" on page 1470
- "selectex() — Monitor activity on files or sockets and message queues" on page 1479
- "send() — Send data on a socket" on page 1494
- "sendmsg() — Send messages on a socket" on page 1500
- "sendto() — Send data on a socket" on page 1506
- "setsockopt() — Set options associated with a socket" on page 1578
- "socket() — Create a socket" on page 1688
- "write() — Write data on a file or socket" on page 2085
- "writev() — Write data on a file or socket from an array" on page 2092

re_exec() — Match regular expression

Standards

Standards / Extensions	C or C++	Dependencies
XPG4.2	both	

Format

```
#define _XOPEN_SOURCE_EXTENDED 1
#include <re_comp.h>

int re_exec(const char *string);
```

General description

Restriction: This function is not supported in AMODE 64.

The re_exec() function attempts to match the string pointed to by the *string* argument with the last regular expression passed to re_comp().

The parameter *string* is a pointer to a character string to be compared.

Strings passed to re_exec() must be terminated by a NULL byte, and may include newline characters.

Notes:

1. The re_comp() and re_exec() functions are provided for historical reasons. These functions were part of the Legacy Feature in Single UNIX Specification, Version 2. They have been withdrawn and are not supported as part of Single UNIX Specification, Version 3. New applications should use the newer functions fnmatch(), glob(), regcomp() and regexec(), which provide full internationalized regular expression functionality compatible with IEEE Std 1003.1-2001.
2. The z/OS UNIX implementation of the re_exec() function supports only the POSIX locale. Any other locales will yield unpredictable results.
3. The re_comp() and re_exec() functions are supported on the thread-level. They must be issued from the *same* thread to work properly.

The re_exec() function supports simple regular expressions, which are defined in "re_comp() — Compile regular expression" on page 1395.

Returned value

If successful, re_exec() returns 1 if the input *string* matches the last compiled regular expression.

If unsuccessful, re_exec() returns 0 if the input *string* fails to match the last compiled regular expression, and -1 if the compiled regular expression is invalid (indicating an internal error).

Related information

- "re_comp.h" on page 61
- "fnmatch() — Match file name or path name" on page 564
- "glob() — Generate path names matching a pattern" on page 807
- "re_comp() — Compile regular expression" on page 1395
- "regcomp() — Compile regular expression" on page 1415
- "regexec() — Execute compiled regular expression" on page 1421

regcmp() — Compile regular expression

Standards

Standards / Extensions	C or C++	Dependencies
XPG4.2	C only	

Format

```
#define _XOPEN_SOURCE_EXTENDED  1
#include <libgen.h>

char *regcmp(const char *pattern[,...], (char *)0);

char *regex(const char *cmppat, const char *subject[,subexp,...]);

extern char *__loc1;
```

General description

Restriction: This function is not supported in AMODE 64.

The regcmp() function concatenates regular expression (RE) patterns specified by a list of one or more *pattern* arguments. The end of this list must be delimited by a NULL pointer. The regcmp() function then converts the concatenated RE pattern into an internal form suitable for use by the pattern matching regex() function. If conversion is successful, regcmp() returns a pointer to the converted pattern. Otherwise, it returns a NULL pointer. The regcmp() function uses malloc() to obtain storage for the converted pattern. It is the application's responsibility to free unneeded space so allocated.

The regex() function executes a converted pattern *cmppat* against a *subject* string. If *cmppat* matches all or part of the *subject* string, the regex() function returns a pointer to the next unmatched character in the *subject* string and sets the external variable `__loc1` to point the first matched character in the *subject* string. If no match is found between *cmppat* and the *subject* string, the regex() function returns a NULL pointer.

The regcmp() and regex() functions are supported in any locale. However, results are unpredictable if they are not run in the same locale.

Following are valid RE symbols and their meaning to the regcmp() and regex() functions:

Expression
Meaning

NUL Terminate RE pattern and text string

c Any non-special character, c, is a one-character RE which matches itself.

\s A backslash (\) followed by a special character, s, is a one-character RE which matches the special character itself.

The following characters are special:
period, ., asterisk, *, plus, +, dollar, $, left square bracket, [, left brace, {, right brace, }, left parenthesis, (, right parenthesis,), and backslash, \, are always special except when they appear within square brackets ([]). caret (^) is special at the beginning of an entire RE (which is another name for a pattern).

Note: An non-special character preceded by \ is a one-character RE which matches the non-special character.

yz Concatenation of REs y and z matches concatenation of strings matched by y and z.

. The period (.) special character RE matches any single character except the <newline> character.

^ The caret (^) at the beginning of an entire RE is an RE which matches the beginning of a string. Thus, it **anchors** or limits matches by the entire RE to the beginning of strings.

$** The dollar ($) at the end of an entire RE is an RE which only the end of a string (delimited by the <NUL> character). Thus, it **anchors or limits matches by the entire RE to the end of strings.

Note: \n (the C language designation for a <newline> character) must be used in an entire RE to match any embedded or trailing <newline> character in a text string.

(...) Parentheses are used to delimit a sub-expression which matches whatever the REs comprising the sub-expression would have matched without the delimiting parentheses.

(...)$n $n, where n is a digit between 0 and 9, inclusive, may be used to tag a sub-expression. The tag tells the regex() function to return the substring matched by the sub-expression at address specified by (n+1)th argument after *subject*.

***** A one-character RE or sub-expression followed by an asterisk (*) is a RE that matches zero or more occurrences of the one-character RE or sub-expression. If there is any choice, the longest leftmost string that permits a match is chosen.

+ A one-character RE or sub-expression followed by a plus (+) is a RE that matches one or more occurrences of the one-character RE or sub-expression. Whenever a choice exists, the RE matches as many occurrences as possible.

{m,n} A one-character RE or sub-expression followed by integer values, m and n, enclosed in braces is a RE which matches repeated occurrences of whatever the preceding one-character RE or sub-expression matched. The value of m, which must be in the range 0 to 255, inclusive, is the minimum number of occurrences required for a match. The value of n which, if specified, must also must be in the range 0 to 255, inclusive, is the maximum. The value of n, if specified, must be greater than or equal to the value m. The following brace expressions are valid:

- **{m}** Matches exactly m occurrences of the preceding one-character RE or sub-expression.
- **{m,}** Matches m or more occurrences of the preceding one-character RE or sub-expression. There is no limit on the number of occurrences which will be matched. The plus (+) and asterisk (*) operations are equivalent to {1,} and {0,}, respectively.
- **{m,n}** Matches between m and n occurrences, inclusive.

Whenever a choice exists, the RE matches as many occurrence as possible.

[...] A non-empty list of characters enclosed by square brackets is a one-character RE that matches any one character in the list.

[^...] A non-empty list of characters preceded by a caret (^) enclosed by square brackets is a one-character RE that matches any character except <newline> and the characters in the list. The ^ has special meaning only if it is the first character after the left bracket ([).

[c1-c2] The hyphen (-) between two characters c1 and c2 within square brackets designates the list of characters whose collating values fall between the collating values of c1 and c2 in the current locale. The collating value of c2 must be greater than or equal to c1. Also, c2 may not be used as the ending point of one range and the starting point of another range. In other words, c1-c2-c3 is invalid.

The - loses special meaning if it occurs first or last in the bracket expression or if it is used for c1 or c2.

The right bracket,], does not terminate a bracket expression when it is the first character within it (after an initial ^, if any). For example, the expression []0-9] matches a right bracket or a digit in the range 0-9, inclusive.

Notes:

1. Multiple duplication symbols applied to the same RE will be interpreted in the following order of precedence:
   a. *
   b. +
   c. {}
2. RE Order of precedence is as follows, from high to low:
   a. escaped character \character
   b. bracket expression [...]
   c. sub-expression (...)
   d. duplication * + {}
   e. concatenation yz
   f. anchors ^ $

Note:

The regcmp() and regex() functions are provided for historical reasons. These functions were part of the Legacy Feature in Single UNIX Specification, Version 2. They have been withdrawn and are not supported as part of Single UNIX Specification, Version 3. New applications should use the newer functions fnmatch(), glob(), regcomp() and regexec(), which provide full internationalized regular expression functionality compatible with IEEE Std 1003.1-2001.

If it is necessary to continue using these functions in an application written for Single UNIX Specification, Version 3, define the feature test macro _UNIX03_WITHDRAWN before including any standard system headers. The macro exposes all interfaces and symbols removed in Single UNIX Specification, Version 3.

Returned value

If the pattern formed by concatenating the list of *pattern* arguments is successfully converted, regcmp() returns a pointer to the converted pattern. Otherwise, it returns a NULL pointer. If regcmp() is unable to allocate storage for the converted pattern, it sets errno to ENOMEM.

If regex() successfully matches the converted pattern *cmppat* to all or part of the *subject* string, it returns a pointer to the next unmatched character in *subject*. Otherwise, it returns a NULL pointer.

Related information

- "libgen.h" on page 38
- "fnmatch() — Match file name or path name" on page 564
- "glob() — Generate path names matching a pattern" on page 807
- "re_comp() — Compile regular expression" on page 1395
- "re_exec() — Match regular expression" on page 1410
- "regcomp() — Compile regular expression"
- "regexec() — Execute compiled regular expression" on page 1421

regcomp() — Compile regular expression

Standards

Standards / Extensions	C or C++	Dependencies
XPG4 XPG4.2 Single UNIX Specification, Version 3 z/OS UNIX	both	

Format

```
#include <regex.h>

int regcomp(regex_t *_restrict_ preg, const char *_restrict_ pattern, int cflags);
```

General description

Compiles the regular expression specified by *pattern* into an executable string of op-codes.

preg is a pointer to a compiled regular expression.

pattern is a pointer to a character string defining a source regular expression (described below).

cflags is a bit flag defining configurable attributes of compilation process:

REG_EXTENDED
Support extended regular expressions.

REG_ICASE
Ignore case in match.

REG_NEWLINE
Eliminate any special significance to the newline character.

REG_NOSUB
Report only success or fail in regexec(), that is, verify the syntax of a regular expression. If this flag is set, the regcomp() function sets *re_nsub* to the number of parenthesized sub-expressions found in *pattern*. Otherwise, a sub-expression results in an error.

The regcomp() function under z/OS XL C/C++ will use the definition of characters according to the current LC_SYNTAX category. The characters, [,], {, }, |, ^, and $, have varying code points in different encoded character sets.

Regular expressions

The functions regcomp(), regerror(), regexec(), and regfree() use regular expressions in a similar way to the UNIX awk, ed, grep, and egrep commands.

The simplest form of regular expression is a string of characters with no special meaning. The following characters do have special meaning; they are used to form extended regular expressions:

Symbol
Description

.
The period symbol matches any one character except the terminal newline character.

[*character***–***character***]**
The hyphen symbol, within square brackets, means "through". It fills in the intervening characters according to the current collating sequence. For example, [a–z] can be equivalent to [abc...xyz] or, with a different collating sequence, it can be equivalent to [aAbBcC...xXyYzZ].

[*string***]**
A string within square brackets specifies any of the characters in *string*. Thus `[abc]`, if compared to other strings, would match any that contained a, b, or c.

No assumptions are made at compile time about the actual characters contained in the range.

{*m***} {***m***,} {***m***,***u***}**
Integer values enclosed in {} indicate the number of times to apply the preceding regular expression. *m* is the minimum number, and *u* is the maximum number. *u* must not be greater than RE_DUP_MAX (see "limits.h" on page 39).

If you specify only *m*, it indicates the exact number of times to apply the regular expression. {*m*,} is equivalent to {*m,u*}. They both match m or more occurrences of the expression.

*****
The asterisk symbol indicates 0 or more of any characters. For example, `[a*e]` is equivalent to any of the following: 99ae9, aaaaae, a999e99.

$
The dollar symbol matches the end of the string. (Use `\n` to match a newline character.)

*character***+**
The plus symbol specifies one or more occurrences of a character. Thus, `smith+ern` is equivalent to, for example, `smithhhern`.

[^*`string`*]
The caret symbol, when inside square brackets, negates the characters within the square brackets. Thus `[^abc]`, if compared to other strings, would *fail* to match any that contains even one a, b, or c.

(*expression***)$***n*
Stores the value matched by the enclosed regular expression in the (n+1)th *ret* parameter. Ten enclosed regular expressions are allowed. Assignments are made unconditionally.

(*expression***)**
Groups a sub-expression allowing an operator, such as *, +, or [].], to work on the sub-expression enclosed in parentheses. For example, `(a*(cb+)*)$0`.

Note:

1. Do *not* use multibyte characters.
2. You can use the] (right square bracket) alone within a pair of square brackets, but only if it immediately follows either the opening left square bracket or if it immediately follows [^. For example: []–] matches the] and – characters.
3. All the preceding symbols are *special*. You precede them with \ to use the symbol itself. For example, `a\.e` is equivalent to `a.e`.
4. You can use the – (hyphen) by itself, but only if it is the first or last character in the expression. For example, the expression []--0] matches either the] or else the characters – through 0. Otherwise, use \–.

Returned value

If successful, regcomp() returns 0.

If unsuccessful, regcomp() returns nonzero, and the content of *preg* is undefined.

Example

CELEBR07

```
/* CELEBR07

   This example compiles an extended regular expression.

 */
#include <regex.h>
#include <locale.h>
#include <stdio.h>
#include <stdlib.h>

main() {
    regex_t    preg;
    char       *string = "a simple string";
    char       *pattern = ".*(simple).*";
    int        rc;

    if ((rc = regcomp(&preg, pattern, REG_EXTENDED)) != 0) {
       printf("regcomp() failed, returning nonzero (%d)", rc);
       exit(1);
    }
}
```

Related information

- “regex.h” on page 61
- “regerror() — Return error message”
- “regexec() — Execute compiled regular expression” on page 1421
- “regfree() — Free memory for regular expression” on page 1423

regerror() — Return error message

Standards

Standards / Extensions	C or C++	Dependencies
XPG4 XPG4.2 Single UNIX Specification, Version 3	both	

Format

```
#include <regex.h>

size_t regerror(int errcode, const regex_t *_restrict_preg,
                char *_restrict_errbuf, size_t errbuf_size);
```

General description

Finds the description for *errcode*. (For a description of regular expressions, see "Regular expressions" on page 1416.)

Returned value

regerror() returns the integer value that is the size of the buffer needed to hold the generated description string for the error condition corresponding to *errcode*.

regerror() returns the following messages.

errcode
    Description String

REG_BADBR
    Invalid \{ \} range exp

REG_BADPAT
    Invalid regular expression

REG_BADRPT
    ?*+ not preceded by valid RE

REG_EBOL
    ¬ anchor and not BOL

REG_EBRACE
    \{ \} or { } imbalance

REG_EBRACK
    [] imbalance

REG_ECHAR
    Invalid multibyte character

REG_ECOLLATE
    Invalid collating element

REG_ECTYPE
    Invalid character class

REG_EEOL
    $ anchor and not EOL

REG_EESCAPE
    Last character is \

REG_EPAREN
    \(\) or () imbalance

REG_ERANGE
    Invalid range exp endpoint

REG_ESPACE
    Out of memory

REG_ESUBREG
Invalid number in \digit

REG_NOMATCH
RE pattern not found

The LC_SYNTAX characters in the messages will be converted to the code points from the current LC_SYNTAX category.

Example

CELEBR08

```
/* CELEBR08

   This example compiles an invalid regular expression, and
   print error message &regerror..

 */
#include <regex.h>
#include <locale.h>
#include <stdio.h>
#include <stdlib.h>

main() {
    regex_t    preg;
    char       *pattern = "a[missing.bracket";
    int        rc;
    char       buffer[100];

    if ((rc = regcomp(&preg, pattern, REG_EXTENDED)) != 0) {
       regerror(rc, &preg, buffer, 100);
       printf("regcomp() failed with '%s'\n", buffer);
       exit(1);
    }
}
```

Related information

- See the topics about internationalization in *z/OS XL C/C++ Programming Guide*.
- “regex.h” on page 61
- “regcomp() — Compile regular expression” on page 1415
- “regexec() — Execute compiled regular expression” on page 1421
- “regfree() — Free memory for regular expression” on page 1423

regex() — Execute compiled regular expression

Standards

Standards / Extensions	C or C++	Dependencies
XPG4.2	C only	

Format

```
#define _XOPEN_SOURCE_EXTENDED  1
#include <libgen.h>

char *regex(const char *cmppat, const char *subject[,subexp,...]);

extern char *__loc1;
```

General description

Restriction: This function is not supported in AMODE 64.

The regex() function executes a converted pattern *cmppat* produced by the regcomp() function against a *subject* string. If *cmppat* matches all or part of the *subject* string, the regex() function returns a pointer to the next unmatched character in the *subject* string and sets the external variable `__loc1` to point the first matched character in the *subject* string. If no match is found between *cmppat* and the *subject* string, the regex() function returns a NULL pointer.

The regex() and regcomp() functions are supported in any locale. However, results are unpredictable if they are not run in the same locale.

Refer to “regcmp() — Compile regular expression” on page 1411 for a description of regular expression syntax and semantics supported by the regex() and regcomp() functions.

Note:

The regcmp() and regex() functions are provided for historical reasons. These functions were part of the Legacy Feature in Single UNIX Specification, Version 2. They have been withdrawn and are not supported as part of Single UNIX Specification, Version 3. New applications should use the newer functions fnmatch(), glob(), regcomp() and regexec(), which provide full internationalized regular expression functionality compatible with IEEE Std 1003.1-2001.

If it is necessary to continue using these functions in an application written for Single UNIX Specification, Version 3, define the feature test macro _UNIX03_WITHDRAWN before including any standard system headers. The macro exposes all interfaces and symbols removed in Single UNIX Specification, Version 3.

Returned value

If regex() successfully matches the converted pattern *cmppat* to all or part of the *subject* string, it returns a pointer to the next unmatched character in *subject*.

If unsuccessful, regex() returns a NULL pointer.

Related information

- “libgen.h” on page 38
- “fnmatch() — Match file name or path name” on page 564
- “glob() — Generate path names matching a pattern” on page 807
- “re_comp() — Compile regular expression” on page 1395
- “regcomp() — Compile regular expression” on page 1415
- “re_exec() — Match regular expression” on page 1410
- “regexec() — Execute compiled regular expression” on page 1421

regexec() — Execute compiled regular expression

Standards

Standards / Extensions	C or C++	Dependencies
XPG4 XPG4.2	both	

Format

```
#include <regex.h>

int regexec(const regex_t *preg, const char *string,
            size_t nmatch, regmatch_t *pmatch, int eflags);
```

XPG4

```
#define _XOPEN_SOURCE
#include <regex.h>

int regexec(const regex_t *__ preg,
            const char *__ string,
         size_t nmatch, regmatch_t *__ pmatch, int eflags);
```

General description

Compares the NULL-terminated string specified by *string* against the compiled regular expression, *preg*. (For a description of regular expressions, see "Regular expressions" on page 1416.)

preg is a pointer to a compiled regular expression to compare against STRING.

string is a pointer to a string to be matched.

nmatch is the number of sub-expressions to match.

pmatch is an array of offsets into STRING which matched the corresponding sub-expressions in *preg*.

eflags is a bit flag defining customizable behavior of regexec().

REG_NOTBOL
Indicates that the first character of STRING is not the beginning of the line.

REG_NOTEOL
Indicates that the first character of STRING is not the end of the line.

If *nmatch* parameter is 0 or REG_NOSUB was set on the call to regcomp(), regexec() ignores the *pmatch* argument. Otherwise, the *pmatch* argument points to an array of at least *nmatch* elements. The regexec() function fills in the elements of the array with offsets of the substrings of STRING that correspond to the parenthesized sub-expressions of the original *pmatch* specified to regcomp(). The 0th element of the array corresponds to the entire pattern. If there are more than *nmatch* sub-expressions, only the first *nmatch*-1 are recorded.

When matching a basic or extended regular expression, any given parenthesized sub-expression of *pmatch* might participate in the match of several different substrings of STRING. The following rules determine which substrings are reported in *pmatch*.

1. If a sub-expression participated in a match several times, the offset of the last matching substring is reported in *pmatch*.
2. If a sub-expression did not match in the source STRING, the offset shown in *pmatch* is set to -1.
3. If a sub-expression contains sub-expressions, the data in *pmatch* refers to the last such sub-expression.
4. If a sub-expression matches a zero-length string, the offsets in *pmatch* refer to the byte immediately following the matching string.

If EREG_NOSUB was set when regcomp() was called, the contents of *pmatch* are unspecified.

If REG_NEWLINE was set when regcomp() was called, newline characters are allowed in STRING.

Notes:

1. With z/OS XL C/C++, the string passed to the regexec() function is assumed to be in the initial shift state, unless REG_NOTBOL is specified. If REG_NOTBOL is specified, the shift state used is the shift state after the last call to the regexec() function.
2. The information returned by the regexec() function in the `regmatch_t` structure has the shift-state at the start and end of the string added. This will assist an application to perform replacements or processing of the partial string. To perform replacements, the application must add the required shift-out and shift-in characters where necessary. No library functions are available to assist the application.
3. If MB_CUR_MAX is specified as 4, but the charmap file does not specify the DBCS characters, and a collating-element (for example, `[:a:]`) is specified in the pattern, the DBCS characters will not match against the collating-element even if they have an equivalent weight to the collating-element.

Returned value

If a match is found, regexec() returns 0.

If unsuccessful, regexec() returns nonzero indicating either no match or an error.

Example

CELEBR09

```
/* CELEBR09

   This example compiles an extended regular expression, and
   match against a string.

 */
#include <regex.h>
#include <locale.h>
#include <stdio.h>
#include <stdlib.h>

main() {
    regex_t    preg;
    char       *string = "a simple string";
    char       *pattern = ".*(simple).*";
    int        rc;
    size_t     nmatch = 2;
    regmatch_t pmatch[2];

    if ((rc = regcomp(&preg, pattern, REG_EXTENDED)) != 0) {
       printf("regcomp() failed, returning nonzero (%d)\n", rc);
       exit(1);
    }

    if ((rc = regexec(&preg, string, nmatch, pmatch, 0)) != 0) {
       printf("failed to ERE match '%s' with '%s',returning %d.\n",
       string, pattern, rc);
    }

    regfree(&preg);
}
```

Related information

- See the topic about internationalization in *z/OS XL C/C++ Programming Guide*
- “regex.h” on page 61
- “regcomp() — Compile regular expression” on page 1415
- “regerror() — Return error message” on page 1417
- “regfree() — Free memory for regular expression”

regfree() — Free memory for regular expression

Standards

Standards / Extensions	C or C++	Dependencies
XPG4 XPG4.2 Single UNIX Specification, Version 3	both	

Format

```
#include <regex.h>

void regfree(regex_t *preg);
```

General description

Frees any memory that was allocated by regcomp() to implement *preg*. The expression defined by *preg* is no longer a compiled regular or extended expression. (For a description of regular expressions, see “Regular expressions” on page 1416.)

Example

CELEBR10

```
/* CELEBR10

   This example compiles an extended regular expression and a
   free regular expression.

 */
#include <regex.h>
#include <locale.h>
#include <stdio.h>
#include <stdlib.h>

main() {
    regex_t    preg;
    char       *pattern = ".*(simple).*";
    int        rc;
    if ((rc = regcomp(&preg, pattern, REG_EXTENDED)) != 0) {
       printf("regcomp() failed, returning nonzero (%d)\n", rc);
       exit(1);
    }

    regfree(&preg);
}
```

Related information

- Chapter s about internationalization in *z/OS XL C/C++ Programming Guide*
- “regex.h” on page 61
- “regcomp() — Compile regular expression” on page 1415
- “regerror() — Return error message” on page 1417
- “regexec() — Execute compiled regular expression” on page 1421

release() — Delete a load module

Standards

Standards / Extensions	C or C++	Dependencies
z/OS UNIX	C only	

Format

```
#include <stdlib.h>

int release(void(*fetch_ptr)());
```

General description

Removes from memory the load modules retrieved by fetch() or fetch control blocks created by fetchep(). The *fetch_ptr* parameter is obtained from a call to fetch() or fetchep(). Once released, the fetch() and any associated fetchep() pointers are no longer valid.

To avoid infringing on the user's name space, this nonstandard function has two names. One name, the external entry point name is prefixed with two underscore characters, and the other name is not. The name without the prefix underscore characters is exposed only when you use LANGLVL(EXTENDED).

Note: The external entry point name for release() is __rlse(), *NOT* __release().

To use this function, you must either invoke the function using its external entry point name (that is, the name that begins with two underscore characters __rlse(), or compile with LANGLVL(EXTENDED). When you use LANGLVL(EXTENDED) any relevant information in the header is also exposed.

All fetched modules and fetch control blocks created by fetchep() are released automatically on program termination.

Using release() on a module obtained by using fetch() will also cause the release() of any child fetch control blocks created by fetchep() for this module. However, using release() on a child fetch control block will have no effect on the parent modules or sibling fetch control blocks obtained by using fetch(). Trying to use a fetch control block after it has been released will result in undefined behavior. (A Fetch Control Block (FECB) is an internal executable control block. The fetch pointer points to it.

When non-reentrant modules have been fetched multiple times, you should release them in the reverse order; otherwise, the load modules may not be deleted immediately.

Returned value

If successful, release() returns 0.

If unsuccessful, release() returns nonzero.

Example

```
/* The following C example uses the fetch() function to load a module, and
   later uses release() to delete the module from memory.
 */
#include <stdlib.h>

void (*fetch_ptr)();

int main(void) {
   fetch_ptr = fetch("sample");
⋮
   release(fetch_ptr);   /* all modules are released */
}
```

Related information

- "stdlib.h" on page 70
- "fetch() — Get a load module" on page 507
- "fetchep() — Share writable static" on page 519

remainder(), remainderf(), remainderl() — Computes the remainder x REM y

Standards

Standards / Extensions	C or C++	Dependencies
XPG4.2 C99 Single UNIX Specification, Version 3 C++ TR1 C99	both	

Format

```
#define _XOPEN_SOURCE_EXTENDED 1
#include <math.h>

double remainder(double x, double y);
```

C99

```
#define _ISOC99_SOURCE
#include <math.h>

float remainderf(float x, float y);
long double remainderl(long double x, long double y);
```

C++ TR1 C99

```
#define _TR1_C99
#include <math.h>

float remainder(float x, float y);
long double remainder(long double x, long double y);
```

General description

The remainder() function returns the floating-point remainder when *y* is nonzero and following the relation

$$r = x - ny$$

The value *n* is the integral value nearest the exact value *x*/*y* and when

$$| n - x/y | = 1/2$$

then the value of n is even.

Note: The following table shows the viable formats for these functions. See "IEEE binary floating-point" on page 94 for more information about IEEE Binary Floating-Point.

Function	Hex	IEEE
remainder	X	X
remainderf	X	X
remainderl	X	X

Restriction: The remainderf() function does not support the _FP_MODE_VARIABLE feature test macro.

Returned value

If successful, remainder() returns the remainder of the division of *x* by *y* as described.

If *y* is zero, remainder() returns HUGE_VAL and sets errno to EDOM.

If *r* = 0, then its sign will be that of *x*.

Special behavior for IEEE: If successful, remainder() returns the remainder of the division of *x* by *y*.

If *y* is zero, remainder() returns NaNQ and sets errno to EDOM..

Example

```
/*
 * This program illustrates the use of remainder() function
 *
 */
#define _ISOC99_SOURCE
#include <math.h>
#include <stdio.h>
```

```
void main() {

  double number1=3.0, number2=3.5;

  printf("Illustrates the remainder() function");

  #ifdef __BFP__
    printf(" (IEEE version)\n\n");
  #else
    printf(" (HFP version)\n\n");
  #endif

  printf("remainder(%.2f,%.2f)=%.2f\n",number1,number2,
                           remainder(number1,number2));
  number1=1; number2=2;
  printf("remainder(%.2f,%.2f)=%.2f\n",number1,number2,
                           remainder(number1,number2));
  number1=1; number2=0;
  printf("remainder(%.2f,%.2f)=%.2f\n",number1,number2,
                           remainder(number1,number2));
}
```

```
Output

Illustrates the remainder() function (IEEE version)

remainder(3.00,3.50)=-0.50
remainder(1.00,2.00)=1.00
remainder(1.00,0.00)=NaNQ(1)
```

Related information

- “math.h” on page 44
- “abs(), absf(), absl() — Calculate integer absolute value” on page 103

remainderd32(), remainderd64(), remainderd128() - Computes the remainder x REM y

Standards

Standards / Extensions	C or C++	Dependencies
C/C++ DFP	both	z/OS V1.10

Format

```
#define __STDC_WANT_DEC_FP__
#include <math.h>

_Decimal32  remainderd32(_Decimal32 x, _Decimal32 y);
_Decimal64  remainderd64(_Decimal64 x, _Decimal64 y);
_Decimal128 remainderd128(_Decimal128 x, _Decimal128 y);

_Decimal32  remainder(_Decimal32 x, _Decimal32 y);    /* C++ only */
_Decimal64  remainder(_Decimal64 x, _Decimal64 y);    /* C++ only */
_Decimal128 remainder(_Decimal128 x, _Decimal128 y);   /* C++ only */
```

General description

The remainder() function returns the decimal floating-point remainder when *y* is nonzero and following the relation

$$r = x - ny$$

The value *n* is the integral value nearest the exact value *x/y* and when then the value of n is even.

$$| n - x/y | = 1/2$$

These functions work in IEEE decimal floating-point format. See “IEEE decimal floating-point” on page 95 for more information.

Note: To use IEEE decimal floating-point, the hardware must have the Decimal Floating-Point Facility installed.

Returned value

If successful, remainder() returns the remainder of the division of *x* by *y*.

If *y* is zero, remainder() returns NaNQ and sets errno to EDOM.

Example

CELEBR23

```
/* CELEBR23

   This example illustrates the remainderd32() function.

*/

#define __STDC_WANT_DEC_FP__
#include <math.h>
#include <stdio.h>

void main() {
  _Decimal32 n1=3.0DF, n2=3.5DF;

  printf("Illustrates the remainderd32() function\n");

  printf("remainderd32(%.2Hf,%.2Hf)=%.2Hf\n",n1,n2,remainderd32(n1,n2));
  n1=1.0DF; n2=2.0DF;
  printf("remainderd32(%.2Hf,%.2Hf)=%.2Hf\n",n1,n2,remainderd32(n1,n2));
  n1=1.0DF; n2=0.0DF;
  printf("remainderd32(%.2Hf,%.2Hf)=%.2Hf\n",n1,n2,remainderd32(n1,n2));
}
```

Related information

- “math.h” on page 44

remove() — Delete file

Standards

Standards / Extensions	C or C++	Dependencies
ISO C POSIX.1 XPG4 XPG4.2 C99 Single UNIX Specification, Version 3	both	

Format

```
#include <stdio.h>

int remove(const char *filename);
```

General description

Deletes the file specified by *filename*, unless the file is open. The remove() function removes memory files and DASD data sets. (Non-DASD data sets, such as tapes, are not supported.) It also removes individual members of PDSs and PDSEs, and even removes memory files that simulate PDSs.

The interpretation of the file name passed to remove() depends on whether POSIX(ON) is specified. For full details about *filename* considerations, see the topics about opening files in *z/OS XL C/C++ Programming Guide*.

Memory files must exist and they must be closed. However, if you have z/OS UNIX C application running POSIX(ON), memory files don't need to be closed when removing an HFS memory file. The z/OS UNIX services rules of interoperability apply. See the topics about opening files in *z/OS XL C/C++ Programming Guide*, for specifying file names for MVS data sets and HFS files.

Special behavior for XPG4: If *filename* does not name a directory, *remove(filename)* is equivalent to *unlink(filename)*. If *filename* names a directory, *remove(filename)* is equivalent to *rmdir(filename)*.

Returned value

If successful, remove() returns 0.

If unsuccessful, remove() returns nonzero to indicate an error.

Example

CELEBR12

```
/* CELEBR12

   When you invoke this example with a file name, the program attempts to
   remove that file.
   It issues a message if an error occurs.

 */
#include <stdio.h>

int main(int argc, char ** argv)
{
  if ( argc != 2 )
    printf( "Usage: %s fn\n", argv[0] );
  else
    if ( remove( argv[1] ) != 0 )
      printf( "Could not remove file\n" );
}
```

Related information

- "stdio.h" on page 68
- "fopen() — Open a file" on page 565
- "rename() — Rename file" on page 1433

remque() — Remove an element from a double linked list

Standards

Standards / Extensions	C or C++	Dependencies
XPG4.2 Single UNIX Specification, Version 3	both	

Format

```
#define _XOPEN_SOURCE_EXTENDED 1
#include <search.h>

void remque(void *element);
```

General description

The remque() function removes the element pointed to by *element* from a doubly-linked list. The function operates on pointers to structures which have a pointer to their successor in the list as their first element, and a pointer to their predecessor as the second. The application is free to define the remaining contents of the structure, and manages all storage itself.

Returned value

remque() returns no values.

Related information

- “search.h” on page 62
- “insque() — Insert an element into a doubly-linked list” on page 868

remquo(), remquof(), remquol() — Computes the remainder.

Standards

Standards / Extensions	C or C++	Dependencies
C99 Single UNIX Specification, Version 3 C++ TR1 C99	both	z/OS V1R5

Format

```
#define _ISOC99_SOURCE
#include <math.h>

double remquo(double x, double y, int *quo);
float remquof(float x, float y, int *quo);
long double remquol(long double x, long double y, int *quo);
```

C++ TR1 C99

```
#define _TR1_C99
#include <math.h>

float remquo(float x, float y, int *quo);
long double remquo(long double x, long double y, int *quo);
```

General description

The remquo functions compute the same remainder as the remainder functions. In the object pointed to by *quo* they store a value whose sign is the sign of *x/y* and whose magnitude is congruent modulo 2 to the power *n* to the magnitude of the integral quotient of *x/y*, where n is an implementation defined integer greater than or equal to 3.

Note: The following table shows the viable formats for these functions. See "IEEE binary floating-point" on page 94 for more information about IEEE Binary Floating-Point.

Function	Hex	IEEE
remquo	X	X
remquof	X	X
remquol	X	X

Restriction: The remquof() function does not support the _FP_MODE_VARIABLE feature test macro.

Returned value

The remquo functions return *x* REM *y*.

Example

```
/*
 * This program illustrates the use of remquol() function
 *
 */
#define _ISOC99_SOURCE
#include <math.h>
#include <stdio.h>


void main() {

  long double number1=3.0L, number2=3.5L;
  int quo=0;

  printf("Illustrates the remquol() function");

  #ifdef __BFP__
    printf(" (IEEE version)\n\n");
  #else
    printf(" (HFP version)\n\n");
  #endif

  printf("remquol(%.2Lf,%.2Lf,&quo)=%.2Lf",number1,number2,remquol(number1,number2,&(quo)));
  printf("   quo=%i\n",quo);
  number1=1.0L; number2=2.0L;
  printf("remquol(%.2Lf,%.2Lf,&quo)=%.2Lf",number1,number2,remquol(number1,number2,&(quo)));
  printf("    quo=%i\n",quo);
  number1=1.0L; number2=0.0L;
  printf("remquol(%.2Lf,%.2Lf,&quo)=%.2Lf",number1,number2,remquol(number1,number2,&(quo)));
  printf(" quo=%i\n",quo);
}
```

```
Output

Illustrates the remquol() function (IEEE version)

remquol(3.00,3.50,&quo)=-0.50   quo=1
remquol(1.00,2.00,&quo)=1.00    quo=0
remquol(1.00,0.00,&quo)=NaNQ(1) quo=0
```

Related information

- "math.h" on page 44

__remquod32(), __remquod64(), __remquod128() — Computes the remainder.

Standards

Standards / Extensions	C or C++	Dependencies
Language Environment	both	z/OS V1.11

Format

```
#define __STDC_WANT_DEC_FP__
#include <math.h>

_Decimal32  __remquod32(_Decimal32 x, _Decimal32 y, int *quo);
_Decimal64  __remquod64(_Decimal64 x, _Decimal64 y, int *quo);
_Decimal128 __remquod128(_Decimal128 x, _Decimal128 y, int *quo);
```

General description

The __remquo() functions compute the same remainder as the remainder functions. In the object pointed to by *quo* they store a value whose sign is the sign of *x* or *y* and whose magnitude is congruent modulo 2 to the power *n* to the magnitude of the integral quotient of *x* or *y*, where n is an implementation defined integer greater than or equal to 3.

Notes:

1. These functions work in IEEE decimal floating-point format. See "IEEE decimal floating-point" on page 95 for more information.
2. To use IEEE decimal floating-point, the hardware must have the Decimal Floating-Point Facility installed.

Returned value

The __remquo() functions return *x* REM *y*.

Example

```
/* CELEBR24

   This example illustrates the __remquod64() function.

*/

#define __STDC_WANT_DEC_FP__
#include <math.h>
#include <stdio.h>
void main() {
   _Decimal64 x, y, z;
   int n;

   x = 3.0DD;
   y = 3.5DD;
   z = __remquod64(x, y, &n);

   printf("__remquod64( %Df, %Df, %d ) = %Df\n", x, y, n, z);
}
```

Related information

- “math.h” on page 44

rename() — Rename file

Standards

Standards / Extensions	C or C++	Dependencies
ISO C POSIX.1 XPG4 XPG4.2 C99 Single UNIX Specification, Version 3	both	

Format

```
#include <stdio.h>

int rename(const char *oldname, const char *newname);
```

General description

Changes the name of the file, from the name pointed to by *oldname* to the name pointed to by *newname*.

The *oldname* pointer must point to the name of an existing file. The *newname* pointer must not specify the name of an existing file. You cannot rename an open file. In case of an error, the name of the file is not changed.

The rename() function renames memory files and DASD data sets. (Non-DASD data sets, such as tapes, are not supported.) It also renames individual members of PDSs (and PDSEs); it even renames files that simulate PDSs.

Special behavior for POSIX C: Memory files must be closed unless you are working under z/OS UNIX services.

The interpretation of the file name passed to rename() depends on whether the program is running POSIX(ON) or POSIX(OFF).

You cannot rename an HFS file to an MVS data set name or rename an MVS data set to an HFS file name.

Both *oldname* and *newname* must be of the same type, that is, both directories or both files.

If *newname* already exists, it is removed before *oldname* is renamed to *newname*. Thus, if *newname* specifies the name of an existing directory, it must be an empty directory.

If the *oldname* argument points to a symbolic link, the symbolic link is renamed. If the *newname* argument points to a symbolic link, the link is removed and *oldname* is renamed to *newname*. rename() does not affect any file or directory named by the contents of the symbolic link.

For rename() to succeed, the process needs write permission on the directory containing *oldname* and the directory containing *newname*. If *oldname* and *newname* are directories, rename() also needs write permission on the directories themselves.

If *oldname* and *newname* both refer to the same file, rename() returns successfully and performs no other action.

When rename() is successful, it updates the change and modification times for the parent directories of *oldname* and *newname*.

Returned value

If successful, rename() returns 0.

If unsuccessful, rename() returns nonzero and sets errno to one of the following values:

Error Code
    Description

EACCES
    An error occurred for one of these reasons:
- The process did not have search permission on some component of the old or new path name.
- The process did not have write permission on the parent directory of the file or directory to be renamed.
- *oldname* or *newname* were directories.
- The process did not have write permission on *oldname* or *newname*.

EBUSY
    oldname and *newname* specify directories, but one of them cannot be renamed because it is in use as a root or a mount point.

EINVAL
    This error occurs for one of these reasons:
- *oldname* is part of the path name prefix of *newname*.
- *oldname* or *newname* refers to either `.` (dot) or `..` (dot-dot).

EIO A physical I/O error has occurred.

EISDIR
    newname is a directory, but *oldname* is not a directory.

ELOOP
    A loop exists in symbolic links. This error is issued if the number of symbolic links encountered during resolution of *oldname* or *newname* is greater than POSIX_SYMLOOP.

ENAMETOOLONG
    pathname is longer than **PATH_MAX** characters, or some component of *pathname* is longer than **NAME_MAX** characters while _POSIX_NO_TRUNC is in effect. For symbolic links, the length of the path name string substituted for a symbolic link exceeds **PATH_MAX**. The **PATH_MAX** and **NAME_MAX** values can be determined using pathconf().

ENOENT
    No file or directory named *oldname* was found, or either *oldname* or *newname* was not specified.

ENOSPC
The directory intended to contain *newname* cannot be extended.

ENOTDIR
A component of the path name prefix for *oldname* or *newname* is not a directory, or *oldname* is a directory and *newname* is a file that is not a directory.

ENOTEMPTY
newname specifies a directory, but the directory is not empty.

EPERM or EACCES
The S_ISVTX flag is set on the directory containing the file referred to by *oldname* and the caller is not the file owner, nor is the caller the directory owner, nor does the caller have appropriate privileges; or *newname* refers to an existing file, the S_ISVTX flag is set on the directory containing this file and the caller is not the file owner, nor is the caller the directory owner, nor does the caller have appropriate privileges.

EROFS
Renaming would require writing on a read-only file system.

EXDEV
oldname and *newname* identify files or directories on different file systems. z/OS UNIX services do not support links between different files systems.

Example

CELEBR13

```
/* CELEBR13

   This example takes two file names as input and uses rename() to change
   the file name from the first name to the second name.

 */
#include <stdio.h>

int main(int argc, char ** argv )
{
  if ( argc != 3 )
    printf( "Usage: %s old_fn new_fn\n", argv[0] );
  else if ( rename( argv[1], argv[2] ) != 0 )
    printf( "Could not rename file\n" );
}
```

Related information

- "stdio.h" on page 68
- "fopen() — Open a file" on page 565
- "remove() — Delete file" on page 1428

res_init() — Domain name resolver initialization

Standards

Standards / Extensions	C or C++	Dependencies
BSD 4.3	both	OS/390 V2R8

Format

```
#define _XOPEN_SOURCE_EXTENDED 1
#include <sys/types.h>
#include <netinet/in.h>
#include <arpa/nameser.h>
#include <resolv.h>

int res_init(void);
struct __res_state _res;
```

General description

The res_init() function is the Resolver function that initializes the __res_state structure for use by other Resolver functions. Initialization normally occurs on the first call to any of the IP address resolution routines commonly called the XL C/C++ Runtime Library Resolver.

The res_init() routine does its initialization by passing the __res_state structure to the CS for z/OS Resolver. The Resolver reads the "TCPIP.DATA" configuration file and updates the __res_state structure. The data in the __res_state structure is filled in based on the contents of the "TCPIP.DATA" configuration file and can then be referenced in the *_res* variable. Global configuration and state information that is used by the Resolver routines is kept in the structure *_res*. Most of the values have reasonable defaults and can be left unchanged.

Value Description

_res.retrans
Retransmission time interval is taken from the ResolverTimeOut statement found in the "TCPIP.DATA" configuration file.

_res.retry
The number of times to retransmit a request. It is taken from the ResolverUDPRetries statement found in the "TCPIP.DATA" configuration file.

_res.options
Options stored in _res.options are defined in <resolv.h> and are listed below. Options are stored as a simple bit mask containing the bitwise OR of the options enabled.

Option Description

RES_INIT
True after the initial name server address and default domain name are initialized, because res_init() has been called. This option should only be tested but not set, except by the res_init() function.

RES_DEBUG
Print debugging messages.

RES_AAONLY
Accept authoritative answers only. With this option, res_send() should continue until it finds an authoritative answer or finds an error. Currently this is not implemented.

RES_USEVC
Use TCP connections for queries instead of UDP datagrams.

RES_STAYOPEN
Used with RES_USEVC to keep the TCP connection open between

queries. This is useful only in programs that regularly do many queries. UDP should be the normal mode used.

RES_IGNTC
Ignore truncation errors, that is, don't retry with TCP. Currently unused.

RES_RECURSE
Set the recursion-desired bit in queries. This is the default. (res_send() does not do iterative queries and expects the name server to handle recursion.)

RES_DEFNAMES
If set, res_search() will append the default domain name to single-component names (those that do not contain a dot). This option is enabled by default.

RES_DNSRCH
If this option is set, res_search() will search for host names in the current domain and in parent domains. This is used by the standard host lookup routine gethostbyname(). This option is enabled by default.

RES_NOALIASES
This option turns off the user level aliasing feature controlled by the "HOSTALIASES" environment variable. Network daemons should set this option.

_res.nscount
The number of name servers specified in the "TCPIP.DATA" configuration file.

_res.*nsaddr_list[0]
The addresses of name servers specified by the NSINTERADDR or NameServer statements found in the "TCPIP.DATA" configuration file.

_res.dnsrch[0]
The beginning of the list of domains to be searched, as specified in the SEARCH statement found in the "TCPIP.DATA" configuration file. The structure will have either a Default DOMAIN or SEARCH.

_res.defdname[0]
The Default Domain name, as specified in the Domain or DomainOrigin statement found in the "TCPIP.DATA" configuration file. The structure will have either a Default DOMAIN or SEARCH.

_res.pfcode
Currently this is not implemented.

_res.ndots
The threshold for the number of dots in the domain name, as specified by the OPTIONS statement value ndots:n found in the "TCPIP.DATA" configuration file. The default is 1.

_res.nsort
The number of elements in sort_list[] as listed in the SORTLIST statement found in the "TCPIP.DATA" configuration file.

_res.sort_list[0]
The network address and subnet mask in the SORTLIST statement found in the "TCPIP.DATA" configuration file.

Returned value

If successful, res_init() returns 0.

If unsuccessful, res_init() returns -1 and sets h_errno to one of the following values:

Error Code
Description

NO_RECOVERY
An error occurred that will continue to fail if tried again. Storage could not be obtained for this thread to contain the _res structure.

TRY_AGAIN
An error occurred while initializing the __res_state structure name selected, which can be retried.

If successful, *_res* returns the address of __res_state structure.

If unsuccessful, *_res* returns NULL and sets errno to one of the following values:

Error Code
Description

ENOMEM
The storage needed to define the *_res* structure could not be obtained.

Related information

- For additional information on the TCPIP.DATA configuration, see *z/OS Communications Server: IP Configuration Guide, SC27-3650*.
- "arpa/nameser.h" on page 17
- "netinet/in.h" on page 53
- "resolv.h" on page 62
- "sys/types.h" on page 75
- "dn_comp() — Resolver domain name compression" on page 398
- "dn_expand() — Resolver domain name expansion" on page 399
- "dn_find() — Resolver domain name find" on page 400
- "dn_skipname() — Resolver domain name skipping" on page 401
- "gethostbyname() — Get a host entry by name" on page 719
- "res_mkquery() — Make resolver query for domain name servers"
- "res_query() — Resolver query for domain name servers" on page 1439
- "res_querydomain() — Build domain name and resolver query" on page 1441
- "res_search() — Resolver query for domain name servers" on page 1442
- "res_send() — Send resolver query for domain name servers" on page 1443

res_mkquery() — Make resolver query for domain name servers

Standards

Standards / Extensions	C or C++	Dependencies
BSD 4.3	both	OS/390 V2R8

Format

```
#define _XOPEN_SOURCE_EXTENDED 1
#include <sys/types.h>
#include <netinet/in.h>
#include <arpa/nameser.h>
```

```
#include <resolv.h>

int res_mkquery(int op, const char *dname, int class, int type, const u_char *data,
                int datalen, const u_char *newrr_in, u_char *buf, int buflen);
```

General description

This routine is one of several functions used for making, sending and interpreting query and reply messages with Internet domain name servers (DNS).

The res_mkquery() function constructs a standard query message and places it in *buf*. It returns the size of the query, or -1 if the query is larger than *buflen*. The query type *op* is usually QUERY, but can be any of the query types defined in <arpa/nameser.h>. The domain name for the query given by *dname*. The argument *newrr_in* is currently unused but is intended for making update messages.

Note: The res_mkquery() function has a dependency on the level of the Enhanced ASCII Extensions. See "Enhanced ASCII support" on page 2113 for details.

Returned value

If successful, res_mkquery() returns the size of the query.

If unsuccessful, res_mkquery() returns -1. The errors defined in <arpa/nameser.h> can be found in the buf.rcode, if an answer was supplied in the *buf* buffer.

Related information

- "arpa/nameser.h" on page 17
- "netinet/in.h" on page 53
- "resolv.h" on page 62
- "sys/types.h" on page 75
- "dn_comp() — Resolver domain name compression" on page 398
- "dn_expand() — Resolver domain name expansion" on page 399
- "dn_find() — Resolver domain name find" on page 400
- "dn_skipname() — Resolver domain name skipping" on page 401
- "res_init() — Domain name resolver initialization" on page 1435
- "res_query() — Resolver query for domain name servers"
- "res_querydomain() — Build domain name and resolver query" on page 1441
- "res_search() — Resolver query for domain name servers" on page 1442
- "res_send() — Send resolver query for domain name servers" on page 1443

res_query() — Resolver query for domain name servers

Standards

Standards / Extensions	C or C++	Dependencies
BSD 4.3	both	OS/390 V2R8

Format

```
#define _XOPEN_SOURCE_EXTENDED 1
#include <sys/types.h>
#include <netinet/in.h>
#include <arpa/nameser.h>
#include <resolv.h>

int res_query(const char *dname, int class, int type, u_char *answer, int anslen);
```

General description

This routine is one of several functions used for making, sending and interpreting query and reply messages with Internet domain name servers (DNS).

The res_query() function provides an interface to the server query mechanism. It constructs a query, sends it to the local server, awaits a response, and makes preliminary checks on the reply. The query requests information of the specified type and class for the specified fully-qualified domain name *dname*. The reply message is left in the *answer* buffer with length *anslen* supplied by the caller.

Note: The res_query() function has a dependency on the level of the Enhanced ASCII Extensions. See "Enhanced ASCII support" on page 2113 for details.

Returned value

If successful, res_query() returns the reply message in the *answer* buffer with length *anslen*.

If unsuccessful, res_query() returns -1 and sets herrno to one of the following values:

Error Code
 Description

HOST_NOT_FOUND
 The host name provided is not known at any of the domain name servers queried for this request.

NO_DATA
 An answer was received but no data was supplied in the *answer* buffer.

NO_RECOVERY
 An error occurred that will continue to fail if tried again.

TRY_AGAIN
 A error occurred querying the name selected, which can be retried.

Related information

- "arpa/nameser.h" on page 17
- "netinet/in.h" on page 53
- "resolv.h" on page 62
- "sys/types.h" on page 75
- "dn_comp() — Resolver domain name compression" on page 398
- "dn_expand() — Resolver domain name expansion" on page 399
- "dn_find() — Resolver domain name find" on page 400
- "dn_skipname() — Resolver domain name skipping" on page 401
- "res_init() — Domain name resolver initialization" on page 1435
- "res_mkquery() — Make resolver query for domain name servers" on page 1438
- "res_querydomain() — Build domain name and resolver query" on page 1441
- "res_search() — Resolver query for domain name servers" on page 1442
- "res_send() — Send resolver query for domain name servers" on page 1443

res_querydomain() — Build domain name and resolver query

Standards

Standards / Extensions	C or C++	Dependencies
BSD 4.3	both	OS/390 V2R8

Format

```
#define _XOPEN_SOURCE_EXTENDED 1
#include <sys/types.h>
#include <netinet/in.h>
#include <arpa/nameser.h>
#include <resolv.h>

int res_querydomain(const char *name, const char *domain, int class, int type,
                    u_char *answer, int anslen);
```

General description

This routine is one of several functions used for making, sending and interpreting query and reply messages with Internet domain name servers (DNS).

The res_querydomain() function builds a fully qualified domain name and returns a res_query() to the caller.

Note: The res_querydomain() function has a dependency on the level of the Enhanced ASCII Extensions. See "Enhanced ASCII support" on page 2113 for details.

Returned value

If successful, res_querydomain() returns a res_query() to the caller.

If unsuccessful, res_querydomain() returns -1 and sets herrno to one of the following values:

Error Code
Description

HOST_NOT_FOUND
The host name provided is not known at any of the domain name servers queried for this request.

NO_DATA
An answer was received but no data was supplied in the *answer* buffer.

NO_RECOVERY
An error occurred that will continue to fail if tried again.

TRY_AGAIN
A error occurred querying the name selected, which can be retried.

Related information

- "arpa/nameser.h" on page 17
- "netinet/in.h" on page 53
- "resolv.h" on page 62
- "sys/types.h" on page 75
- "dn_comp() — Resolver domain name compression" on page 398

- “dn_expand() — Resolver domain name expansion” on page 399
- “dn_find() — Resolver domain name find” on page 400
- “dn_skipname() — Resolver domain name skipping” on page 401
- “res_init() — Domain name resolver initialization” on page 1435
- “res_mkquery() — Make resolver query for domain name servers” on page 1438
- “res_query() — Resolver query for domain name servers” on page 1439
- “res_search() — Resolver query for domain name servers”
- “res_send() — Send resolver query for domain name servers” on page 1443

res_search() — Resolver query for domain name servers

Standards

Standards / Extensions	C or C++	Dependencies
BSD 4.3	both	OS/390 V2R8

Format

```
#define _XOPEN_SOURCE_EXTENDED 1
#include <sys/types.h>
#include <netinet/in.h>
#include <arpa/nameser.h>
#include <resolv.h>

int res_search(const char *dname, int class, int type, u_char *answer, int anslen);
```

General description

This routine is one of several functions used for making, sending and interpreting query and reply messages with Internet domain name servers (DNS).

The res_search() routine makes a query and awaits a response like res_query() but, in addition, it implements the default and search rules controlled by the RES_DEFNAMES and RES_DNSRCH options. It returns the first successful reply.

Note: The res_search() function has a dependency on the level of the Enhanced ASCII Extensions. See “Enhanced ASCII support” on page 2113 for details.

Returned value

If successful, res_search() returns the first successful reply.

If unsuccessful, res_search() returns -1 and sets herrno to one of the following values:

Error Code
Description

HOST_NOT_FOUND
The host name provided is not known at any of the domain name servers queried for this request.

NO_DATA
An answer was received but no data was supplied in the *answer* buffer.

NO_RECOVERY
An error occurred that will continue to fail if tried again.

TRY_AGAIN
A error occurred querying the name selected, which can be retried.

Related information

- “arpa/nameser.h” on page 17
- “netinet/in.h” on page 53
- “resolv.h” on page 62
- “sys/types.h” on page 75
- “dn_comp() — Resolver domain name compression” on page 398
- “dn_expand() — Resolver domain name expansion” on page 399
- “dn_find() — Resolver domain name find” on page 400
- “dn_skipname() — Resolver domain name skipping” on page 401
- “res_init() — Domain name resolver initialization” on page 1435
- “res_mkquery() — Make resolver query for domain name servers” on page 1438
- “res_query() — Resolver query for domain name servers” on page 1439
- “res_querydomain() — Build domain name and resolver query” on page 1441
- “res_send() — Send resolver query for domain name servers”

res_send() — Send resolver query for domain name servers

Standards

Standards / Extensions	C or C++	Dependencies
BSD 4.3	both	OS/390 V2R8

Format

```
#define _XOPEN_SOURCE_EXTENDED 1
#include <sys/types.h>
#include <netinet/in.h>
#include <arpa/nameser.h>
#include <resolv.h>

int res_send(const u_char *msg, int msglen, u_char *answer, int anslen);
```

General description

This routine is one of several functions used for sending query and reply messages with Internet domain name servers (DNS).

The res_send() routine sends a pre-formatted query and returns an answer. It will call res_init() if RES_INIT is not set, send the query to the local name server, and handle timeouts and retries. The length of the reply message is returned, or -1 if there were errors.

Note: The res_send() function has a dependency on the level of the Enhanced ASCII Extensions. See “Enhanced ASCII support” on page 2113 for details.

Returned value

If successful, res_send() returns the length of the reply message.

If unsuccessful, res_send() returns -1. The errors defined in <arpa/nameser.h> can be found in the buf.rcode, if an answer was supplied in the *answer* buffer.

Related information

- “arpa/nameser.h” on page 17
- “netinet/in.h” on page 53
- “resolv.h” on page 62
- “sys/types.h” on page 75
- “dn_comp() — Resolver domain name compression” on page 398
- “dn_expand() — Resolver domain name expansion” on page 399
- “dn_find() — Resolver domain name find” on page 400
- “dn_skipname() — Resolver domain name skipping” on page 401
- “res_init() — Domain name resolver initialization” on page 1435
- “res_mkquery() — Make resolver query for domain name servers” on page 1438
- “res_query() — Resolver query for domain name servers” on page 1439
- “res_querydomain() — Build domain name and resolver query” on page 1441
- “res_search() — Resolver query for domain name servers” on page 1442

__reset_exception_handler() — Unregister an exception handler routine

Standards

Standards / Extensions	C or C++	Dependencies
	both	

Format

```
#include <__le_api.h>

int  __reset_exception_handler(void);
```

General description

A nonstandard function that unregisters the 'Exception Handler' function, that was previously registered via the __set_exception_handler() function, for the current stack frame.

Returned value

If successful, __reset_exception_handler() returns 0. Otherwise, -1 is returned and errno is set to indicate the error. The following is a possible value for errno:

- EINVAL — No Exception Handler is registered in the current stack frame.

Related information

- “__set_exception_handler() — Register an exception handler routine” on page 1533

rewind() — Set file position to beginning of file

Standards

Standards / Extensions	C or C++	Dependencies
ISO C POSIX.1 XPG4 XPG4.2 C99 Single UNIX Specification, Version 3 Language Environment	both	

Format

```
#include <stdio.h>

void rewind(FILE *stream);

#define _OPEN_SYS_UNLOCKED_EXT 1
#include <stdio.h>

void rewind_unlocked(FILE *stream);
```

General description

Repositions the file position indicator of the stream pointed to by *stream*. A call to rewind() is the same as the statement below, except that rewind() also clears the error indicator for the *stream*.

```
(void) fseek(stream, 0L, SEEK_SET);
```

rewind_unlocked() is functionally equivalent to rewind() with the exception that it is not thread-safe. This function can safely be used in a multithreaded application if and only if it is called while the invoking thread owns the (FILE*) object, as is the case after a successful call to either the flockfile() or ftrylockfile() function.

Returned value

rewind() returns no values.

If an error occurs, errno is set. After the error, the file position does not change. The next operation may be either a read or a write operation.

Special behavior for XPG4.2: The rewind() function returns -1 and sets errno to ESPIPE if the underlying file type for the stream is a PIPE or a socket.

Example

CELEBR14

```
/* CELEBR14

   This example first opens a file myfile for input
   and output.
   It writes integers to the file, uses &rewind. to reposition
   the file pointer to the beginning of the file, and then reads
   the data back in.

 */
#include <stdio.h>
```

```
int main(void)
{
   FILE *stream;
   int data1, data2, data3, data4;
   data1 = 1; data2 = -37;

      /* Place data in the file */
   stream = fopen("myfile.dat", "w+");
   fprintf(stream, "%d %d\n", data1, data2);

      /* Now read the data file */
   rewind(stream);
   fscanf(stream, "%d", &data3);
   fscanf(stream, "%d", &data4);
   printf("The values read back in are: %d and %d\n",
       data3, data4);
}
```

Output

```
The values read back in are: 1 and -37
```

Related information

- "stdio.h" on page 68
- "fgetpos() — Get file position" on page 528
- "fseek() — Change file position" on page 639
- "fsetpos() — Set file position" on page 649
- "ftell() — Get current file position" on page 657

rewinddir() — Reposition a directory stream to the beginning

Standards

Standards / Extensions	C or C++	Dependencies
POSIX.1 XPG4 XPG4.2 Single UNIX Specification, Version 3	both	

Format

```
#define_POSIX_SOURCE
#include <dirent.h>

void rewinddir(DIR *dir);
```

General description

Repositions an open directory stream to the beginning. *dir* points to a DIR object associated with an open directory.

The next call to readdir() reads the first entry in the directory. If the contents of the directory have changed since the directory was opened, a call to rewinddir() updates the directory stream so that a subsequent readdir() can read the new contents.

Returned value

rewinddir() returns no values.

Example

CELEBR15

```
/* CELEBR15

   This example produces the contents of a directory by opening it,
   rewinding it, and closing it.

 */
#define _POSIX_SOURCE
#include <dirent.h>
#include <errno.h>
#include <sys/types.h>
#undef _POSIX_SOURCE
#include <stdio.h>

main() {
  DIR *dir;
  struct dirent *entry;

  if ((dir = opendir("/")) == NULL)
    perror("opendir() error");
  else {
    puts("contents of root:");
    while ((entry = readdir(dir)) != NULL)
      printf("%s ", entry->d_name);
    rewinddir(dir);
    puts("");
    while ((entry = readdir(dir)) != NULL)
      printf("%s ", entry->d_name);
    closedir(dir);
    puts("");
  }
}
```

Output

```
contents of root:
  . .. bin dev etc lib tmp u usr
  . .. bin dev etc lib tmp u usr
```

Related information

- “dirent.h” on page 22
- “stdio.h” on page 68
- “sys/types.h” on page 75
- “closedir() — Close a directory” on page 291
- “opendir() — Open a directory” on page 1158
- “readdir() — Read an entry from a directory” on page 1382
- “seekdir() — Set position of directory stream” on page 1469
- “telldir() — Current location of directory stream” on page 1866

rexec() — Execute commands one at a time on a remote host

Standards

Standards / Extensions	C or C++	Dependencies
z/OS UNIX	both	POSIX(ON)

Format

```
#include <rexec.h>

int rexec(char **Host, int Port, char *User, char *Password,
          char *Command, int *ErrFileDescParam)
```

General description

The rexec (remote execution) subroutine allows the calling process to execute commands on a remote host. If the rexec connection succeeds, a socket in the Internet domain of type SOCK_STREAM is returned to the calling process and is given to the remote command as standard input and standard output.

Host contains the name of a remote host that is listed in the /etc/hosts file or /etc/resolv.config file. If the name of the host is not found in either file, the rexec fails.

Port specifies the well-known Defense Advanced Research Projects Agency (DARPA) Internet port to use for the connection. A pointer to the structure that contains the necessary port can be obtained by issuing the following library call: getservbyname("exec","tcp").

User and Password points to a user ID and password valid at the host. Password phrases are not supported.

Command points to the name of the command to be executed at the remote host.

ErrFileDescParam specifies one of the following values:

- Not 0 (zero) = an auxiliary channel to a control process is set up, and a descriptor for it is placed in the ErrFileDescParam parameter. The control process provides diagnostic output from the remote command on this channel and also accepts bytes as signal numbers to be forwarded to the process group of the command. This diagnostic information does not include remote authorization failure, since this connection is set up after authorization has been verified.
- 0 (zero) = the standard error of the remote command is the same as standard output, and no provision is made for sending arbitrary signals to the remote process. In this case, however, it may be possible to send out-of-band data to the remote command.

This function is supported only in a POSIX program.

Note: The rexec() function has a dependency on the level of the Enhanced ASCII Extensions. See “Enhanced ASCII support” on page 2113 for details.

Returned value

If rexec() is successful, the system returns a socket to the remote command.

If rexec() is unsuccessful, the system returns a -1 indicating that the specified host name does not exist.

Related information

- “rexec.h” on page 62
- “getservbyname() — Get a server entry by name” on page 770
- “rexec_af() — Execute commands one at a time on a remote host” on page 1449

rexec_af() — Execute commands one at a time on a remote host

Standards

Standards / Extensions	C or C++	Dependencies
RCF2292	both	z/OS V1R4

Format

```
#define  _OPEN_SYS_SOCK_IPV6
#include <rexec.h>

int rexec_af(char **ahost, unsigned short rport,
              const char *name, const char *pass, const char *cmd,
              int *fd2p, int af);
```

General description

The rexec_af() function behaves the same as the rexec() function. Instead of creating an AF_INET socket, rexec_af can also create an AF_INET6 socket. The *af* argument specifies the address family. It is set to either AF_INET or AF_INET6.

Note: The rexec_af() function has a dependency on the level of the Enhanced ASCII Extensions. See “Enhanced ASCII support” on page 2113 for details.

Returned value

When successful, rexec_af() returns a socket to the remote command. If unsuccessful, rexec_af() returns -1 and may set errno to one of the following:

EAFNOSUPPORT
The specified address family is not supported.

Related information

- “rexec.h” on page 62
- “getservbyname() — Get a server entry by name” on page 770
- “rexec() — Execute commands one at a time on a remote host” on page 1447

rindex() — Search for character

Standards

Standards / Extensions	C or C++	Dependencies
XPG4.2 Single UNIX Specification, Version 3	both	

Format

```
#define  _XOPEN_SOURCE_EXTENDED 1
#include <strings.h>

char *rindex(const char *string, int c);
```

General description

The rindex() function locates the last occurrence of *c* (converted to an unsigned char) in the string pointed to by *string*.

The string argument to the function must contain a NULL character (\0) marking the end of the string.

The rindex() function is identical to “strrchr() — Find last occurrence of character in string” on page 1759.

Note: The rindex() function has been moved to the Legacy Option group in Single UNIX Specification, Version 3 and may be withdrawn in a future version. The strrchr() function is preferred for portability.

Returned value

If successful, rindex() returns a pointer to the first occurrence of *c* (converted to an unsigned character) in the string pointed to by *string*.

If *c* was not found, rindex() returns a NULL pointer.

There are no errno values defined.

Related information

- “strings.h” on page 72
- “index() — Search for character” on page 841
- “memchr() — Search buffer” on page 1061
- “strchr() — Search for character” on page 1729
- “strrchr() — Find last occurrence of character in string” on page 1759
- “strspn() — Search string” on page 1761
- “strstr() — Locate substring” on page 1762

rint(), rintf(), rintl() — Round to nearest integral value

Standards

Standards / Extensions	C or C++	Dependencies
XPG4.2 C99 Single UNIX Specification, Version 3 C++ TR1 C99	both	

Format

```
#define _XOPEN_SOURCE_EXTENDED 1
#include <math.h>

double rint(double x);
```

C99

```
#define _ISOC99_SOURCE
#include <math.h>

float rintf(float x);
long double rintl(long double x);
```

C++ TR1 C99

```
#define _TR1_C99
#include <math.h>

float rint(float x);
long double rint(long double x);
```

General description

The rint() functions return the integral value (represented in a floating-point mode) nearest *x* using the round to nearest mode and may raise the “inexact” floating-point exception if the result differs in value from the argument.

Note: The following table shows the viable formats for these functions. See “IEEE binary floating-point” on page 94 for more information about IEEE Binary Floating-Point.

Function	Hex	IEEE
rint	X	X
rintf	X	X
rintl	X	X

Returned value

rint() is always successful in IEEE.

Special behavior for hex: The rint() functions always round toward zero in hexadecimal math.

Related information

- “math.h” on page 44
- “abs(), absf(), absl() — Calculate integer absolute value” on page 103
- “isnan() — Test for NaN” on page 916

rintd32(), rintd64(), rintd128() — Round to nearest integral value

Standards

Standards / Extensions	C or C++	Dependencies
C/C++ DFP	both	z/OS V1.8

Format

```
#define __STDC_WANT_DEC_FP__
#include <math.h>

_Decimal32  rintd32(_Decimal32 x);
_Decimal64  rintd64(_Decimal64 x);
_Decimal128 rintd128(_Decimal128 x);

_Decimal32  rint(_Decimal32 x);      /* C++ only */
_Decimal64  rint(_Decimal64 x);      /* C++ only */
_Decimal128 rint(_Decimal128 x);     /* C++ only */
```

General description

These functions return the integral value (represented in a decimal floating-point mode) nearest *x* according to the rounding mode and might raise the “inexact” decimal floating-point exception if the result differs in value from the argument.

Notes:

1. To use IEEE decimal floating-point, the hardware must have the Decimal Floating-Point Facility installed.
2. These functions work in IEEE decimal floating-point format. See "IEEE Decimal Floating-Point" for more information.

Returned value

These functions are always successful.

Example

```
/* CELEBR21

   This example illustrates the rintd32() function.

*/

#pragma strings(readonly)

#define __STDC_WANT_DEC_FP__
#include <fenv.h>
#include <math.h>
#include <stdio.h>

static void try_rm(int);

/* pass back printable rounding mode */

static
char *rm_str(int rm)
{
  char *s = "undetermined";

  switch (rm)
  {
    case    FE_DEC_TONEAREST            :
      s =  "FE_DEC_TONEAREST"           ;  break;
    case    FE_DEC_TOWARDZERO           :
      s =  "FE_DEC_TOWARDZERO"          ;  break;
    case    FE_DEC_UPWARD               :
      s =  "FE_DEC_UPWARD"              ;  break;
    case    FE_DEC_DOWNWARD             :
      s =  "FE_DEC_DOWNWARD"            ;  break;
    case    FE_DEC_TONEARESTFROMZERO    :
      s =  "FE_DEC_TONEARESTFROMZERO"   ;  break;
    case   _FE_DEC_TONEARESTTOWARDZERO  :
      s = "_FE_DEC_TONEARESTTOWARDZERO" ;  break;
    case   _FE_DEC_AWAYFROMZERO         :
      s = "_FE_DEC_AWAYFROMZERO"        ;  break;
    case   _FE_DEC_PREPAREFORSHORTER    :
      s = "_FE_DEC_PREPAREFORSHORTER"   ;  break;
   }

   return s;
}
```

```
/* Try out one passed-in number with rounding mode */

static void try_rm(int rm)
{
  _Decimal32 r32;
  _Decimal32 d32 = 500.99DF;

  (void)fe_dec_setround(rm);

  r32 = rintd32(d32);

  printf("rintd32(%.2HF) = %HG - rounding mode = %s\n",
         d32, r32, rm_str(rm)
        );

  return;
}


int main()
{
  try_rm( FE_DEC_TONEAREST          );
  try_rm( FE_DEC_TOWARDZERO         );
  try_rm( FE_DEC_UPWARD             );
  try_rm( FE_DEC_DOWNWARD           );
  try_rm( FE_DEC_TONEARESTFROMZERO  );
  try_rm(_FE_DEC_TONEARESTTOWARDZERO);
  try_rm(_FE_DEC_AWAYFROMZERO       );
  try_rm(_FE_DEC_PREPAREFORSHORTER  );

  return 0;
}
```

Related information

- "math.h" on page 44
- "isnan() — Test for NaN" on page 916
- "rint(), rintf(), rintl() — Round to nearest integral value" on page 1450

rmdir() — Remove a directory

Standards

Standards / Extensions	C or C++	Dependencies
POSIX.1 XPG4 XPG4.2 Single UNIX Specification, Version 3	both	

Format

```
#define_POSIX_SOURCE
#include <unistd.h>

int rmdir(const char *pathname);
```

General description

Removes a directory, *pathname*, provided that the directory is empty. *pathname* must not end in . (dot) or .. (dot-dot).

If *pathname* refers to a symbolic link, rmdir() does not affect any file or directory named by the contents of the symbolic link. rmdir() does not remove a directory that still contains files or subdirectories.

Special behavior for XPG4.2: If *pathname* refers to a symbolic link, rmdir() fails and sets errno to ENOTDIR.

If no process currently has the directory open, rmdir() deletes the directory itself. The space occupied by the directory is freed for new use. If one or more processes have the directory open when it is removed, the directory itself is not removed until the last process closes the directory. New files cannot be created under a directory after the last link is removed, even if the directory is still open.

rmdir() removes the directory even if it is the working directory of a process.

If rmdir() is successful, the change and modification times for the parent directory are updated.

Returned value

If successful, rmdir() returns 0.

If unsuccessful, rmdir() returns -1 and sets errno to one of the following values:

Error Code
Description

EACCES
The process did not have search permission for some component of *pathname*, or it did not have write permission for the directory containing the directory to be removed.

EBUSY
pathname cannot be removed, because it is currently being used by the system or a process.

EINVAL
The last component of *pathname* contains a . (dot) or a .. (dot-dot).

EIO A physical I/O error has occurred.

ELOOP
A loop exists in symbolic links. More than POSIX_SYMLOOP (an integer defined in the limits.h header file) symbolic links are detected in the resolution of *pathname*.

ENAMETOOLONG
pathname is longer than **PATH_MAX** characters or some component of *pathname* is longer than **NAME_MAX** characters while _POSIX_NO_TRUNC is in effect. For symbolic links, the length of the path name string substituted for a symbolic link exceeds **PATH_MAX**. The **PATH_MAX** and **NAME_MAX** values can be determined using pathconf().

ENOENT
pathname does not exist, or it is an empty string.

ENOTDIR
Some component of the *pathname* prefix is not a directory.

ENOTEMPTY
The directory still contains files or subdirectories.

EPERM or EACCES
The S_ISVTX flag is set on the parent directory of the directory to be removed and the caller is not the owner of the directory to be removed, nor is the caller the owner of the parent directory, nor does the caller have the appropriate privileges.

EROFS
The directory to be removed is on a read-only file system.

Example

CELEBR16

```
/* CELEBR16

   This example removes a directory.

 */
#define _OPEN_SYS
#include <fcntl.h>
#include <sys/stat.h>
#include <sys/stat.h>
#include <unistd.h>
#undef _POSIX_SOURCE
#include <stdio.h>

main() {
  char new_dir[]="new_dir";
  char new_file[]="new_dir/new_file";
  int  fd;

  if (mkdir(new_dir, S_IRWXU|S_IRGRP|S_IXGRP) != 0)
    perror("mkdir() error");
  else if ((fd = creat(new_file, S_IWUSR)) < 0)
    perror("creat() error");
  else {
    close(fd);
    unlink(new_file);
  }

  if (rmdir(new_dir) != 0)
    perror("rmdir() error");
  else
    puts("removed!");
}
```

Related information

- “unistd.h” on page 82
- “mkdir() — Make a directory” on page 1068
- “unlink() — Remove a directory entry” on page 1962

round(), roundf(), roundl() — Round to the nearest integer

Standards

Standards / Extensions	C or C++	Dependencies
C99 Single UNIX Specification, Version 3 C++ TR1 C99	both	z/OS V1R7

Format

```
#define _ISOC99_SOURCE
#include <math.h>

double round(double x);
float roundf(float x);
long double roundl(long double x);
```

C++ TR1 C99

```
#define _TR1_C99
#include <math.h>

float round(float x);
long double round(long double x);
```

General description

The round() family of functions round *x* to the nearest integer, in floating-point format and rounding halfway cases away from zero, regardless of the current rounding mode.

Note: The following table shows the viable formats for these functions. See "IEEE binary floating-point" on page 94 for more information about IEEE Binary Floating-Point.

Function	Hex	IEEE
round	X	X
roundf	X	X
roundl	X	X

Returned value

The round() family of functions returns the rounded integer value.

Related information

- "math.h" on page 44
- "ceil(), ceilf(), ceill() — Round up to integral value" on page 249
- "floor(), floorf(), floorl() — Round down to integral value" on page 548
- "llround(), llroundf(), llroundl() — Round to the nearest integer" on page 975
- "lrint(), lrintf(), lrintl() and llrint(), llrintf(), llrintl() — Round the argument to the nearest integer" on page 1012
- "lround(), lroundf(), lroundl() — Round a decimal floating-point number to its nearest integer" on page 1016
- "nearbyint(), nearbyintf(), nearbyintl() — Round the argument to the nearest integer" on page 1132
- "trunc(), truncf(), truncl() — Truncate an integer value" on page 1915

roundd32(), roundd64(), roundd128() — Round to the nearest integer

Standards

Standards / Extensions	C or C++	Dependencies
C/C++ DFP	both	z/OS V1.8

Format

```
#define __STDC_WANT_DEC_FP__
#include <math.h>

_Decimal32 roundd32(_Decimal32 x);
_Decimal64 roundd64(_Decimal64 x);
_Decimal128 roundd128(_Decimal128 x);

_Decimal32 round(_Decimal32 x);      /* C++ only */
_Decimal64 round(_Decimal64 x);      /* C++ only */
_Decimal128 round(_Decimal128 x);    /* C++ only */
```

General description

These functions round *x* to the nearest integer, in decimal floating-point format and rounding halfway cases away from zero, regardless of the current rounding mode.

Notes:

1. To use IEEE decimal floating-point, the hardware must have the Decimal Floating-Point Facility installed.
2. These functions work in IEEE decimal floating-point format. See "IEEE Decimal Floating-Point" for more information.

Returned value

These functions return the rounded integer value.

Example

```
/* CELEBR22

   This example illustrates the round64() function.

*/

#pragma strings(readonly)

#define __STDC_WANT_DEC_FP__
#include <fenv.h>
#include <math.h>
#include <stdio.h>

static void try_rm(int, _Decimal64);

/* pass back printable rounding mode */

static
char *rm_str(int rm)
{
  char *s = "undetermined";

  switch (rm)
  {
    case    FE_DEC_TONEAREST            :
      s =  "FE_DEC_TONEAREST"           ;  break;
    case    FE_DEC_TOWARDZERO           :
      s =  "FE_DEC_TOWARDZERO"          ;  break;
    case    FE_DEC_UPWARD               :
      s =  "FE_DEC_UPWARD"              ;  break;
    case    FE_DEC_DOWNWARD             :
      s =  "FE_DEC_DOWNWARD"            ;  break;
    case    FE_DEC_TONEARESTFROMZERO    :
      s =  "FE_DEC_TONEARESTFROMZERO"   ;  break;
```

```
     case   _FE_DEC_TONEARESTTOWARDZERO  :
       s = "_FE_DEC_TONEARESTTOWARDZERO" ;  break;
     case   _FE_DEC_AWAYFROMZERO         :
       s = "_FE_DEC_AWAYFROMZERO"        ;  break;
     case   _FE_DEC_PREPAREFORSHORTER    :
       s = "_FE_DEC_PREPAREFORSHORTER"   ;  break;
    }

    return s;
}


/* Try out one passed-in number with rounding mode */

static void try_rm(int rm, _Decimal64 d64)
{
  _Decimal64 r64;

  (void)fe_dec_setround(rm);

  r64 = roundd64(d64);

  printf("roundd64(%+.2DF)  = %+DG - rounding mode = %s\n",
         d64 , r64, rm_str(rm)
        );

  return;
}


int main()
{
  try_rm( FE_DEC_TONEAREST          ,  501.50DD);
  try_rm( FE_DEC_TOWARDZERO         ,  501.50DD);
  try_rm( FE_DEC_UPWARD             , -501.51DD);
  try_rm( FE_DEC_DOWNWARD           , -501.49DD);
  try_rm( FE_DEC_TONEARESTFROMZERO  ,  500.50DD);
  try_rm(_FE_DEC_TONEARESTTOWARDZERO, -501.50DD);
  try_rm(_FE_DEC_AWAYFROMZERO       ,  500.49DD);
  try_rm(_FE_DEC_PREPAREFORSHORTER  ,  501.50DD);

  return 0;
}
```

Related information

- “math.h” on page 44
- “ceild32(), ceild64(), ceild128() — Round up to integral value” on page 250
- “floord32(), floord64(), floord128() — Round down to integral value” on page 549
- “llroundd32(), llroundd64(), llroundd128() — Round to the nearest integer” on page 977
- “lrintd32(), lrintd64(), lrintd128() and llrintd32(), llrintd64(), llrintd128() — Round the argument to the nearest integer” on page 1014
- “lroundd32(), lroundd64(), lroundd128() — Round a floating-point number to its nearest integer” on page 1017
- “nearbyintd32(), nearbyintd64(), nearbyintd128() — Round the argument to the nearest integer” on page 1134
- “round(), roundf(), roundl() — Round to the nearest integer” on page 1455
- “truncd32(), truncd64(), truncd128() — Truncate an integer value” on page 1916

rpmatch() — Test for a yes or no response match

Standards

Standards / Extensions	C or C++	Dependencies
z/OS UNIX	both	

Format

```
#include <stdlib.h>

int rpmatch(const char *response);
```

External entry point: @@RPMTCH, __rpmtch

General description

Tests whether a string pointed to by *response* matches either the affirmative or the negative response set by LC_MESSAGES category in the current locale.

To avoid infringing on the user's name space, this nonstandard function has two names. One name is prefixed with two underscore characters, and one name is not. The name without the prefix underscore characters is exposed only when you use LANGLVL(EXTENDED).

To use this function, you must either invoke the function using its external entry point name (that is, the name that begins with two underscore characters), or compile with LANGLVL(EXTENDED). When you use LANGLVL(EXTENDED) any relevant information in the header is also exposed.

Returned value

If the string pointed to by *response* matches the affirmative expression in the current locale, rpmatch() returns:

1 If the response string matches the affirmative expression.

0 If the response string matches the negative expression.

-1 If the response string does not match either the affirmative or the negative expression.

Example

CELEBR17

```
/* CELEBR17

   This example asks for a reply, and checks the response.

 */
#include "locale.h"
#include "stdio.h"
#include "stdlib.h"

main() {
   char *response;
   char  buffer??(100??);
   int   rc;

   printf("Enter reply");
   response = fgets(buffer, 100, stdin);
   rc = rpmatch(response);
   if (rc > 0)
      printf("Response was affirmative\n");
   else if (rc == 0)
      printf("Response was negative\n");
   else
      printf("Response was neither negative or affirmative\n");
}
```

Related information

- "stdlib.h" on page 70

samequantumd32(), samequantumd64(), samequantumd128() — Determine if exponents X and Y are the same

Standards

Standards / Extensions	C or C++	Dependencies
C/C++ DFP	both	z/OS V1.8

Format

```
#define __STDC_WANT_DEC_FP__
#include <math.h>

_Bool samequantumd32(_Decimal32 x, _Decimal32 y);
_Bool samequantumd64(_Decimal64 x, _Decimal64 y);
_Bool samequantumd128(_Decimal128 x, _Decimal128 y);
```

General description

The samequantum functions determine if the representation exponents of x and y are the same. If both x and y are NaN or infinity, they have the same representation exponents. If exactly one operand is infinity or exactly one operand is NaN, they do not have the same representation exponents. The samequantum functions raise no floating point exceptions.

Argument	Description
x	First input value
y	Second input value

Note:

- To use IEEE decimal floating-point, the hardware must have the Decimal Floating-Point Facility installed.
- These functions work in IEEE decimal floating-point format. See "IEEE Decimal Floating-Point" for more information.

Returned value

The samequantum functions return true when x and y have the same representation exponents, and false otherwise.

Example

```
/* CELEBS72

   This example illustrates the samequantumd64() function

*/

#define  __STDC_WANT_DEC_FP__
#include <math.h>
#include <stdio.h>
#include <stdlib.h>

int main(void)
{
  _Decimal64 a1 = strtod64("1.23"  , NULL);
  _Decimal64 a2 = strtod64("0.01"  , NULL);
  _Decimal64 b1 = strtod64("1.234" , NULL);
  _Decimal64 b2 = strtod64("0.01"  , NULL);
  _Decimal64 c1 = strtod64("1.000" , NULL);
  _Decimal64 c2 = strtod64("1.00"  , NULL);
  _Decimal64 d1 = strtod64("0.000" , NULL);
  _Decimal64 d2 = strtod64("0.00"  , NULL);

  printf( "x=%-8.2DF y=%-8.2DF samequantum=%d\n"
          "x=%-8.3DF y=%-8.2DF samequantum=%d\n"
          "x=%-8.3DF y=%-8.2DF samequantum=%d\n"
          "x=%-8.3DF y=%-8.2DF samequantum=%d\n"

        ,  a1, a2, (int)samequantumd64(a1, a2)
        ,  b1, b2, (int)samequantumd64(b1, b2)
        ,  c1, c2, (int)samequantumd64(c1, c2)
        ,  d1, d2, (int)samequantumd64(d1, d2)
        );

  return 0;
}
```

Related information

- “math.h” on page 44
- “quantized32(), quantized64(), quantized128() — Set the exponent of X to the exponent of Y” on page 1366
- “quantexpd32(), quantexpd64(), quantexpd128() - Compute the quantum exponent” on page 1365

sbrk() — Change space allocation

Standards

Standards / Extensions	C or C++	Dependencies
XPG4.2 Single UNIX Specification, Version 2	both	

Format

Non-Single UNIX Specification, Version 2:

```
#define _XOPEN_SOURCE_EXTENDED 1
#include <unistd.h>

void *sbrk(int incr);
```

Single UNIX Specification, Version 2:

```
#define _XOPEN_SOURCE 500
#include <unistd.h>

void *sbrk(intptr_t incr);
```

General description

Restriction: This function is not supported in AMODE 64.

The sbrk() function is used to change the space allocated for the calling process. The change is made by adding incr bytes to the process's break value and allocating the appropriate amount of space. The amount of allocated space increases when incr is positive and decreases when incr is negative. If incr is zero the current value of the program break is returned by sbrk(). The newly-allocated space is set to 0. However, if the application first decrements and then increments the break value, the contents of the reallocated space are not zeroed.

The storage space from which the brk() and sbrk() functions allocate storage is separate from the storage space that is used by the other memory allocation functions (malloc(), calloc(), etc.). Because this storage space must be a contiguous segment of storage, it is allocated from the initial heap segment only and thus is limited to the initial heap size specified for the calling program or the largest contiguous segment of storage available in the initial heap at the time of the first brk() or sbrk() call. Since this is a separate segment of storage, the brk() and sbrk() functions can be used by an application that is using the other memory allocation functions. However, it is possible that the user's region may not be large enough to support extensive usage of both types of memory allocation.

Prior usage of the sbrk() function has been limited to specialized cases where no other memory allocation function performed the same function. Because the sbrk() function may be unable to sufficiently increase the space allocation of the process when the calling application is using other memory functions, the use of other memory allocation functions, such as mmap(), is now preferred because it can be used portably with all other memory allocation functions and with any function that uses other allocation functions. Applications that require the use of brk() and/or sbrk() should refrain from using the other memory allocation functions and should be run with an initial heap size that will satisfy the maximum storage requirements of the program.

The sbrk() function is not supported from a multithreaded environment, it will return in error if it is invoked in this environment.

Note:

This function is kept for historical reasons. It was part of the Legacy Feature in Single UNIX Specification, Version 2, but has been withdrawn and is not supported as part of Single UNIX Specification, Version 3. New applications should use malloc() instead of brk() or sbrk().

If it is necessary to continue using this function in an application written for Single UNIX Specification, Version 3, define the feature test macro _UNIX03_WITHDRAWN before including any standard system headers. The macro exposes all interfaces and symbols removed in Single UNIX Specification, Version 3.

Returned value

If successful, sbrk() returns the previous break value.

If unsuccessful, sbrk() returns -1 and sets errno to one of the following values:

Error Code
Description

ENOMEM
The requested change would allocate more space than allowed for the calling process.

Related information

- “unistd.h” on page 82
- “brk() — Change space allocation” on page 217

scalb() — Load exponent

Standards

Standards / Extensions	C or C++	Dependencies
XPG4.2 Single UNIX Specification, Version 3	both	

Format

```
#define _XOPEN_SOURCE_EXTENDED 1
#include <math.h>

double scalb(double x, double n);
```

General description

The scalb() function computes
.

$$x \bullet radix^{n}$$

If *n* is not an integer, it is silently truncated.

Note: This function works in both IEEE Binary Floating-Point and hexadecimal floating-point formats. The *radix* is 16 for hexadecimal floating-point and 2 for IEEE Binary Floating-Point. See “IEEE binary floating-point” on page 94 for more information about IEEE Binary Floating-Point.

Returned value

If it succeeds, scalb() returns the function of its arguments as described above.

scalb() will fail under the following conditions:

- If the result would underflow, scalb() will return 0 and set errno to ERANGE.
- If the result would overflow, scalb() will return ±HUGE_VAL according to the sign of *x* and set errno to ERANGE.

Special behavior for IEEE: If successful, scalb() returns the value of the *x* parameter times 2 to the power of the *y* parameter.

If the result would overflow, scalb() returns ±HUGE_VAL according to the sign of *x* and sets errno to ERANGE. No other errors can occur.

Related information

- "math.h" on page 44
- "ldexp(), ldexpf(), ldexpl() — Multiply by a power of two" on page 941

scalbn(), scalbnf(), scalbnl(), scalbln(), scalblnf(), scalblnl() — Load exponent functions

Standards

Standards / Extensions	C or C++	Dependencies
C99 Single UNIX Specification, Version 3 C++ TR1 C99	both	z/OS V1R7

Format

```
#define _ISOC99_SOURCE
#include <math.h>

double scalbn(double x, int n);
float scalbnf(float x, int n);
long double scalbnl(long double x, int n);

double scalbln(double x, long int n);
float scalblnf(float x, long int n);
long double scalblnl(long double x, long int n);
```

C++ TR1 C99

```
#define _TR1_C99
#include <math.h>

float scalbln(float x, long n);
long double scalbln(long double x, long n);
float scalbln(float x, long int n);
long double scalbln(long double x, long int n);
```

General description

The scalbn() and scalbln() families of functions compute (*x* * (**FLT_RADIX**) raised to *n*) efficiently, not normally, by computing **FLT_RADIX** raised to *n* explicitly.

The radix for z/OS C applications, **FLT_RADIX**, is defined to be 16 under HEX implementation and 2 under IEEE implementation.

Note: The following table shows the viable formats for these functions. See "IEEE binary floating-point" on page 94 for more information about IEEE Binary Floating-Point.

Function	Hex	IEEE
scalbn	X	X
scalbnf	X	X
scalbnl	X	X
scalbln	X	X

Function	Hex	IEEE
scalblnf	X	X
scalblnl	X	X

Restriction: The scalbnf() and scalblnf() functions do not support the _FP_MODE_VARIABLE feature test macro.

Returned value

The scalbn() and scalbln() families of functions return (*x* * (**FLT_RADIX**) raised to *n*).

Related information

- “math.h” on page 44
- “exp(), expf(), expl() — Calculate exponential function” on page 448
- “expm1(), expm1f(), expm1l() — Exponential minus one” on page 450
- “exp2(), exp2f(), exp2l() — Calculate the base-2 exponential” on page 454
- “frexp(), frexpf(), frexpl() — Extract mantissa and exponent of the floating-point value” on page 623
- “ilogb(), ilogbf(), ilogbl() — Integer unbiased exponent” on page 835
- “ldexp(), ldexpf(), ldexpl() — Multiply by a power of two” on page 941
- “log(), logf(), logl() — Calculate natural logarithm” on page 990
- “logb(), logbf(), logbl() — Unbiased exponent” on page 991
- “log1p(), log1pf(), log1pl() — Natural log of x+1” on page 998
- “log10(), log10f(), log10l() — Calculate base 10 logarithm” on page 1000
- “log2(), log2f(), log2l() — Calculate the base-2 logarithm” on page 1003
- “modf(), modff(), modfl() — Extract fractional and integral parts of floating-point value” on page 1086

scalbnd32(), scalbnd64(), scalbnd128() and scalblnd32(), scalblnd64(), scalblnd128() — Load exponent functions

Standards

Standards / Extensions	C or C++	Dependencies
C/C++ DFP	both	z/OS V1.8

Format

```
#define __STDC_WANT_DEC_FP__
#include <math.h>

_Decimal32  scalbnd32(_Decimal32 x, int n);
_Decimal64  scalbnd64(_Decimal64 x, int n);
_Decimal128 scalbnd128(_Decimal128 x, int n);
_Decimal32  scalbn(_Decimal32 x, int n);    /* C++ only */
_Decimal64  scalbn(_Decimal64 x, int n);    /* C++ only */
_Decimal128 scalbn(_Decimal128 x, int n);   /* C++ only */

_Decimal32  scalblnd32(_Decimal32 x, long int n);
_Decimal64  scalblnd64(_Decimal64 x, long int n);
_Decimal128 scalblnd128(_Decimal128 x, long int n);
_Decimal32  scalbln(_Decimal32 x, long int n);    /* C++ only */
_Decimal64  scalbln(_Decimal64 x, long int n);    /* C++ only */
_Decimal128 scalbln(_Decimal128 x, long int n)    /* C++ only */
```

General description

The scalbn() and scalbln() families of functions compute (x * 10 raised to n) efficiently, not normally, by computing 10 raised to n explicitly.

Notes:

1. To use IEEE decimal floating-point, the hardware must have the Decimal Floating-Point Facility installed.
2. These functions work in IEEE decimal floating-point format. See "IEEE Decimal Floating-Point" for more information.

Returned value

The scalbn() and scalbln() families of functions return (x * 10 raised to n).

Example

```
/* CELEBS68

   This example illustrates the scalbnd128() function.

*/

#define __STDC_WANT_DEC_FP__
#include <math.h>
#include <stdio.h>

int main(void)
{
   _Decimal128 x, y;

   x = 7.2DL;
   y = scalbnd128(x, 6000);

   printf("scalbnd128(%DDf, 6000) = %DDe\n", x, y);
}
```

Related information

- “math.h” on page 44
- “expd32(), expd64(), expd128() — Calculate exponential function” on page 449
- “frexpd32(), frexpd64(), frexpd128() — Extract mantissa and exponent of the decimal floating-point value” on page 624
- “ilogbd32(), ilogbd64(), ilogbd128() — Integer unbiased exponent” on page 837
- “ldexpd32(), ldexpd64(), ldexpd128() — Multiply by a power of ten” on page 942
- “logd32(), logd64(), logd128() — Calculate natural logarithm” on page 994
- “log10d32(), log10d64(), log10d128() — Calculate base 10 logarithm” on page 1002
- “modfd32(), modfd64(), modfd128() — Extract fractional and integral parts of decimal floating-point value” on page 1087
- “scalbn(), scalbnf(), scalbnl(), scalbln(), scalblnf(), scalblnl() — Load exponent functions” on page 1464

scanf() — Read and format data

The information for this function is included in “fscanf(), scanf(), sscanf() — Read and format data” on page 625.

sched_yield() — Release the processor to other threads

Standards

Standards / Extensions	C or C++	Dependencies
Single UNIX Specification, version 3	both	z/OS V1R7 POSIX(ON)

Format

```
#define _UNIX03_SOURCE
#include <sched.h>

int sched_yield(void);
```

General description

The sched_yield() function allows a thread to give up control of a processor so that another thread can have the opportunity to run. It takes no arguments.

The speed at which the sched_yield() function releases a processor can be configured by using the _EDC_PTHREAD_YIELD and _EDC_PTHREAD_YIELD_MAX environment variables. The _EDC_PTHREAD_YIELD environment variable is used to configure the sched_yield() function to release the processor immediately, or to release the processor after a delay. The _EDC_PTHREAD_YIELD_MAX environment variable is used to change the maximum delay to a value less than the default (32 milliseconds).

For more information about the _EDC_PTHREAD_YIELD and _EDC_PTHREAD_YIELD_MAX environment variables, see “Using Environment Variables” in *z/OS XL C/C++ Programming Guide*.

Returned value

sched_yield() always returns 0.

There are no documented errno values. Use perror() or strerror() to determine the cause of the error.

Related information

- “sched.h” on page 62
- “pthread_yield() — Release the processor to other threads” on page 1344

seed48() — Pseudo-random number initializer

Standards

Standards / Extensions	C or C++	Dependencies
XPG4 XPG4.2 Single UNIX Specification, Version 3	both	

Format

```
#define _XOPEN_SOURCE
#include <stdlib.h>

unsigned short int *seed48(unsigned short int seed16v[3]);
```

General description

The drand48(), erand48(), jrand48(), lrand48(), mrand48() and nrand48() functions generate uniformly distributed pseudo-random numbers using a linear congruential algorithm and 48-bit integer arithmetic.

The lcong48(), seed48(), and srand48() functions are initialization functions, one of which should be invoked before either the drand48(), lrand48() or mrand48() function is called.

The drand48(), lrand48() and mrand48() functions generate a sequence of 48-bit integer values, X(i), according to the linear congruential formula:

```
X(n+1) = (aX(n) + c)mod(2**48)     n>=0
```

The initial values of X, a, and c are:

```
X(0)= 1
a   = 5deece66d (base 16)
c   = b         (base 16)
```

C/370 provides storage to save the most recent 48-bit integer value of the sequence, X(i). This storage is shared by the drand48(), lrand48() and mrand48() functions. The seed48() function is used to reinitialize the most recent 48-bit value in this storage. The seed48() function replaces the low-order (rightmost) 16 bits of this storage with *seed16v*[0], the middle-order 16 bits with *seed16v*[1], and the high-order 16 bits with *seed16v*[2].

The values a and c, may be changed by calling the lcong48() function. The seed48() function restores the initial values of a and c.

Special behavior for z/OS UNIX Services: You can make the seed48() function and other functions in the drand48 family thread-specific by setting the environment variable _RAND48 to the value THREAD before calling any function in the drand48 family.

If you do not request thread-specific behavior for the drand48 family, C/370 serializes access to the storage for X(n), a and c by functions in the drand48 family when they are called by a multithreaded application.

If thread-specific behavior is requested, calls to the drand48(), lrand48() and mrand48() functions from thread t generate a sequence of 48-bit integer values, X(t,i), according to the linear congruential formula:

```
X(t,n+1) = (a(t)X(t,n) + c(t))mod(2**48)     n>=0
```

C/370 provides thread-specific storage to save the most recent 48-bit integer value of the sequence, X(t,i). When the seed48()function is called from thread t, it reinitializes the most recent 48-bit value in this storage. The seed48() function replaces the low-order (rightmost) 16 bits of this storage with *seed16v*[0], the middle-order 16 bits with *seed16v*[1], and the high-order 16 bits with *seed16v*[2].

The values of a(t) and c(t) may be changed by calling the lcong48() function from thread `t`. When the seed48()function is called from this thread, it restores the initial values of a(t) and c(t) for the thread which are:

```
a(t)   = 5deece66d (base 16)
c(t)   = b         (base 16)
```

Returned value

When seed48() is called, it saves the most recent 48-bit integer value in the sequence, X(i), in an array of unsigned short ints provided by C/370 before reinitializing storage for the most recent value in the sequence, X(i). seed48() returns a pointer to the array containing the saved value.

Special behavior for z/OS UNIX Services: If thread-specific behavior is requested for the drand48 family and seed48() is called on thread `t`, it saves the most recent 48-bit integer value in the sequence, X(t,i), for the thread in a thread-specific array of unsigned short ints before reinitializing storage for the most recent value in the sequence, X(t,i). seed48() returns a pointer to this thread-specific array containing the saved value.

Related information

- “stdlib.h” on page 70
- “drand48() — Pseudo-random number generator” on page 402
- “erand48() — Pseudo-random number generator” on page 426
- “jrand48() — Pseudo-random number generator” on page 928
- “lcong48() — Pseudo-random number initializer” on page 939
- “lrand48() — Pseudo-random number generator” on page 1010
- “mrand48() — Pseudo-random number generator” on page 1101
- “nrand48() — Pseudo-random number generator” on page 1148
- “srand48() — Pseudo-random number initializer” on page 1718

seekdir() — Set position of directory stream

Standards

Standards / Extensions	C or C++	Dependencies
XPG4 XPG4.2 Single UNIX Specification, Version 3	both	

Format

```
#define _XOPEN_SOURCE
#include <dirent.h>

void seekdir(DIR *dirp, long int loc);
```

General description

The seekdir() function sets the position of the next readdir() operation on the directory stream specified by *dirp* to the position specified by *loc*. The value of *loc* should have been returned from an earlier call to telldir(). The new position reverts to the one associated with the directory stream when telldir() was performed. If the value of *loc* was not obtained from an earlier call to telldir() or if a call to rewinddir() occurred between the call to telldir() and the call to seekdir(), the result of subsequent calls to readdir() are unspecified.

Note: If files were added or removed from the directory after telldir() was called and before seekdir() is done, the results are also unspecified.

Returned value

seekdir() returns no values.

If the *loc* argument is negative, the directory stream is unchanged.

Related information

- “dirent.h” on page 22
- “stdio.h” on page 68
- “sys/types.h” on page 75
- “closedir() — Close a directory” on page 291
- “opendir() — Open a directory” on page 1158
- “readdir() — Read an entry from a directory” on page 1382
- “rewinddir() — Reposition a directory stream to the beginning” on page 1446
- “telldir() — Current location of directory stream” on page 1866

select(), pselect() — Monitor activity on files or sockets and message queues

Standards

Standards / Extensions	C or C++	Dependencies
XPG4.2 Single UNIX Specification, Version 3	both	

Format

X/Open:

```
#define _XOPEN_SOURCE_EXTENDED 1
#define _OPEN_MSGQ_EXT
#include <sys/types.h>
#include <sys/time.h>
#include <sys/msg.h>

int select(int nmsgsfds, fd_set *__restrict__ readlist,
           fd_set *__restrict__ writelist, fd_set *__restrict__ exceptlist,
           struct timeval *__restrict__ timeout);
```

SUSV3:

```
#define _POSIX_C_SOURCE 200112L
#include <sys/select.h>

int pselect(int nmsgsfds, fd_set *__restrict__ readlist,
           fd_set *__restrict__ writelist, fd_set *__restrict__ exceptlist,
           const struct timespec *__restrict__ timeout,
           const sigset *__restrict__ sigmask);
```

Berkeley sockets:

```
#define _OE_SOCKETS
#define _OPEN_MSGQ_EXT
#include <sys/types.h>
#include <sys/time.h>
#include <sys/msg.h>
```

```
int select(int nmsgsfds, fd_set *readlist,
           fd_set *writelist, fd_set *exceptlist,
           struct timeval *timeout);
```

_OPEN_MSGQ_EXT must be defined if message queues are to be monitored (X/Open sockets only).

General description

The pselect() and select() functions monitor activity on a set of sockets and/or a set of message queue identifiers until a timeout occurs, to see if any of the sockets and message queues have read, write, or exception processing conditions pending. This call also works with regular file descriptors, pipes, and terminals.

The select() function is equivalent to the pselect() function, except as follows:

- For the select() function, the timeout period is given in seconds and microseconds in an argument of type struct *timeval*, whereas for the pselect() function the timeout period is given in seconds and nanoseconds in an argument of type struct *timespec*.
- The select() function has no *sigmask* argument; it will behave as pselect() does when *sigmask* is a null pointer.
- Upon successful completion, the select() function can modify the object pointed to by the timeout argument.
- The pselect() function always behaves as if _OPEN_MSGQ_EXT and _OPEN_SYS_HIGH_DESCRIPTORS feature test macros are NOT defined.

Parameter
Description

nmsgsfds
The number of message queues and the number of file or socket descriptors to check.

This parameter is divided into two parts. The first half (the high-order 16 bits) gives the number of elements of an array that contains message queue identifiers. This number must not exceed the value 32767.

The second half (the low-order 16 bits) gives the number of bits within a bit set that correspond to the file or socket descriptors to check. This value should equal the greatest descriptor number to check + 1.

If either half of the *nmsgsfds* parameter is equal to a value of 0, the corresponding bit sets or arrays are assumed not to be present.

If _OPEN_MSGQ_EXT is not defined, only file or socket descriptors may be monitored. In this case *nmsgsfds* must be less than or equal to FD_SETSIZE (defined to be 2048 in sys/time.h), and greater than or equal to zero. Also, FD_SETSIZE may not be defined by your program.

The bit set used to specify file or socket descriptors is fixed in size with 1 bit for every possible file or socket. Use the *nmsgsfds* parameter to force pselect() or select() to check only a subset of the allocated bit set.

If your application allocates sockets 3, 4, 5, 6, and 7 and you want to check all of your allocations, the second half of *nmsgsfds* should be set to 8, the highest descriptor you specified + 1. If your application checks sockets 3 and 4, the second half of *nmsgsfds* should be set to 5.

To select on descriptor numbers between 2048 and 65534, either the _OPEN_MSGQ_EXT or _OPEN_SYS_HIGH_DESCRIPTORS feature test macro must be defined, and a bit set larger than the default size must be used. Note that when you are also selecting on message queues, as is possible when _OPEN_MSGQ_EXT is defined, the largest descriptor number is restricted to 2047. To select on descriptor numbers between 65535 and 524287, feature test macro _OPEN_SYS_HIGH_DESCRIPTORS must be defined and feature test macro _OPEN_MSGQ_EXT must not be defined. In addition, the process' MAXFILEPROC limit must be greater than 65536. With this feature, any number of sockets can be selected on (without message queues). FD_SETSIZE may also be redefined in this case, though it is recommended that the application explicitly allocate the larger bit set using malloc().

readlist,writelist,exceptlist

Pointers to fd_set types, arrays of message queue identifiers, or sellist structures to check for reading, writing, and exceptional conditions, respectively. The type of parameter to pass depends on whether you want to monitor file/socket descriptors, message queue identifiers, or both. To monitor file/socket descriptors only, set the high-order halfword of *nmsgsfds* to 0, the low-order halfword to (highest descriptor number + 1), and use fd_set pointers. To monitor message queues only, set the low-order halfword of *nmsgsfds* to 0, the high-order halfword to the number of elements in each array you want select() to consider, and pass pointers to arrays of message queue identifiers. To monitor both, set *nmsgsfds* as described above, and pass pointers to sellist structures.

The **sellist** structure allows you to specify both file/socket descriptors and message queues. Your program must define the **sellist** structure in the following form:

```
struct sellist {
   fd_set fdset;           /* file/socket descriptor bit set */
   int    msgids[max_size]; /* array of message queue identifiers */
   };
```

If you use a sellist structure, the highest descriptor you can monitor is 2047.

The description of the type *fd_set* is given below. Each integer of the msgids array specifies a message queue identifier whose status is to be checked. Elements with a value of -1 are acceptable and will be ignored. The value contained in the first half of *nmsgsfds* determines exactly how many elements of the array are to be checked.

timeout

The pointer to the time to wait for the pselect() or select() call to complete.

sigmask

The signal mask of the caller by the set of signals pointed to by *sigmask* before examining the descriptors, and will restore the signal mask of the calling thread before returning.

If *timeout* is not a NULL pointer, it specifies a maximum interval to wait for the selection to complete. The maximum timeout value is 31 days. If *timeout* is a NULL pointer, the pselect() and select() call blocks until a socket or message becomes ready. To poll the sockets and return immediately, *timeout* should be a non-NULL pointer to a zero-valued **timeval** structure or timespec structure.

If *sigmask* is not a null pointer, then the pselect() function will replace the signal mask of the caller by the set of signals pointed to by *sigmask* before examining the descriptors, and will restore the signal mask of the calling thread before returning.

To allow you to test more than one socket at a time, the sockets to test are placed into a bit set of type *fd_set*. A bit set is a string of bits such that if *x* is an element of the set, the bit representing *x* is set to 1. If *x* is not an element of the set, the bit representing *x* is set to 0. For example, if socket 33 is an element of a bit set, then bit 33 is set to 1. If socket 33 is not an element of a bit set, then bit 33 is set to 0.

Because the bit sets contain a bit for every socket that a process can allocate, the size of the bit sets is constant. If your program needs to allocate a large number of sockets, you may need to increase the size of the bit sets. Increasing the size of the bit sets should be done when you compile the program. To increase the size of the bit sets, define FD_SETSIZE before including **sys/time.h**. FD_SETSIZE is the largest value of any socket that your program expects to use pselect() or select() on. It is defined to be 2048 in **sys/time.h**.

Note: FD_SETSIZE may only be defined by the application program if the extended version of select() is used (by defining _OPEN_MSGQ_EXT). Do NOT define FD_SETSIZE in your program if a sellist structure will be used.

Note: The z/OS UNIX POSIX.1 implementation allows you to control the maximum number of open descriptors allowed per process. This maximum possible value is 524288. If your application program requires a large number of either socket or file descriptors, you should protect your code from possible runtime errors by:

- Adding a check before your pselect(), select() or selectex() calls to see if the bit set size contained in *nmsgsfds* is larger than FD_SETSIZE.
- Dynamically allocate bit strings large enough to hold the largest descriptor value in your application program, rather than rely on the static bit strings created at compile time. When allocating your own bit strings, use malloc() to define an area large enough to represent each bit, rounded up to the next 4-byte multiple. For example, if your largest descriptor value is 31, you need 4 bytes; if your largest descriptor is 32, you need 8 bytes.
- If you dynamically allocate your own bit strings, the FD_ZERO() macro will *not* work. The application must zero that storage, by using the memset function—that is, memset(*ptr,0,mallocsize*). The other macros can be used with the dynamically allocated bit strings, as long as the descriptor you are manipulating is within the bit string. If the descriptor number is larger than the bit string, unpredictable results can occur.

The application program must make sure that the parameters *readlist*, *writelist*, and *exceptlist* point to bit strings that are as large as the bit string size in parameter *nmsgsfds* z/OS UNIX services will try to access bits 0 through *n*-1 (where *n* = the value of the second halfword of *nmsgsfds*), for each of the bit strings. If the bit strings are too short, you will receive unpredictable results when you run your application program.

The following macros are provided to manipulate bit sets.

Macro Description

FD_ZERO(&*fdset*)
Sets all bits in the bit set *fdset* to zero. After this operation, the bit set does

not contain sockets as elements. This macro should be called to initialize the bit set before calling FD_SET() to set a socket as a member.

Note: If you used malloc() to dynamically allocate a new area, the FD_ZERO() macro can cause unpredictable results and should *not* be used. You should zero the area using the memset() function.

FD_SET(*sock***, &***fdset***)**
Sets the bit for the socket *sock* to a 1, making *sock* a member of the bit set *fdset*.

FD_CLR(*sock***, &***fdset***)**
Clears the bit for the socket *sock* in bit set *fdset*. This operation sets the appropriate bit to a zero.

FD_ISSET(*sock***, &***fdset***)**
Returns nonzero if *sock* is a member of the bit set *fdset*. Returns 0 if *sock* is not a member of *fdset*. (This operation returns the bit representing *sock*.)

The following macros are provided to manipulate the *nmsgsfds* parameter and the return value from pselect() and select():

Macro Description

_SET_FDS_MSGS(*nmsgsfds, nmsgs, nfds***)**
Sets the high-order halfword of *nmsgsfds* to *nmsgs*, and sets the low-order halfword of *nmsgsfds* to *nfds*.

_NFDS(*n***)**
If the return value *n* from pselect() or select() is nonnegative, returns the number of descriptors that meet the read, write, and exception criteria. A descriptor may be counted multiple times if it meets more than one given criterion.

_NMSGS(*n***)**
If the return value *n* from pselect() or select() is nonnegative, returns the number of message queues that meet the read, write, and exception criteria. A message queue may be counted multiple times if it meets more than one given criterion.

A socket is ready for reading when incoming data is buffered for it or when a connection request is pending. To test whether any sockets are ready for reading, use either FD_ZERO() or memset(), if the function was dynamically allocated, to initialize the fdset bit set in *readlist* and invoke FD_SET() for each socket to test.

A socket is ready for writing if there is buffer space for outgoing data. A socket is ready for reading if there is data on the socket to be received. For a nonblocking stream socket in the process of connecting the connect() will return with a -1. The program needs to check the errno. If the errno is EINPROGRESS, the socket is selected for write when the connect() completes. In the situation where the errno is not EINPROGRESS, the socket will still be selected for write which indicates that there is a pending error on the socket. Acall to write(), send(), or sendto() does not block provided that the amount of data is less than the amount of buffer space. If a socket is selected for write, the amount of available buffer space is guaranteed to be at least as large as the size returned from using SO_SNDBUF with getsockopt(). To test whether any sockets are ready for writing, initialize the fdset bit set in *writelist* with either FD_ZERO() or memset(), if dynamically allocated, and use FD_SET() for each socket to test.

A message queue is ready for reading when any time it has a message on it. It is considered ready for writing when any time it is not full. A message queue is full when it has either reached its number of messages limit or its number of bytes limit. An exception condition exists when a message queue is deleted while a select() caller is waiting on the queue.

The programmer can pass NULL for any of the *readlist*, *writelist*, and *exceptlist* parameters. However, when they are not NULL, they must all point to the same type of structures. For example, suppose the *readlist* points to a sellist. If the *writelist* is not NULL, it must point to a sellist also. Now, let us say the *writelist* is not NULL. If the programmer wants to check a set of file descriptors for read status only, the appropriate bits in the bit set in the sellist structure pointed to by the *writelist* must be set to 0. If the programmer wants to check a set of message queues for write status only, the appropriate elements in the array in the sellist structure pointed to by the *readlist* must be set to -1. Regular files are always ready for reading and writing.

Because the sets of sockets passed to pselect() and select() are bit sets, the pselect() and select() call must test each bit in each bit set before polling the socket for its status. The pselect() and select() call tests only sockets in the range 0 to *n*-1 (where *n* = the value of the second halfword of *nmsgsfds*).

Special behavior for C++: To use this function with C++, you must use the _XOPEN_SOURCE_EXTENDED 1 feature test macro.

Returned value

The value -1 indicates the error code should be checked for an error. The value zero indicates an expired time limit.

When the return value is greater than 0, then it is similar to *nmsgsfds* in that the high-order 16 bits give the number of message queues, and the low-order 16 bits give the number of descriptors. These values indicate the sum total that meet each of the read, write, and exception criteria. Note that a descriptor or a message queue may be counted multiple times if it meets more than one given criterion. Should the return value for message queues exceed the value 32767, only 32767 will be reported. This is to ensure that the return value does not appear to be negative. Should the return value for file/socket descriptors be greater than 65535, only 65535 will be reported.

If the return value is greater than 0, the files/sockets that are ready in each bit set are set to 1. Files/Sockets in each bit set that are not ready are set to zero. Use the macro FD_ISSET() with each file/socket to test its status. For those message queues that do not meet the conditions their identifiers in the msgsid arrays will be replaced with a value of -1.

Error Code
 Description

EBADF
 One of the bit sets specified an invalid socket or a message queue identifier is invalid. FD_ZERO() was probably not called to clear the bit set before the sockets were set.

EFAULT
 One of the parameters contained an invalid address.

EINTR
The pselect() or select() function was interrupted before any of the selected events occurred and before the timeout interval expired.

EINVAL
One of the fields in the **timeval** structure or **timespec** structure is invalid, or there was an invalid *nmsgsfds* value.

EIO One of the descriptors in the select mask has become inoperative and it is being repeatedly included in a select even though other operations against this descriptor have been failing with EIO. A socket descriptor, for example, can become inoperative if TCP/IP is shut down. When a descriptor fails a failure from select could not tell you which descriptor had failed so generally select will succeed and these descriptors will be reported to you as being ready for whatever events were specified on the select. Subsequently when the descriptor is used on a receive or other operation you will receive the EIO failure then and can react to the problem with the individual descriptor. In general you would close() the descriptor and remove it from the next select mask. If the individual descriptor's failing return code is ignored though and an inoperative descriptor is repeatedly selected on and used, even though each time it is used that call fails with EIO, eventually the select call itself will fail with EIO.

Note: The pselect() function can also return errno's set by the sigprocmask() function.

Example

In the following example, select() is used to poll sockets for reading (socket sr), writing (socket sw), and exception (socket se) conditions, and to check message queue ids mr, mw, and me.

```
#define _XOPEN_SOURCE_EXTENDED  1
#define _OPEN_MSGQ_EXT

#include <sys/types.h>
#include <sys/time.h>
#include <sys/msg.h>

struct sellist {
     fd_set fdset;
     int msgids[2];
};

/*
 * sock_msg_stats(sr, sw, se, mr, mw, me) - Print the status of
 *     sockets sr, sw, and se, and of message queue ids mr, mw,
 *     and me.
 */
 int sock_msg_stats(sr, sw, se, mr, mw, me)
 int sr, sw, se, mr, mw, me;
 {
    struct sellist  *reading, *writing, *excepting;
    struct sellist read, write, except;
    struct timeval timeout;
    int rc, max_sock, sock_size, nmsgsfds;
    int msgids[1];       /* we only check 1 message queue */

    /* What's the maximum socket number? */
    max_sock = MAX( sr, sw );
    max_sock = MAX( max_sock, se );
    /* initialize the static bit sets */
    FD_ZERO( &read.fdset );    reading = &read;
    FD_ZERO( &write.fdset );   writing = &write;
    FD_ZERO( &except.fdset );  excepting = &except;

    /* add sr, sw, and se to the appropriate bit set */
    FD_SET( sr, &reading->fdset );
    FD_SET( sw, &writing->fdset );
    FD_SET( se, &excepting->fdset );

    /* initialize the message id arrays */
    reading->msgids[0] = mr;
    writing->msgids[0] = mw;
    excepting->msgids[0] = me;

    /* set the nmsgsfds parameter */
    _SET_FDS_MSGS( nmsgsfds, 1, max_sock+1 );

    /* make select poll by sending a 0 timeval */
    memset( &timeout, 0, sizeof(timeout) );
    /* poll */
    rc = select( nmsgsfds, reading, writing, excepting, &timeout);

    if ( rc < 0 ) {
         /* an error occurred during the SELECT() */
         perror( "select" );
    }
    else if ( rc == 0 ) {
         /* no sockets or messages were ready in our little poll */
         printf( "nobody is home.\n" );
    } else
        if (_NFDS(rc) > 0)  {
         /* at least one of the sockets is ready */
         printf("sr is %s\n",
                FD_ISSET(sr,&reading->fdset) ? "READY" : "NOT READY");
         printf("sw is %s\n",
                FD_ISSET(sw,&writing->fdset) ? "READY" : "NOT READY");
         printf("se is %s\n",
                FD_ISSET(se,&excepting->fdset) ? "READY": "NOT READY");
        } else
            if (_NMSGS(rc) > 0)  {
              /* at least one message queue is ready */
              printf("mr is %s\n",
                reading->msgids[0] == -1 ? "NOT READY" : "READY");
              printf("mw is %s\n",
                writing->msgids[0] == -1 ? "NOT READY" : "READY");
              printf("me is %s\n",
                excepting->msgids[0] == -1 ? "NOT READY" : "READY");
            }
}
```

CELEBP72

```
/* CELEBP72

   This example demonstrates the use of pselect()

   Expected output:
   Parent: Issuing pselect

   This is the child
   Child: Sending signal to the parent at:


   This is the signal handler
   Signal received: 14 (14 is SIGALRM)
   The pselect call was made at:
```

```
   The SIGALRM was caught at:

   TEST PASSED!

*/
#define _POSIX_C_SOURCE 200112L
#include <sys/select.h>
#include <stdio.h>
#include <fcntl.h>
#include <signal.h>
#include <string.h>
#include <time.h>
#include <unistd.h>


time_t t1,t2;

void incatchr(int signum){
   double diff=0;

   time(&t2);
   printf("\n\nThis is the signal handler\n");
   printf("Signal received: %d (14 is SIGALRM) \n",signum);
   printf("The pselect call was made at: \t%s\n",ctime(&t1));
   printf("The SIGALRM was caught at:    \t%s\n",ctime(&t2));
   diff = difftime(t2,t1);
   if(diff <  10) {
      printf("TEST FAILED!\n\n");
   }
   else{
      printf("TEST PASSED!\n\n");
   }
}

int main(void){
   int fd[1], rc, nfds=3, fd1, fd2, fd3;
   pid_t cpid, ppid;
   fd_set fdsread;
   struct sigaction action, info;
   sigset_t pselect_set;
   struct timespec t;
   time_t t3;

   t.tv_sec=10;
   t.tv_nsec=0;

   FD_ZERO(&fdsread);

   action.sa_handler = incatchr;
   action.sa_flags = 0;
   sigaction(SIGALRM,&action,&info);

   sigemptyset(&pselect_set);
   sigaddset(&pselect_set, SIGALRM);

   fd1 = open("./testchd.txt",O_RDWR|O_CREAT);
   fd2 = open("./testchd2.txt",O_RDWR|O_CREAT);
   if((rc=pipe(fd)) != 0){
      printf("Error in pipe\n");
      return(-1);
   }

   FD_SET(fd[0],&fdsread);

   if ((cpid = fork()) < 0){
      printf("Fork error\n");
      return(-1);
   }
   else{
      if (cpid == 0){
         fd3 = open("./testchd.txt",O_RDWR|O_CREAT);
         printf("This is the child\n");
         sleep(2);
         ppid= getppid();
         time(&t3);
         printf("Child: Sending signal to the parent at: ");
         printf("%s",ctime(&t3));
         kill(ppid,SIGALRM);
         sleep(3);
         _exit(0);
      }
      else{
         printf("Parent: Issuing pselect\n\n");
         time(&t1);
         if (pselect(nfds,&fdsread,NULL,NULL,&t,&pselect_set) == -1)
            printf("Error in pselect\n");
      }
      close(fd[0]);
   }

   return 0;
}
```

Related information

- “sys/msg.h” on page 74
- “sys/times.h” on page 75
- “sys/types.h” on page 75
- “msgctl() — Message control operations” on page 1104
- “msgget() — Get message queue” on page 1105
- “msgrcv() — Message receive operation” on page 1108
- “msgsnd() — Message send operations” on page 1113
- “poll() — Monitor activity on file descriptors and message queues” on page 1185
- “selectex() — Monitor activity on files or sockets and message queues”

selectex() — Monitor activity on files or sockets and message queues

Standards

Standards / Extensions	C or C++	Dependencies
z/OS UNIX	both	

Format

X/Open:

```
#define _XOPEN_SOURCE_EXTENDED 1
#define _ALL_SOURCE
#define _OPEN_MSGQ_EXT
#include <sys/types.h>
#include <sys/time.h>
#include <sys/msg.h>

int selectex(int nmsgsfds, fd_set *readlist,
             fd_set *writelist,
             fd_set *exceptlist,
             struct timeval *timeout, int *ecbptr);
```

Berkeley sockets:

```
#define _OE_SOCKETS
#define _ALL_SOURCE
#define _OPEN_MSGQ_EXT
#include <sys/types.h>
#include <sys/time.h>
#include <sys/msg.h>

int selectex(int nmsgsfds, fd_set *readlist,
             fd_set *writelist,
             fd_set *exceptlist,
             struct timeval *timeout, int *ecbptr);
```

_OPEN_MSGQ_EXT must be defined if message queues are to be monitored (X/Open sockets only).

General description

The selectex() function provides an extension to the select() call by allowing you to use an ECB that defines an event not described by *readlist*, *writelist*, or *exceptlist*.

The selectex() call monitors activity on a set of files/sockets and message queues until a timeout occurs, or until the ECB is posted, to see if any of the files/sockets and message queues have read, write, or exception processing conditions pending.

When the storage key of the first (or only) ECB matches the caller's PSW key, the kernel performs the wait in the caller's PSW key; otherwise, the kernel performs the wait in the TCB key (TCBPFK). However, if the caller is running in key 0, then the kernel performs the wait in key 0, regardless of the storage key.

See select() for more information.

Parameter
  Description

nmsgsfds
  The number of message queues and the number of file or socket descriptors to check. (Refer to select() for a full description of this and other parameters below.)

  Note: This function is limited to descriptor numbers less than or equal to 65535.

readlist
  A pointer to an fd_set type, array of message queue identifiers, or *sellist* structure specifying descriptors and message queues to check for reading.

writelist
  A pointer to an fd_set type, array of message queue identifiers, or *sellist* structure specifying descriptors and message queues to check for writing.

exceptlist
  A pointer to an fd_set type, array of message queue identifiers, or *sellist* structure specifying descriptors and message queues to be checked for exceptional pending conditions.

timeout
  The pointer to the time to wait for the selectex() call to complete.

ecbptr
  This variable can contain one of the following values:

  1. A pointer to a user event control block. To specify this usage of *ecbptr*, the high-order bit must be set to '0'B.

2. A pointer to a list of ECBs. To specify this usage of *ecbptr*, the high-order bit must be set to `'1'B`.

   The list can contain the pointers for up to 1013 ECBs. The high-order bit of the last pointer in the list must be set to `'1'B`.
3. A NULL pointer. This indicates no ECBs are specified.

Special behavior for C++: To use this function with C++, you must use the _XOPEN_SOURCE_EXTENDED 1 feature test macro.

Returned value

The value -1 indicates the error code should be checked for an error. The value 0 indicates an expired time limit or that the ECB is posted.

When the return value is greater than 0, then it is similar to *nmsgsfds* in that the high-order 16 bits give the number of message queues, and the low-order 16 bits give the number of descriptors. These values indicate the sum total that meet each of the read, write, and exception criteria. Note that a descriptor or a message queue may be counted multiple times if it meets more than one requested criterion. Should the return value for message queues exceed the value 32767, only 32767 will be reported. This is to ensure that the return value does not appear to be negative. Should the return value for file/socket descriptors be greater than 65535, only 65535 will be reported.

If the return value is greater than 0, the files/sockets that are ready in each bit set are set to 1. Files/Sockets in each bit set that are not ready are set to zero. Use the macro FD_ISSET() with each socket to test its status. For those message queues that do not meet the conditions their identifiers in the msgsid array will be replaced with a value of -1.

Error Code
Description

EBADF
One of the descriptor sets specified an incorrect descriptor or a message queue identifier is invalid.

EFAULT
One of the parameters contained an invalid address.

EINTR
selectex() was interrupted before any of the selected events occurred and before the timeout interval expired.

EINVAL
One of the fields in the *timeval* structure is incorrect.

EIO
One of the descriptors in the select mask has become inoperative and it is being repeatedly included in a select even though other operations against this descriptor have been failing with EIO. A socket descriptor, for example, can become inoperative if TCP/IP is shut down. A failure from select can not tell you which descriptor has failed so generally select will succeed and these descriptors will be reported to you as being ready for whatever event they were being selected for. Subsequently when the descriptor is used on a receive or other operation you will receive the EIO failure and can react to the problem with the individual descriptor. In general you would close() the descriptor and remove it from the next select mask. If the individual descriptor's failing return code is ignored though

and an inoperative descriptor is repeatedly selected on and used, even though each time it is used that call fails with EIO, eventually the select call itself will fail with EIO.

Related information

- "sys/msg.h" on page 74
- "sys/times.h" on page 75
- "sys/types.h" on page 75
- "accept() — Accept a new connection on a socket" on page 105
- "connect() — Connect a socket" on page 308
- "msgctl() — Message control operations" on page 1104
- "msgget() — Get message queue" on page 1105
- "msgrcv() — Message receive operation" on page 1108
- "msgsnd() — Message send operations" on page 1113
- "poll() — Monitor activity on file descriptors and message queues" on page 1185
- "recv() — Receive data on a socket" on page 1398
- "send() — Send data on a socket" on page 1494

semctl() — Semaphore control operations

Standards

Standards / Extensions	C or C++	Dependencies
XPG4 XPG4.2 Single UNIX Specification, Version 3	both	

Format

```
#define _XOPEN_SOURCE
#include <sys/sem.h>

int semctl(int semid, int semnum, int cmd, ...);
```

General description

The semctl() function performs control operations in semaphore set *semid* as specified by the argument *cmd*.

Depending on the value of argument *cmd*, argument *semnum* may be ignored or identify one specific semaphore number.

The fourth argument is optional and depends upon the operation requested. If required, it is of type *union semun*, which the application program must explicitly declare:

```
union semun {
     int             val;
     struct semid_ds *buf;
     unsigned short  *array;
} arg;
```

Each semaphore in the semaphore set is represented by the following anonymous data structure:

unsigned short int	`semval`	Semaphore value
pid_t	`sempid`	Process ID of last operation

unsigned sort int	`semcnt`	Number of processes waiting for `semval` to become greater than current value
unsigned short int	`semzcnt`	Number of processes waiting for `semval` to become zero

When semctl() is used to identify one specific semaphore number for commands GETVAL, SETVAL, GETPID, GETNCNT, and GETZCNT, then references are made to this anonymous data structure for the semaphore *semnum*.

The following semaphore control operations as specified by argument *cmd* may be specified. The level of permission required for each operation is shown with each command. These symbolic constants are defined by the <sys/sem.h> header:

GETVAL
Returns the value of `semval`, if the current process has read permission.

SETVAL
Sets the value of `semval` to `arg.val`, where `arg` is the value of the fourth argument to semctl(). When this command is successfully executed, the `semadj` value corresponding to the specified semaphore in all processes is cleared. This command requires alter permission. For an __IPC_BINSEM semaphore set the only values that may be set are zero and one.

GETPID
Returns the most recent process to update the semaphore (sempid), if the current process has read permission.

GETNCNT
Returns the number of threads waiting on the semaphore to become greater than the current value, if the current process has read permission.

GETZCNT
Returns the number of threads waiting on the semaphore to become zero, if the current process has read permission. For an __IPC_BINSEM semaphore set this operation will always return a zero; threads are not allowed to wait for the semaphore to become zero in this type of semaphore set.

GETALL
Stores `semvals` for each semaphore in the semaphore set and place into the array pointed to by `arg.array`, where `arg.` is the fourth argument to semctl(). GETALL requires read permission. It is the caller's responsibility to ensure that the storage allocated is large enough to hold the number of semaphore elements. The number of semaphore values stored is `sem_nsems`, which may be obtained using the **IPC_STAT** command.

SETALL
Sets `semval` values for each semaphore in the semaphore set according to the array pointed to by arg.array, where `arg` is the fourth argument to semctl(). SETALL requires alter permission. Each `semval` value must be zero or positive. When this command is successfully executed, the `semadj` values corresponding to each specified semaphore in all processes are cleared. It is the caller's responsibility to ensure that the storage allocated is large enough to hold the number of semaphore elements. The number of semaphore values set is `sem_nsems`, which may be obtained using the **IPC_STAT** command. If __IPC_BINSEM was specified on the semget, this option should not be used while there is the possibility of other threads performing semaphore operations on this semaphore, as there may be no

serialization while updating the semaphore values; therefore a SETALL will not be allowed after a semop has been done to the __IPC_BINSEM semaphore set. Also, for the __IPC_BINSEM semaphore set, the only values that may be set are zero and one.

IPC_STAT
This command obtains status information for the semaphore identifier specified by *semid*. This requires read permission. This information is stored in the address specified by the fourth argument defined by data structure `semid_ds`.

IPC_SET
Set the value of the `sem_perm.uid`, `sem_perm.gid`, and `sem_perm.mode` in `semid_ds` data structure for the semaphore identifier specified by *semid*. These values are set to the values found in `semid_ds` structure pointed to by the fourth argument.

Any value for `sem_perm.uid` and `semperm.gid` may be set.

Only mode bits defined under semget() function argument *semflg* may be set in `sem_perm.mode`.

This command can only be executed by a process that has an effective user ID equal to either that of a process with appropriate privileges or to the value of `sem_perm.cuid` or `sem_perm.uid` in the `semid_ds` structure associated with *semid*.

IPC_RMID
Remove the semaphore identifier specified by argument *semid* from the system and free the storage for the set of semaphores in the `semid_ds` structure.

This command can only be executed by a process that has an effective user ID equal to either that of a process with appropriate privileges or to the value of `sem_perm.cuid` or `sem_perm.uid` in the `semid_ds` structure associated with *semid*. For an __IPC_BINSEM semaphore set, it is recommended that all use of semop should be completed before removing the semaphore ID.

Returned value

If successful, the value returned by semctl() depends on the value of the argument *cmd* as follows:

GETVAL
value of semval is returned

GETPID
value of sempid is returned

GETNCNT
value of semncnt is returned

GETZCNT
value of semzcnt is returned

All others
value of zero is returned

If unsuccessful, semctl() returns -1 and sets errno to one of the following values:

Error Code
Description

EACCES
Operation permission (read or write) is denied to the calling process.

EINVAL
The value of argument *semid* is not a valid semaphore identifier, or the value of *semnum* is less than zero or greater than or equal to the number of semaphores in the set, or the argument *cmd* is not a valid command, or the bits specified for `sem_perm.mode` are undefined. Note that the valid range of *semnum* is 0 to (number of semaphores in the set minus 1).

EPERM
The argument *cmd* has a value of **IPC_RMID** or **IPC_SET** and the effective user ID of the caller is not that of a process with appropriate privileges and is not the value of `sem_perm.cuid` or `sem_perm.uid` in the `semid_ds` data structure associated with *semid*.

ERANGE
The argument *cmd* has a value of **SETVAL** or **SETALL** and the `semval` value to be set exceeds the system limit as defined in <sys/sem.h>.

Related information

- “sys/ipc.h” on page 73
- “sys/sem.h” on page 74
- “semget() — Get a set of semaphores”
- “semop() — Semaphore operations” on page 1488

semget() — Get a set of semaphores

Standards

Standards / Extensions	C or C++	Dependencies
XPG4 XPG4.2 Single UNIX Specification, Version 3	both	

Format

```
#define _XOPEN_SOURCE
#include <sys/sem.h>

int semget(key_t key, int nsems, int semflg);
```

General description

The semget() function returns the semaphore identifier associated with *key*.

A semaphore identifier is created with a `semid_ds` data structure, see <sys/sem.h>, associated with *nsems* semaphores when any of the following is true:

- Argument *key* has a value of **IPC_PRIVATE**
- Argument *key* is not associated with a semaphore ID and (`semflg` & **IPC_CREAT**) is non zero.

Valid values for the field *semflg* include any combination of the following defined in <sys/ipc.h> and <sys/modes.h>:

IPC_CREAT
Creates a semaphore if the *key* specified does not already have an associated ID. **IPC_CREATE** is ignored when **IPC_PRIVATE** is specified.

IPC_EXCL
Causes the semget() function to fail if the *key* specified has an associated ID. **IPC_EXCL** is ignored when **IPC_CREAT** is not specified or **IPC_PRIVATE** is specified.

__IPC_BINSEM
Binary semaphore - semaphore must behave in a binary manner: number of semaphore operations must be 1 and the semop must be 1 with a semval of 0 or the semop must be -1 with a semval of 0 or 1. SEM_UNDO is now allowed on a semop() with this option. The use of this flag will cause improved performance if the PLO instruction is available on the hardware.

See *z/OS XL C/C++ Programming Guide* for further information on semaphore performance.

__IPC_SHORTHOLD
This flag states that it is known that the application will only hold the resource being serialized for extremely short time intervals. When this flag is combined with the __IPC_BINSEM flag, the default first-in-first-out (FIFO) ordering of semaphore obtain requesters will be bypassed, to allow short duration requesters to successfully obtain the semaphore (and hopefully release it) within the interval it normally takes to dispatch the next pending waiter for that semaphore.

S_IRUSR
Permits read access when the effective user ID of the caller matches either `sem_perm.cuid` or `sem_perm.uid`.

S_IWUSR
Permits write access when the effective user ID of the caller matches either `sem_perm.cuid` or `sem_perm.uid`.

S_IRGRP
Permits read access when the effective group ID of the caller matches either `sem_perm.cgid` or `sem_perm.gid`.

S_IWGRP
Permits write access when the effective group ID of the caller matches either `sem_perm.cgid` or `sem_perm.gid`.

S_IROTH
Permits others read access

S_IWOTH
Permits others write access

When a semaphore set associated with argument *key* already exists, setting **IPC_EXCL** and **IPC_CREAT** in argument *semflg* will force semget() to fail.

When a `semid_ds` data structure is created the following anonymous data structure is created for each semaphore in the set:

unsigned short int	`semval`	Semaphore value
pid_t	`sempid`	Process ID of last operation

unsigned sort int	`semcnt`	Number of processes waiting for `semval` to become greater than current value
unsigned short int	`semzcnt`	Number of processes waiting for `semval` to become zero

The following fields are initialized when a `semid_ds` data structure is created:

- The fields `sem_perm.cuid` and `sem_perm.uid` are set equal to the effective user ID of the calling process.
- The fields `sem_perm.cgid.` and `sem_perm.gid` are set equal to effective group ID of the calling process.
- The low-order 9 bits of `sem_perm.mode` are set to the value in the low-order 9 bits of *semflg*.
- The field `sem_nsems` is set to the value of *nsems*.
- The field `sem_otime` is set to 0.
- The field `sem_ctime` is set to the current time.
- The anonymous data structure containing `semval` for each semaphore is not initialized. semctl() commands **SETVAL** and **SETALL** should be used to initialize each semaphore's `semval` value.

Usage notes

1. Semaphores created with __IPC_BINSEM will show this bit and may show the IPC_PLOINUSE bit in the S_MODE byte returned with *w_getipc*.

Returned value

If successful, semget() returns a nonnegative semaphore identifier.

If unsuccessful, semget() returns -1 and sets errno to one of the following values:

Error Code
Description

EACCES
A semaphore identifier exists for the argument *key*, but access permission as specified by the low-order 9 bits of *semflg* could not be granted.

EEXIST
A semaphore identifier exists for the argument *key* and both **IPC_CREAT** and **IPC_EXCL** are specified in *semflg*.

EINVAL
The value of *nsems* is either less than zero or greater than the system limit. A semaphore identifier associated with *key* does not exist and the *nsems* is zero. A semaphore identifier associated with *key* already exists and the *nsems* value specified on semget() when the semaphore identifier was created is less than the *nsems* value on the current semget(). The *semflg* argument specified flags not currently supported.

ENOENT
A semaphore identifier does not exist for the argument *key* and **IPC_CREAT** is not specified.

ENOSPC
A system limit of number of semaphore identifiers has been reached.

When *semflg* equals 0, the following applies:

- If a semaphore identifier has already been created with *key* earlier, and the calling process of this semget() has read and/or write permissions to it, then semget() returns the associated semaphore identifier.
- If a semaphore identifier has already been created with *key* earlier, and the calling process of this semget() does not have read and/or write permissions to it, then semget() returns-1 and sets errno to EACCES.
- If a semaphore identifier has not been created with *key* earlier, then semget() returns -1 and sets errno to ENOENT.

Related information

- "sys/ipc.h" on page 73
- "sys/sem.h" on page 74
- "sys/stat.h" on page 75
- "sys/types.h" on page 75
- "ftok() — Generate an interprocess communication (IPC) key" on page 662
- "semctl() — Semaphore control operations" on page 1482
- "semop() — Semaphore operations"

semop() — Semaphore operations

Standards

Standards / Extensions	C or C++	Dependencies
XPG4 XPG4.2 Single UNIX Specification, Version 3	both	

Format

```
#define _XOPEN_SOURCE
#include <sys/sem.h>

int semop(int semid, struct sembuf *sops, size_t nsops);
```

General description

The semop() function performs semaphore operations atomically on a set of semaphores associated with argument *semid*. The argument *sops* is a pointer to an array of `sembuf` data structures. The argument *nsops* is the number of `sembuf` structures in the array.

The structure `sembuf` is defined as follows:

short	`sem_num`	Semaphore number in the range 0 to (nsems - 1)
short	`sem_op`	Semaphore operation
short	`sem_flg`	Operation flags

Each semaphore in the semaphore set, identified by `sem_num`, is represented by the following anonymous data structure. This data structure for all semaphores is updated atomically when semop() returns successfully:

unsigned short int	`semval`	Semaphore value
pid_t	`sempid`	Process ID of last operation
unsigned sort int	`semcnt`	Number of processes waiting for `semval` to become greater than current value

unsigned short int	`semzcnt`	Number of processes waiting for `semval` to become zero

Each semaphore operation specified by `sem_op` is performed on the corresponding semaphore specified by *semid* and `sem_num`.

The variable `sem_op` specifies one of three semaphore operations:

1. If `sem_op` is a negative integer and the calling process has alter permission, one of the following will occur:
   - If semval, see <sys/sem.h>, is greater than or equal to the absolute value of `sem_op`, the absolute value of `sem_op` is subtracted from `semval`.
   - If `semval` is less than the absolute value of `sem_op` and (`sem_flg` & **IPC_NOWAIT**) is nonzero, semop() will return immediately.
   - If *semval* is less than the absolute value of `sem_op` and (`sem_flg` & **IPC_NOWAIT**) is zero, semop() will increment the `semncnt` associated with the specified semaphore and suspend execution of the calling process until one of the following conditions occurs:
     - The value of `semval` becomes greater than or equal to the absolute value of `sem_op`. When this occurs, the value of semncnt associated with the specified semaphore is decremented, the absolute value of `sem_op` is subtracted from `semval`.
     - The *semid* for which the calling process is awaiting action is removed from the system. When this occurs, errno is set equal to EIDRM and -1 is returned.
     - The calling process receives a signal that is to be caught. When this occurs, the value of `semncnt` associated with the specified semaphore is decremented, and the calling process resumes execution in the manner prescribed in sigaction().
2. If `sem_op` is a positive integer and the calling process has alter permission, the value of `sem_op` is added to `semval`.
3. If `sem_op` is zero and the calling process has read permission, one of the following will occur:
   - If `semval` is zero, semop() will return immediately.
   - If `semval` is nonzero and (`sem_flg`&**IPC_NOWAIT**) is nonzero, semop() will return immediately.
   - If `semval` is nonzero and (`sem_flg`&**IPC_NOWAIT**) is 0, semop() will increment the semzcnt associated with the specified semaphore and suspend execution of the calling thread until one of the following occurs:
     - The value of `semval` becomes 0, at which time the value of `semzcnt` associated with the specified semaphore is decremented.
     - The *semid* for which the calling process is awaiting action is removed from the system. When this occurs, errno is set equal to EIDRM and -1 is returned.
     - The calling process receives a signal that is to be caught. When this occurs, the value of `semzcnt` associated with the specified semaphore is decremented, and the calling process resumes execution in the manner prescribed in sigaction().
     - Upon successful completion, the value of `sempid` for each semaphore specified in the array pointed to by *sops* is set equal to the process ID of the calling process.

sem_flg contains the **IPC_NOWAIT** and **SEM_UNDO** flags described as follows:

IPC_NOWAIT
Will cause semop() to return EAGAIN rather than place the thread into wait state.

SEM_UNDO
Will result in semadj adjustment values being maintained for each semaphore on a per process basis. If sem_op value is not equal to zero and **SEM_UNDO** is specified, then sem_op value is subtracted from the current process's semadj value for that semaphore. When the current process is terminated, see exit(), the semadj value(s) will be added to the semval for each semaphore. The semctl() command **SETALL** may be used to clear all semadj values in all processes. If __IPC_BINSEM was specified on semget for this semaphore, the Sem_UNDO flag will cause an error to be returned.

A semaphore set created with the __IPC_BINSEM flag must behave in the following manner: number of semaphore operations must be 1 and the semop must be +1 with a semval of 0 or the semop must be -1 with a semval of 0 or 1. SEM_UNDO is not allowed on a semop() with this option. The use of this flag will cause improved performance if the PLO instruction is available on the hardware.

Returned value

If successful, semop() returns 0. Also the *semid* parameter value for each semaphore that is operated upon is set to the process ID of the calling process.

If unsuccessful, semop() returns -1 and sets errno to one of the following values:

Error Code
Description

E2BIG The value *nsops* is greater than the system limit.

EACCES
Operation permission is denied to the calling process. Read access is required when sem_op is zero. Write access is required when sem_op is not zero.

EAGAIN
The operation would result in suspension of the calling process but **IPC_NOWAIT** in sem_flg was specified.

EFBIG
sem_num is less than zero or greater or equal to the number of semaphores in the set specified on in semget() argument *nsems*.

EIDRM
semid was removed from the system while the invoker was waiting.

EINTR
semop() was interrupted by a signal.

EINVAL
The value of argument *semid* is not a valid semaphore identifier. For an __IPC_BINSEM semaphore set, the sem_op is other than +1 for a sem_val of 0 or -1 for a sem_val of 0 or 1. Also, for an __IPC_BINSEM semaphore set, the number of semaphore operations is greater than one.

ENOSPC
The limit on the number of individual processes requesting a **SEM_UNDO** would be exceeded.

ERANGE
An operation would cause `semval` or `semadj` to overflow the system limit as defined in <sys/sem.h>.

Related information

- “sys/ipc.h” on page 73
- “sys/sem.h” on page 74
- “sys/types.h” on page 75
- “exec functions” on page 436
- “exit() — End program” on page 443
- “_exit() — End a process and bypass the cleanup” on page 445
- “fork() — Create a new process” on page 571
- “rexec() — Execute commands one at a time on a remote host” on page 1447
- “semctl() — Semaphore control operations” on page 1482
- “semget() — Get a set of semaphores” on page 1485
- “__semop_timed() — Semaphore operations with timeout”

__semop_timed() — Semaphore operations with timeout

Standards

Standards / Extensions	C or C++	Dependencies
z/OS UNIX	both	OS/390 V2R10

Format

```
#define _OPEN_SYS_TIMED_EXT 1
#include <time.h>
#include <sys/sem.h>

int __semop_timed(int semid, struct sembuf *sops, size_t nsops,
                  struct timespec *set)
```

General description

Performs semaphore operations atomically on a set of semaphores associated with argument *semid*. The argument *sops* is a pointer to an array of `sembuf` data structures. The argument *nsops* is the number of `sembuf` structures in the array. The argument *set* the structure timespec with the timeout values.

The structure `sembuf` is defined as follows:

short	`sem_num`	Semaphore number in the range 0 to (nsems - 1)
short	`sem_op`	Semaphore operation
short	`sem_flg`	Operation flags

Each semaphore in the semaphore set, identified by `sem_num`, is represented by the following anonymous data structure. This data structure for all semaphores is updated automatically when semop() returns successfully:

unsigned short int	`semval`	Semaphore value
pid_t	`sempid`	Process ID of last operation

unsigned sort int	semcnt	Number of processes waiting for semval to become greater than current value
unsigned short int	semzcnt	Number of processes waiting for semval to become zero

Each semaphore operation specified by sem_op is performed on the corresponding semaphore specified by *semid* and sem_num.

The variable sem_op specifies one of three semaphore operations:

1. If sem_op is a negative integer and the calling process has alter permission, one of the following will occur:
   - If semval, see <sys/sem.h>, is greater than or equal to the absolute value of sem_op, the absolute value of sem_op is subtracted from semval.
   - If semval is less than the absolute value of sem_op and (sem_flg & **IPC_NOWAIT**) is nonzero, semop() will return immediately.
   - If *semval* is less than the absolute value of sem_op and (sem_flg & **IPC_NOWAIT**) is zero, semop() will increment the semncnt associated with the specified semaphore and suspend execution of the calling process until one of the following conditions occurs:
     - The value of semval becomes greater than or equal to the absolute value of sem_op. When this occurs, the value of semncnt associated with the specified semaphore is decremented, the absolute value of sem_op is subtracted from semval.
     - The *semid* for which the calling process is awaiting action is removed from the system. When this occurs, errno is set equal to EIDRM and -1 is returned.
     - The calling process receives a signal that is to be caught. When this occurs, the value of semncnt associated with the specified semaphore is decremented, and the calling process resumes execution in the manner prescribed in sigaction().
2. If sem_op is a positive integer and the calling process has alter permission, the value of sem_op is added to semval.
3. If sem_op is zero and the calling process has read permission, one of the following will occur:
   - If semval is zero, semop() will return immediately.
   - If semval is nonzero and (sem_flg&**IPC_NOWAIT**) is nonzero, semop() will return immediately.
   - If semval is nonzero and (sem_flg&**IPC_NOWAIT**) is 0, semop() will increment the semzcnt associated with the specified semaphore and suspend execution of the calling thread until one of the following occurs:
     - The value of semval becomes 0, at which time the value of semzcnt associated with the specified semaphore is decremented.
     - The *semid* for which the calling process is awaiting action is removed from the system. When this occurs, errno is set equal to EIDRM and -1 is returned.
     - The calling process receives a signal that is to be caught. When this occurs, the value of semzcnt associated with the specified semaphore is decremented, and the calling process resumes execution in the manner prescribed in sigaction().

  - Upon successful completion, the value of sempid for each semaphore specified in the array pointed to by *sops* is set equal to the process ID of the calling process.

The variable, *set*, gives the timeout specification.

- If the __semop_timed() function finds that none of the semaphores specified by *semid* are received, it waits for the time interval specified in the **timespec** structure referenced by *set*. If the **timespec** structure pointed to by *set* is zero-valued and if none of the semaphores specified by *semid* are received, then __semop_timed() returns immediately with EAGAIN. A **timespec** with the tv_sec field set with *INT_MAX*, as defined in <limits.h>, will cause the __semop_timed() service to wait until a semaphore is received. If *set* is the NULL pointer, it will be treated the same as when **timespec** structure was supplied with with the tv_sec field set with *INT_MAX*.

Returned value

If successful, __semop_timed() returns 0. Also the *semid* parameter value for each semaphore that is operated upon is set to the process ID of the calling process.

If unsuccessful, __semop_timed() returns -1 and sets errno to one of the following values:

Error Code
Description

E2BIG The value *nsops* is greater than the system limit.

EACCES
Operation permission is denied to the calling process. Read access is required when sem_op is zero. Write access is required when sem_op is not zero.

EAGAIN
The operation would result in suspension of the calling process but **IPC_NOWAIT** in sem_flg was specified. This would result if the timeout specified expires before a semop is posted.

EFBIG
sem_num is less than zero or greater or equal to the number of semaphores in the set specified on in semget() argument *nsems*.

EIDRM
semid was removed from the system while the invoker was waiting.

EINTR
__semop_timed() was interrupted by a signal.

EINVAL
The value of argument *semid* is not a valid semaphore identifier.

ENOSPC
The limit on the number of individual processes requesting a **SEM_UNDO** would be exceeded.

ERANGE
An operation would cause semval or semadj to overflow the system limit as defined in <sys/sem.h>.

Related information

- "sys/sem.h" on page 74
- "time.h" on page 79
- "semop() — Semaphore operations" on page 1488

send() — Send data on a socket

Standards

Standards / Extensions	C or C++	Dependencies
XPG4.2 Single UNIX Specification, Version 3	both	

Format

X/Open:

```
#define _XOPEN_SOURCE_EXTENDED 1
#include <sys/socket.h>

ssize_t send(int socket, const void *buffer, size_t length, int flags);
```

Berkeley sockets:

```
#define _OE_SOCKETS
#include <sys/socket.h>

int send(int socket, char *buffer, int length, int flags);
```

General description

The send() function sends data on the socket with descriptor *socket*. The send() call applies to all connected sockets.

Parameter
Description

socket The socket descriptor.

msg The pointer to the buffer containing the message to transmit.

length The length of the message pointed to by the *msg* parameter.

flags The *flags* parameter is set by If more than one flag is specified, the logical OR operator (|) must be used to separate them.

MSG_OOB

Sends out-of-band data on sockets that support this notion. Only SOCK_STREAM sockets support out-of-band data. The out-of-band data is a single byte.

Before out-of-band data can be sent between two programs, there must be some coordination of effort. If the data is intended to not be read inline, the recipient of the out-of-band data must specify the recipient of the SIGURG signal that is generated when the out-of-band data is sent. If no recipient is set, no signal is sent. The recipient is set up by using F_SETOWN operand of the fcntl command, specifying either a pid or gid. For more information on this operand, refer to the fcntl command.

The recipient of the data determines whether to receive out-of-band data inline or not inline by the setting of the SO_OOBINLINE option of setsockopt(). For more information on receiving out-of-band data, refer to the setsockopt(), recv(), recvfrom() and recvmsg() commands.

MSG_DONTROUTE
The SO_DONTROUTE option is turned on for the duration of the operation. This is usually used only by diagnostic or routing programs.

If there is not enough available buffer space to hold the socket data to be transmitted, and the socket is in blocking mode, send() blocks the caller until additional buffer space becomes available. If the socket is in nonblocking mode, send() returns -1 and sets the error code to EWOULDBLOCK. See "fcntl() — Control open file descriptors" on page 474 or "ioctl() — Control device" on page 869 for a description of how to set nonblocking mode.

For datagram sockets, this call sends the entire datagram, provided that the datagram fits into the TCP/IP buffers. Stream sockets act like streams of information with no boundaries separating data. For example, if an application wishes to send 1000 bytes, each call to this function can send 1 byte, or 10 bytes, or the entire 1000 bytes. Therefore, applications using stream sockets should place this call in a loop, calling this function until all data has been sent.

Special behavior for C++: To use this function with C++, you must use the _XOPEN_SOURCE_EXTENDED 1 feature test macro.

Returned value

If successful, send() returns 0 or greater indicating the number of bytes sent. However, this does not assure that data delivery was complete. A connection can be dropped by a peer socket and a SIGPIPE signal generated at a later time if data delivery is not complete.

If unsuccessful, send() returns -1 indicating locally detected errors and sets errno to one of the following values. No indication of failure to deliver is implicit in a send() routine.

Error Code
Description

EBADF
socket is not a valid socket descriptor.

ECONNRESET
A connection was forcibly closed by a peer.

EDESTADDRREQ
The socket is not connection-oriented and no peer address is set.

EFAULT
Using the *msg* and *length* parameters would result in an attempt to access storage outside the caller's address space.

EINTR
A signal interrupted send() before any data was transmitted.

EIO There has been a network or transport failure.

EMSGSIZE
The message was too big to be sent as a single datagram.

ENOBUFS
Buffer space is not available to send the message.

ENOTCONN
The socket is not connected.

ENOTSOCK
The descriptor is for a file, not for a socket.

EOPNOTSUPP
The *socket* argument is associated with a socket that does not support one or more of the values set in *flags*.

EPIPE For a connected stream socket the connection to the peer socket has been lost. A SIGPIPE signal is sent to the calling process.

EWOULDBLOCK
socket is in nonblocking mode and no data buffers are available or the SO_SNDTIMEO timeout value was reached before buffers became available.

Related information

- "sys/socket.h" on page 74
- "connect() — Connect a socket" on page 308
- "fcntl() — Control open file descriptors" on page 474
- "getsockopt() — Get the options associated with a socket" on page 777
- "ioctl() — Control device" on page 869
- "read() — Read from a file or socket" on page 1376
- "readv() — Read data on a file or socket and store in a set of buffers" on page 1389
- "recv() — Receive data on a socket" on page 1398
- "recvfrom() — Receive messages on a socket" on page 1401
- "recvmsg() — Receive messages on a socket and store in an array of message headers" on page 1405
- "select(), pselect() — Monitor activity on files or sockets and message queues" on page 1470
- "selectex() — Monitor activity on files or sockets and message queues" on page 1479
- "sendmsg() — Send messages on a socket" on page 1500
- "sendto() — Send data on a socket" on page 1506
- "socket() — Create a socket" on page 1688
- "write() — Write data on a file or socket" on page 2085
- "writev() — Write data on a file or socket from an array" on page 2092

send_file() — Send file data over a socket

Standards

Standards / Extensions	C or C++	Dependencies
z/OS UNIX	both	

Format

```
#define _OPEN_SYS_SOCK_EXT2
#include <sys/socket.h>

int send_file(int *socket_ptr, struct sf_parms *sf_struct, int options);
```

General description

The send_file() function sends data from the file associated with the open file handle over the connection associated with the socket.

The function takes the following arguments:

socket_ptr
A pointer to a socket file descriptor.

sf_struct
A pointer to a structure that contains variables needed by sendfile - header information, file information, trailer information and results of operation. See below for details.

options Specifies one of the following options:

SF_CLOSE
Close the connection after the data has been successfully sent or queued for transmission.

SF_REUSE
Prepare the socket for reuse after the data has been successfully sent or queued for transmission and the existing connection closed.

0 Do not close the socket (use zero for no options).

Send_File Structure - sf_parms

Argument *sf_struct* points to a struct *sf_parms* that contains the file descriptor, a header data buffer, and a trailer data buffer.

The struct *sf_parms* is defined in <*sys/sockets.h*> and contains the following elements:

header_data
Pointer to a buffer that contains header data which is to be sent before the file data. It may be a NULL pointer if *header_length* is zero.

header_length
Specifies the number of bytes in the *header_data*. It must be set to zero to indicate that header data is not to be sent.

file_descriptor
File descriptor for a file that has been opened for read. This is the descriptor for the file that contains the data to be transmitted.

file_size
The size, in bytes, of the file associated with *file_descriptor*. This field is filled in by the system.

file_offset
Specifies the byte offset into the file from which to start sending data.

file_bytes
Specifies the number of bytes from the file to be transmitted. Setting

file_bytes to -1 will transmit the entire file from the *offset*. In this case the system will replace the -1 with (actual file size - *file_offset*). Setting *file_bytes* to 0 will result in no file data being transmitted and *file_descriptor* is ignored. If *file_descriptor* is not a regular file it may be necessary to supply a specific value for *file_bytes* unless a normal End Of File (EOF) indication is expected from *file_descriptor* during this operation or you simply want the operation to run forever transferring bytes as they arrive.

trailer_data
: Pointer to a buffer that contains trailer data which is to be sent after the file data.

trailer_length
: Specifies the number of bytes in the *trailer_data*.

bytes_sent
: Number of bytes that were sent in this call to send_file(). If it takes multiple calls to send_file() to send all the data (due to signal-handling) then this field contains the value for the last call to send_file(), it is not a running total. This field is set by the system.

The send_file() function attempts to write *header_length* bytes from the buffer pointed to by *header_data*, followed by *file_bytes* from the file associated with *file_descriptor*, followed by *trailer_length* bytes from the buffer pointed to by *trailer_data*, over the connection associated with the socket pointed to by *socket_ptr*.

As data is sent, the system will update elements in *sf_struct* so that if the send_file() function is interrupted by a signal, the application simply needs to reissue send_file()

If the application sets *file_offset* > the actual file size, or *file_bytes* > (the actual file size - *file_offset*), the return value will be -1 and errno set to *EINVAL*.

If O_NONBLOCK is set on the socket file descriptor, the function may return -1 with errno set to EWOULDBLOCK or EAGAIN, or it may complete before all the data is sent. If O_NONBLOCK is not set, send_file() blocks until the requested data can be sent.

`SF_CLOSE` and `SF_REUSE` will only be effective after all the data has been sent successfully.

If *options* = `SF_REUSE`, and socket reuse is not supported, the system will close the socket and set the socket pointed to by *socket_ptr* to -1. See “Application usage” for details.

Application usage

The function send_file() is designed to work with accept_and_recv() to provide an efficient file transfer capability for a connection oriented server with short connection times and high connection rates.

On the first call to accept_and_recv(), it is recommended that the application set the socket pointed to by *accept_socket* to -1. This will cause the system to assign the accepting socket. On the call to send_file(), if the application requests socket reuse (*options* = `SF_REUSE`) and the system does not support it, the system will close the socket pointed to by *socket_ptr* and will set the socket pointed to by *socket_ptr* to -1. The application then passes this value onto the next call to accept_and_recv() (by setting *accept_socket = *socket_ptr*).

To take full advantage of the performance improvements offered by the accept_and_recv() and send_file() functions, a process/thread model different from the one where a parent accepts in a loop and spins off child process threads is needed. The parent/process thread is eliminated. Multiple worker processes/threads are created, and each worker process/thread then executes the accept_and_recv() and send_file() functions in a loop. The performance benefits of accept_and_recv() and send_file() include fewer buffer copies, recycled sockets, and optimal scheduling.

Note: The function send_file() can be useful by itself for long living socket connections. It does not have to be paired with accept_and_recv(). To leave the socket open after send_file(), specify a 0 for the *options* parameter.

Returned value

If successful, send_file() returns 0.

send_file() returns 1 if the request was interrupted by a signal, or because a nonblocking descriptor would have blocked, while sending data. Since the *sf_parms* structure is updated by the system to account for the data that has been sent you can continue the operation from where is was interrupted by recalling send_file() without changing the *sf_parms* structure.

If unsuccessful, send_file() returns -1 and sets errno to one of the following values:

Error Code
    Description

EACCES
    The calling process does not have the appropriate privileges.

EAGAIN
    socket_ptr is in nonblocking mode and no data buffers are available or the SO_SNDTIMEO timeout value was reached before buffers became available.

EBADF
    One of the following occurred:

1. *socket_ptr* is not a valid descriptor or was not open for writing.
2. *file_descriptor* is not a valid descriptor or was not open for reading.

ECONNABORTED
    A connection has been aborted.

ECONNRESET
    A connection has been forcibly closed by a peer.

EFAULT
    The data buffer pointed to by *socket_ptr*, *file_size*, *header_data*, or *trailer_data* was not valid.

EINTR
    send_file() was interrupted by a signal that was caught before any data was sent.

EINVAL
The value specified by *options* is not valid.

EIO An I/O error occurred.

EMSGSIZE
The message is too large to be sent all at once, as the socket requires.

ENETDOWN
The local interface to reach the destination is unknown.

ENETUNREACH
No route to the destination is present.

ENOBUFS
No buffer space is available.

ENOMEM
There was insufficient memory available to complete the operation.

ENOSR
There were insufficient STREAMS resources available for the operation to complete.

ENOSYS
This function is not supported in the current environment.

ENOTCONN
The socket is not connected.

ENOTSOCK
The file descriptor pointed to by the *socket_ptr* argument does not refer to a socket.

EPIPE The socket is shutdown for writing, or the socket is in connection mode and is no longer connected.

EWOULDBLOCK
socket_ptr is in nonblocking mode and no data buffers are available or the SO_SNDTIMEO timeout value was reached before buffers became available.

Related information

- “sys/socket.h” on page 74
- “accept_and_recv() — Accept connection and receive first message” on page 108
- “read() — Read from a file or socket” on page 1376
- “send() — Send data on a socket” on page 1494
- “socket() — Create a socket” on page 1688

sendmsg() — Send messages on a socket

Standards

Standards / Extensions	C or C++	Dependencies
XPG4.2 Single UNIX Specification, Version 3	both	

Format

X/Open:

```
#define _XOPEN_SOURCE_EXTENDED 1
#include <sys/socket.h>

ssize_t sendmsg(int socket, struct msghdr *msg, int flags);
```

Berkeley sockets:

```
#define _OE_SOCKETS
#include <sys/socket.h>

int sendmsg(int socket, struct msghdr *msg, int flags);
```

General description

The sendmsg() function sends messages on a socket with a socket descriptor passed in an array of message headers.

Parameter
Description

socket The socket descriptor.

msg An array of message headers from which messages are sent.

flags Specifying one or more of the following flags. If more than one flag is specified, the logical OR operator (|) must be used to separate them.

MSG_OOB
Sends out-of-band data on the socket. Only SOCK_STREAM sockets support out-of-band data. The out-of-band data is a single byte.

Before out-of-band data can be sent between two programs, there must be some coordination of effort. If the data is intended to not be read inline, the recipient of the out-of-band data must specify the recipient of the SIGURG signal that is generated when the out-of-band data is sent. If no recipient is set, no signal is sent. The recipient is set by setting the *action* parameter of the fcntl() function to F_SETOWN and specifying either a PID or GID. For more information on setting a recipient for out-of-band data, see "fcntl() — Control open file descriptors" on page 474.

The recipient of the data determines whether to receive out-of-band data inline or not inline by the setting of the SO_OOBINLINE socket option using the setsockopt() function. For more information on receiving out-of-band data, see "setsockopt() — Set options associated with a socket" on page 1578, "recv() — Receive data on a socket" on page 1398, "recvfrom() — Receive messages on a socket" on page 1401, and "recvmsg() — Receive messages on a socket and store in an array of message headers" on page 1405.

MSG_DONTROUTE
The SO_DONTROUTE socket option is turned on for the duration of the operation. This flag is typically used by diagnostic or routing programs.

A message header is defined by the msghdr structure, which can be found in the sys/socket.h header file and contains the following elements:

Element
Description

msg_iov
An array of *iovec* buffers containing the message.

msg_iovlen
The number of elements in the *msg_iov* array.

msg_name
An optional pointer to the buffer containing the recipient's address.

msg_namelen
The size of the address buffer.

caddr_t msg_accrights
Access rights sent or received (ignored if specified by the user). This field is ignored by z/OS UNIX services.

int msg_accrightslen
Length of access rights data (ignored if specified by the user). This field is ignored by z/OS UNIX services.

msg_control
Ancillary data.

msg_controllen
Ancillary data buffer length.

msg_flags
Flags on received message.

Ancillary data consists of a sequence of pairs, each consisting of a cmsghdr structure followed by a data array. The data array contains the ancillary data message and the cmsghdr structure contains descriptive information that allows an application to correctly parse the data.

The sys/socket.h header file defines the cmsghdr structure that includes at least the following elements:

Element
Description

cmsg_len
Data byte count, including header.

cmsg_level
Originating protocol.

cmsg_type
Protocol-specific type.

The following ancillary data are available at the IPv4 level:

Ancillary data
Description

IP_PKTINFO
(RAW and UDP) Specifies the interface packets are sent over and the IP address used as the packet source IP. The data is passed in an in_pktinfo structure as defined in netinet/in.h.

The following ancillary data are available at the IPv6 level:

Ancillary data
Description

IPV6_HOPLIMIT
(RAW, TCP, and UDP) Specifies the maximum hop limit for an outgoing packet. The data is passed in a structure as defined in netinet/in.h.

IPV6_PATHMTU
(RAW and UDP) Specifies the path MTU value for the destination of a connected socket. The data is passed in a structure as defined in netinet/in.h.

IPV6_PKTINFO
(RAW and UDP) Specifies the interface packets are sent over and the IP address used as the packet source IP. The data is passed in an in6_pktinfo structure as defined in netinet/in.h.

The following ancillary data are available at the socket level:

Ancillary data
Description

SCM_RIGHTS
Specifies the data array that contains the access rights to be sent or received. This ancillary data is valid only for the AF_UNIX domain. The data is passed in a structure as defined in sys/socket.h.

The sys/socket.h header file defines the following macros to gain access to the data arrays in the ancillary data associated with a message header:

CMSG_DATA*(cmsg)*
If the argument is a pointer to a cmsghdr structure, this macro returns an unsigned character pointer to the data array associated with the cmsghdr structure.

CMSG_NXTHDR*(mhdr,cmsg)*
If the first argument is a pointer to a msghdr structure and the second argument is a pointer to a cmsghdr structure in the ancillary data (pointed to by the *msg_control* field of that msghdr structure), this macro returns a pointer to the next cmsghdr structure or a NULL pointer if this structure is the last cmsghdr structure in the ancillary data.

CMSG_FIRSTHDR*(mhdr)*
If the argument is a pointer to a msghdr structure, this macro returns a pointer to the first cmsghdr structure in the ancillary data associated with this msghdr structure, or a NULL pointer if there is no ancillary data associated with the msghdr structure.

The sendmsg() call applies to sockets regardless of whether they are in the connected state.

This call returns the length of the data sent. If there is not enough available buffer space to hold the socket data to be transmitted, and the socket is in blocking mode, sendmsg() blocks the caller until additional buffer space becomes available. If the socket is in nonblocking mode, sendmsg() returns -1 and sets the error code to EWOULDBLOCK. See “fcntl() — Control open file descriptors” on page 474 or “ioctl() — Control device” on page 869 for a description of how to set nonblocking mode.

For datagram sockets, this call sends the entire datagram, provided that the datagram fits into the TCP/IP buffers. Stream sockets act like streams of information with no boundaries separating data. For example, if an application wishes to send 1000 bytes, each call to this function can send 1 byte, or 10 bytes, or the entire 1000 bytes. Therefore, applications using stream sockets should place this call in a loop, calling this function until all data has been sent.

Socket address structure for IPv6: For an AF_INET6 socket, if msg_name is specified then the address should be in a sockaddr_in6 address structure. The sockaddr_in6 structure is defined in the header file netinet/in.h.

Special behavior for C++: To use this function with C++, you must use the _XOPEN_SOURCE_EXTENDED 1 feature test macro.

Note: The sendmsg() function has a dependency on the level of the Enhanced ASCII Extensions. See "Enhanced ASCII support" on page 2113 for details.

Returned value

If successful, sendmsg() returns the length of the message in bytes.

A value of 0 or greater indicates the number of bytes sent, however, this does not assure that data delivery was complete. A connection can be dropped by a peer socket and a SIGPIPE signal generated at a later time if data delivery is not complete.

If unsuccessful, sendmsg() returns -1 and sets errno to one of the following values:

Error Code
  Description

EADDRNOTAVAIL
  The *ipi6_addr* is not available for use on the *ipi6_ifindex* interface.

EAFNOSUPPORT
  The address family is not supported (it is not AF_UNIX, AF_INET, or AF_INET6).

EBADF
  socket is not a valid socket descriptor.

ECONNREFUSED
  The attempt to connect was rejected.

ECONNRESET
  A connection was forcibly closed by a peer.

EFAULT
  Using *msg* would result in an attempt to access storage outside the caller's address space.

EHOSTUNREACH
  No route to the destination exists over the interface specified by *ifi6_index*.

EINTR
  A signal interrupted sendmsg() before any data was transmitted.

EINVAL
  msg_namelength is not the size of a valid address for the specified address family.

EIO There has been a network or transport failure.

EMSGSIZE
  The message was too big to be sent as a single datagram. The default is

large-envelope-size. (Envelopes are used to hold datagrams and fragments during TCP/IP processing. Large envelopes hold UDP datagrams greater than 2KB while they are processed for output, and when they are waiting for an application program to receive them on input.)

ENETDOWN
The interface specified by *ipi6_ifindex* is not enabled for IPv6 use.

ENOBUFS
Buffer space is not available to send the message.

ENOTCONN
The socket is not connected.

ENOTSOCK
The descriptor is for a file, not for a socket.

ENXIO
The interface specified by *ipi6_ifindex* does not exist.

EOPNOTSUPP
The *socket* argument is associated with a socket that does not support one or more of the values set in *flags*.

EPIPE For a connected stream socket the connection to the peer socket has been lost. A SIGPIPE signal is sent to the calling process.

EWOULDBLOCK
socket is in nonblocking mode and no data buffers are available or the SO_SNDTIMEO timeout value was reached before buffers became available.

The following are for AF_UNIX only:

Error Code
Description

EACCES
Search permission is denied for a component of the path prefix, or write access to the named socket is denied.

EIO An I/O error occurred while reading from or writing to the file system.

ELOOP
Too many symbolic links were encountered in translating the pathname in the socket address.

ENAMETOOLONG
A component of a pathname exceeded **NAME_MAX** characters, or an entire pathname exceeded **PATH_MAX** characters.

ENOENT
A component of the pathname does not name an existing file or the pathname is an empty string.

ENOTDIR
A component of the path prefix of the pathname in the socket address is not a directory.

Related information

- “sys/socket.h” on page 74
- “connect() — Connect a socket” on page 308
- “fcntl() — Control open file descriptors” on page 474

- “getsockopt() — Get the options associated with a socket” on page 777
- “ioctl() — Control device” on page 869
- “read() — Read from a file or socket” on page 1376
- “readv() — Read data on a file or socket and store in a set of buffers” on page 1389
- “recv() — Receive data on a socket” on page 1398
- “recvfrom() — Receive messages on a socket” on page 1401
- “recvmsg() — Receive messages on a socket and store in an array of message headers” on page 1405
- “select(), pselect() — Monitor activity on files or sockets and message queues” on page 1470
- “selectex() — Monitor activity on files or sockets and message queues” on page 1479
- “sendto() — Send data on a socket”
- “setsockopt() — Set options associated with a socket” on page 1578
- “socket() — Create a socket” on page 1688
- “write() — Write data on a file or socket” on page 2085
- “writev() — Write data on a file or socket from an array” on page 2092

sendto() — Send data on a socket

Standards

Standards / Extensions	C or C++	Dependencies
XPG4.2 Single UNIX Specification, Version 3	both	

Format

X/Open:

```
#define _XOPEN_SOURCE_EXTENDED 1
#include <sys/socket.h>

ssize_t sendto(int socket, const void *buffer, size_t length, int flags,
               const struct sockaddr *address, size_t address_len);
```

Berkeley sockets:

```
#define _OE_SOCKETS
#include <sys/socket.h>

int sendto(int socket, char *buffer, int length, int flags,
           struct sockaddr *address, int address_len);
```

General description

The sendto() function sends data on the socket with descriptor *socket*. The sendto() call applies to either connected or unconnected sockets.

Parameter
Description

socket The socket descriptor.

buffer The pointer to the buffer containing the message to transmit.

length The length of the message in the buffer pointed to by the *msg* parameter.

flags Setting these flags is not supported in the AF_UNIX domain. The following flags are available:

MSG_OOB
Sends out-of-band data on the socket. Only SOCK_STREAM sockets support out-of-band data. The out-of-band data is a single byte.

Before out-of-band data can be sent between two programs, there must be some coordination of effort. If the data is intended to not be read inline, the recipient of the out-of-band data must specify the recipient of the SIGURG signal that is generated when the out-of-band data is sent. If no recipient is set, no signal is sent. The recipient is set up by using F_SETOWN operand of the fcntl() command, specifying either a pid or gid. For more information on this operand, refer to the fcntl() command.

The recipient of the data determines whether to receive out-of-band data inline or not inline by the setting of the SO_OOBINLINE option of setsockopt(). For more information on receiving out-of-band data, refer to the setsockopt(), recv(), recvfrom() and recvmsg() commands.

MSG_DONTROUTE
The SO_DONTROUTE option is turned on for the duration of the operation. This is usually used only by diagnostic or routing programs.

address The address of the target.

addr_len
The size of the address pointed to by *address*.

If there is not enough available buffer space to hold the socket data to be transmitted, and the socket is in blocking mode, sendto() blocks the caller until additional buffer space becomes available. If the socket is in nonblocking mode, sendto() returns -1 and sets the error code to EWOULDBLOCK. See "fcntl() — Control open file descriptors" on page 474 or "ioctl() — Control device" on page 869 for a description of how to set nonblocking mode.

For datagram sockets, this call sends the entire datagram, provided that the datagram fits into the TCP/IP buffers. Stream sockets act like streams of information with no boundaries separating data. For example, if an application wishes to send 1000 bytes, each call to this function can send 1 byte, or 10 bytes, or the entire 1000 bytes. Therefore, applications using stream sockets should place this call in a loop, calling this function until all data has been sent.

Socket address structure for IPv6: The sockaddr_in6 structure is added to the **netinit/in.h** header. It is used to pass IPv6 specific addresses between applications and the system.

Special behavior for C++: To use this function with C++, you must use the _XOPEN_SOURCE_EXTENDED 1 feature test macro.

Note: The sendto() function has a dependency on the level of the Enhanced ASCII Extensions. See "Enhanced ASCII support" on page 2113 for details.

Returned value

If successful, sendto() returns the number of characters sent.

A value of 0 or greater indicates the number of bytes sent, however, this does not assure that data delivery was complete. A connection can be dropped by a peer socket and a SIGPIPE signal generated at a later time if data delivery is not complete.

No indication of failure to deliver is implied in the return value of this call when used with datagram sockets.

If unsuccessful, sendto() returns -1 and sets errno to one of the following values:

Error Code
: **Description**

EAFNOSUPPORT
: The address family is not supported (it is not AF_UNIX or AF_INET).

EBADF
: *socket* is not a valid socket descriptor.

ECONNREFUSED
: The attempt to connect was rejected.

ECONNRESET
: A connection was forcibly closed by a peer.

EFAULT
: Using the *msg* and *length* parameters would result in an attempt to access storage outside the caller's address space.

EINTR
: A signal interrupted sendto() before any data was transmitted.

EINVAL
: *addr_len* is not the size of a valid address for the specified address family.

EIO
: There has been a network or transport failure.

EMSGSIZE
: The message was too big to be sent as a single datagram. The default is *large-envelope-size*. (Envelopes are used to hold datagrams and fragments during TCP/IP processing. Large envelopes hold UDP datagrams greater than 2KB while they are processed for output, and when they are waiting for an application program to receive them on input.)

ENOBUFS
: Buffer space is not available to send the message.

ENOTCONN
: The socket is not connected.

ENOTSOCK
: The descriptor is for a file, not for a socket.

EOPNOTSUPP
: The *socket* argument is associated with a socket that does not support one or more of the values set in *flags*.

EPIPE
: For a connected stream socket the connection to the peer socket has been lost. A SIGPIPE signal is sent to the calling process.

EPROTOTYPE
The protocol is the wrong type for this socket. A SIGPIPE signal is sent to the calling process.

EWOULDBLOCK
socket is in nonblocking mode and no data buffers are available or the SO_SNDTIMEO timeout value was reached before buffers became available.

The following are for AF_UNIX only:

Error Code
Description

EACCES
Search permission is denied for a component of the path prefix, or write access to the named socket is denied.

EIO An I/O error occurred while reading from or writing to the file system.

ELOOP
Too many symbolic links were encountered in translating the path name in the socket address.

ENAMETOOLONG
A component of a path name exceeded **NAME_MAX** characters, or an entire path name exceeded **PATH_MAX** characters.

ENOENT
A component of the path name does not name an existing file or the path name is an empty string.

ENOTDIR
A component of the path prefix of the path name in the socket address is not a directory.

Related information

- "sys/socket.h" on page 74
- "read() — Read from a file or socket" on page 1376
- "readv() — Read data on a file or socket and store in a set of buffers" on page 1389
- "recv() — Receive data on a socket" on page 1398
- "recvfrom() — Receive messages on a socket" on page 1401
- "recvmsg() — Receive messages on a socket and store in an array of message headers" on page 1405 "select(), pselect() — Monitor activity on files or sockets and message queues" on page 1470
- "send() — Send data on a socket" on page 1494
- "sendmsg() — Send messages on a socket" on page 1500
- "socket() — Create a socket" on page 1688
- "write() — Write data on a file or socket" on page 2085
- "writev() — Write data on a file or socket from an array" on page 2092

__server_classify() — Set classify area field

Standards

Standards / Extensions	C or C++	Dependencies
z/OS UNIX	both	

Format

```
#include <sys/server.h>

int __server_classify(server_classify_t handle,
                      server_classify_field_t field,
                      const char *value);
```

General description

The __server_classify() function sets fields in a classify data area. The 'classify data area' is created and initialized by invoking the __server_classify_create() function. This 'classify data area' is subsequently used with the __server_pwu() function to interface with WorkLoad Manager (WLM).

The *handle* argument is a 'classify data area' created on a previous invocation of the __server_classify_create() function.

The *field* argument must be one of the following values:

_SERVER_CLASSIFY_ACCTINFO
Set the accounting information. When specified, value contains a NULL-terminated character string of up to 143 characters containing the account information for the work unit to be created.

_SERVER_CLASSIFY_COLLECTION
Set the customer defined name for a group of associated packages. When specified, value contains a NULL-terminated character string of up to 18 characters containing the collection name associated with the work unit to be created.

_SERVER_CLASSIFY_CONNECTION
Set the name associated with the environment creating the work unit. When specified, value contains a NULL-terminated character string of up to 8 characters containing the connection name associated with the environment creating the work unit.

_SERVER_CLASSIFY_CONNTKN
Set the connection token that was returned on a call to __ConnectWorkMgr() or __ConnectServerMgr(). When specified, value contains a integer value representing the connection token returned on a call to __ConnectWorkMgr() or __ConnectServerMgr().

_SERVER_CLASSIFY_CORRELATION
Set the name associated with the user/program creating the work unit. When specified, value contains a NULL-terminated character string of up to 12 characters that contains the name associated with the user/program creating the work unit.

_SERVER_CLASSIFY_LUNAME
Set the local LU name associated with the requester. When specified, value contains a NULL-terminated character string of up to 8 characters containing the local LU name associated with the requester.

_SERVER_CLASSIFY_NETID
Set the network ID associated with the requester. When specified, value contains a NULL-terminated character string of up to 8 characters containing the network ID associated with the requester.

_SERVER_CLASSIFY_PACKAGE
Set the package name for a set of associated SQL statements. When

specified, value contains a NULL-terminated character string of up to 8 characters containing the package name associated with the work unit to be created.

_SERVER_CLASSIFY_PERFORM
Set the performance group number (PGN) associated with the work unit. When specified, value contains a NULL-terminated character string of up to 8 characters containing the PGN associated with the work unit to be created.

_SERVER_CLASSIFY_PLAN
Set the access plan name for a set of associated SQL statements. When specified, value contains a NULL-terminated character string of up to 8 characters containing the access plan name associated with the work unit to be created.

_SERVER_CLASSIFY_PRCNAME
Set the DB2® stored SQL procedure name associated with the work unit. When specified, value contains a NULL-terminated character string of up to 18 characters containing the DB2 stored SQL procedure name associated with the work unit to be created.

_SERVER_CLASSIFY_PRIORITY
Set the priority associated with the work unit to be created. When specified, value contains a integer value representing the priority of the work unit to be created.

_SERVER_CLASSIFY_RPTCLSNM@
Set the pointer to an 8 character buffer to receive the output report class name for the work unit to be created. When specified, value contains the pointer to an 8 character buffer to receive the output report class name for the work unit to be created.

_SERVER_CLASSIFY_SCHEDENV
Set the scheduling environment information. When specified, value contains a NULL-terminated character string of up to 16 characters containing the scheduling environment name associated with the work unit.

_SERVER_CLASSIFY_SERVCLS@
Set the pointer to an integer field to receive the output service class for the work unit to be created. When specified, value contains the pointer to a integer field to receive the output service class for the work unit to be created.

_SERVER_CLASSIFY_SERVCLSNM@
Set the pointer to an 8 character buffer to receive the output service class name for the work unit to be created. When specified, value contains the pointer to an 8 character buffer to receive the output service class name for the work unit to be created.

_SERVER_CLASSIFY_SOURCELU
Set the source LU name associated with the requester. When specified, value contains a NULL-terminated character string of up to 17 characters containing the source LU name associated with the requester.

_SERVER_CLASSIFY_SUBCOLN
Set the subsystem collection name. When specified, the value contains a NULL-terminated character string of up to 8 characters, containing the subsystem collection name associated with the work unit.

_SERVER_CLASSIFY_SUBSYSTEM_PARM
Set the transaction subsystem parameter. When specified, *value* contains a NULL-terminated character string of up to 255 characters containing the subsystem parameter being used for the __server_pwu() call.

_SERVER_CLASSIFY_TRANSACTION_CLASS
Set the transaction class. When specified, *value* contains a NULL-terminated character string of up to 8 characters containing the name of the transaction class for the __server_pwu() call.

_SERVER_CLASSIFY_TRANSACTION_NAME
Set the transaction name. When specified, *value* contains a NULL-terminated character string of up to 8 characters containing the name of the transaction for the __server_pwu() call.

_SERVER_CLASSIFY_USERID
Set the user ID. When specified, *value* contains a NULL-terminated character string of up to 8 characters containing the name of the user for the __server_pwu() call.

The *value* argument is the value that the specified 'classify data area' field is to be set to. (For valid values, refer to *z/OS MVS Programming: Workload Management Services, SC34-2663*.)

The classify area is specific to the calling thread. The __server_classify() function call must be done on the same thread of execution as the __server_classify_create(). Use of the classify area by another thread can lead to unpredictable results.

Returned value

If successful, __server_classify() returns 0.

If unsuccessful, __server_classify() returns -1 and sets errno to one of the following values:

Error Code
Description

E2BIG The character string specified for a classify field is too large.

EINVAL
The classify field symbolic is not valid.

Related information

- “sys/server.h” on page 74
- “__server_classify_create() — Create a classify area”
- “__server_classify_destroy() — Delete a classify area” on page 1513
- “__server_classify_reset() — Reset a classify area to an initial state” on page 1514
- “__server_init() — Initialize server” on page 1515
- “__server_pwu() — Process server work unit” on page 1517

__server_classify_create() — Create a classify area

Standards

Standards / Extensions	C or C++	Dependencies
z/OS UNIX	both	

Format

```
#include <sys/server.h>

server_classify_t __server_classify_create(void);
```

General description

The __server_classify_create() function creates a classify data area to be used on a subsequent __server_pwu() or CreateWorkUnit() call. The resulting classify data area can be filled in by calls to the __server_classify() function. The information in the classify area is used to establish the Transaction class, Transaction Name, user ID, and subsystem parameters for the __server_pwu() call or to establish the classification rules for the CreateWorkUnit() call.

The resulting classify area is specific to the calling thread. Use of the classify area by another thread can lead to unpredictable results.

Returned value

If successful, __server_classify_create() returns a *classify_t* which is a handle to the classify area.

If unsuccessful, __server_classify_create() returns -1 and sets errno to one of the following values:

Error Code
    Description

ENOMEM
    Not enough storage is available.

Related information

The classify data area created by this function can be used without serialization only by the creating thread. In addition, storage for this structure is automatically freed at thread termination.
- “sys/server.h” on page 74
- “__server_classify() — Set classify area field” on page 1509
- “__server_classify_destroy() — Delete a classify area”
- “__server_classify_reset() — Reset a classify area to an initial state” on page 1514
- “__server_init() — Initialize server” on page 1515
- “__server_pwu() — Process server work unit” on page 1517

__server_classify_destroy() — Delete a classify area

Standards

Standards / Extensions	C or C++	Dependencies
z/OS UNIX	both	

Format

```
#include <sys/server.h>

void __server_classify_destroy(server_classify_t area);
```

General description

The __server_classify_destroy() function deletes a classify data area previously created by a __server_classify() call. The *area* parameter specifies the classify area to be deleted. Storage for the classify area is freed. This function must be executed by the same thread that created the classify area.

Returned value

__server_classify_destroy() returns no values.

Related information

- "sys/server.h" on page 74
- "__server_classify() — Set classify area field" on page 1509
- "__server_classify_create() — Create a classify area" on page 1512
- "__server_classify_reset() — Reset a classify area to an initial state"
- "__server_init() — Initialize server" on page 1515
- "__server_pwu() — Process server work unit" on page 1517

__server_classify_reset() — Reset a classify area to an initial state

Standards

Standards / Extensions	C or C++	Dependencies
z/OS UNIX	both	

Format

```
#include <sys/server.h>

void __server_classify_reset(server_classify_t area);
```

General description

The __server_classify_reset() function resets a classify data area to its initial state. This is equivalent to destroying the classify area and creating another, and is intended to be a higher performance path for applications which must repeatedly change parameters in a classify area. The *area* parameter specifies the handle of the classify area to be reset, and was previously obtained by a __server_classify() call. This function must be executed by the same thread that created the classify area.

Returned value

__server_classify_reset() returns no values.

Related information

- "sys/server.h" on page 74
- "__server_classify() — Set classify area field" on page 1509
- "__server_classify_create() — Create a classify area" on page 1512
- "__server_classify_destroy() — Delete a classify area" on page 1513
- "__server_init() — Initialize server" on page 1515
- "__server_pwu() — Process server work unit" on page 1517

__server_init() — Initialize server

Standards

Standards / Extensions	C or C++	Dependencies
z/OS UNIX	both	

Format

```
#include <sys/server.h>

int __server_init(int *managertype, const char *subsystype,
                  const char *subsysname, const char *applenv, int paralleleu);
```

General description

The __server_init() function provides the ability for a server address space to connect to WorkLoad Manager (WLM) for the purpose of queueing and servicing work requests.

The parameters supported are:

**managertype*
Points to one or more of the following values that indicate the type of WLM manager the caller is requesting to become. The following are the supported values:

SRV_WORKMGR
Indicates that WLM work management services be made available to the calling address space. This value can be combined with the SRV_QUEUEMGR and SRV_SERVERMGR values.

SRV_QUEUEMGR
Indicates that WLM queue management services be made available to the calling address space. This value can be combined with the SRV_WORKMGR and SRV_SERVERMGR values.

SRV_SERVERMGR
Indicates that WLM server management services be made available to the calling address space. This value can be combined with the SRV_WORKMGR and SRV_QUEUEMGR values.

SRV_SERVERMGRDYNAMIC
Indicates that the WLM work management is to track the number of threads this address space will need. It prepares the address space to support the __service_thread_query() function. For SRV_SERVERMGRDYNAMIC the *paralleleu* parameter has the same effect as it does for SRV_SERVERMGR, in that, it indicates the maximum number of parallel work units that the server can create. The server would initially create some number of threads less than this maximum. The dynamic capability then allows the server to tap into WLM to tell the server when to increase or decrease the number of threads in the address space.

**subsystype*
Points to a NULL-terminated character string containing the generic subsystem type (CICS, IMS, WEB, etc.). When SRV_WORKMGR is

specified for the *managertype* parameter this is the primary category under which WLM classification rules are grouped. The character string can be up to 4 bytes in length.

**subsysname*
: Points to a NULL-terminated character string containing the subsystem name used for classifying work requests when SRV_WORKMGR is specified for the *managertype* parameter. When SRV_SERVERMGR is specified for the *managertype* parameter the subsystem name should match the subsystem name specified on the corresponding call to __server_init() for a work manager (SRV_WORKMGR *managertype*). The character string can be up to 8 bytes in length. When SRV_QUEUEMGR is specified for the *managertype* parameter the combination of the *subsystype* and *subsysname* parameter values must be unique to a single MVS system.

**applenv*
: Points to a NULL-terminated character string that contains the name of the application environment under which work requests are served. The character string can be up to 32 bytes in length. This parameter is only valid when SRV_SERVERMGR is specified for the *managertype* parameter. It should be NULL for all other *managertype* values.

paralleleu
: Specifies the maximum number of tasks within the address space which will be created to process concurrent work requests. This parameter is valid when both or either SRV_SERVERMGR and SRV_SERVERMGRDYNAMIC are specified for the *managertype* parameter. It is ignored for all other *managertype* values.

A successful call to __server_init() results in the calling address space being connected to WLM for the WLM management services requested. Additionally, for a successful server manager WLM connection call (SRV_SERVERMGR *managertype*), the calling process is made a child of, and is placed in the session and process group of the corresponding work manager. The corresponding work manager is the process that called server_init() for the *managertype* combination SRV_WORKMGR+SRV_QUEUEMGR with the same *subsystype* and *subsysname* values specified as the server manager process. This parent/child relationship allows the server manager and the work manager to use signals to communicate with each other.

Returned value

If successful, __server_init() returns 0.

If unsuccessful, __server_init() returns -1 and sets errno to one of the following values:

Error Code
: **Description**

EINVAL
: The *managertype* parameter contains a value that is not correct.

EMVSSAF2ERR
: An error occurred in the security product.

EMVSWLMERROR
: A WLM service failed. Use __errno2() to obtain the WLM service reason code for the failure.

EPERM
The calling thread's address space is not permitted to the BPX.WLMSERVER Facility class. The caller's address space must be permitted to the BPX.WLMSERVER Facility class, if the BPX.WLMSERVER is defined. If BPX.WLMSERVER is not defined, the calling process is not defined as a superuser (UID=0).

Related information

- "sys/server.h" on page 74
- "__server_classify() — Set classify area field" on page 1509
- "__server_classify_create() — Create a classify area" on page 1512
- "__server_classify_destroy() — Delete a classify area" on page 1513
- "__server_classify_reset() — Reset a classify area to an initial state" on page 1514
- "__server_pwu() — Process server work unit"

__server_pwu() — Process server work unit

Standards

Standards / Extensions	C or C++	Dependencies
z/OS UNIX	both	

Format

```
#include <sys/server.h>

int __server_pwu(int fcncode, const char *transclass,
                 const char *applenv, server_classify_t classify,
                 int *appldatalen, void **appldata,
                 struct __srv_fd_list **fdlstruc);
```

General description

The __server_pwu() function provides a general purpose interface for managing and processing work using WorkLoad Manager (WLM) services. The capabilities this service provides include the ability to put work requests onto the WLM work queues, obtain work from the WLM work queues, transfer work to other WLM work servers, end units of work, delete WLM enclaves and refresh WLM work servers.

The parameters supported are:

fcncode
Contains one or more of the following values that indicate the function that is requested:

SRV_PUT_NEWWRK
Indicates that a new work request be put onto the work queue for an application environment server identified by the *applenv* parameter as part of a newly created WLM enclave. This value cannot be combined with any other *fcncode* value.

SRV_PUT_SUBWRK
Indicates that a new work request be put onto the work queue for an application environment server identified by the *applenv* parameter as part of the WLM enclave associated with the calling thread. This value can be combined with the SRV_END_WRK *fcncode* value.

SRV_TRANSFER_WRK
Indicates that the last work request obtained by the calling thread be transferred to the work queue of the target application environment server. As part of the transfer, the calling thread is disassociated from its WLM enclave. This value cannot be combined with any other *fcncode* value.

SRV_GET_WRK
Indicates that a new work request be obtained from the WLM work queue for the calling application environment server. The SRV_GET_WRK *fcncode* also results in the calling thread being associated with the WLM enclave of the work request. If the calling thread is already associated with a WLM enclave due to a prior call to __server_pwu() for SRV_GET_WRK, it is disassociated from the prior WLM enclave, as well as associated with the obtained work request. When the calling thread goes thru task termination or when its process is terminated the work request is ended and the associated WLM enclave is deleted if it is owned by the terminating task or process. The SRV_GET_WRK caller owns the enclave, if the work was queued using the SRV_PUT_NEWWRK or SRV_TRANSFER_WRK functions. If the caller is a thread created using pthread_create (pthread), the thread task owns the enclave. If the caller is not a pthread, the process owns the enclave. This value can be combined with the SRV_END_WRK and SRV_DEL_ENC *fcncode* values.

SRV_REFRESH_WRK
Indicates that the servers associated with the application environments managed by the calling work and queue manager are to be refreshed. This will cause all servers to complete existing work requests and then terminate. New servers will then be started to process new work. This value cannot be combined with any other *fcncode* value.

SRV_END_WRK
Indicates that the calling thread is to be disassociated from its WLM enclave. This value can be combined with the SRV_DEL_ENC, SRV_PUT_SUBWRK and SRV_GET_WRK *fcncode* values.

SRV_DEL_ENC
Indicates that the WLM enclave associated with the calling thread is to be deleted. This value can be combined with the SRV_GET_WRK and SRV_END_WRK *fcncode* values.

SRV_DISCONNECT
Indicates that the calling server's connection to WLM is to be severed. Once a server is disconnected from WLM, it can no longer use this service to process more requests for the application environment it had been connected to (using a call to the server_init() function). If a SRV_DISCONNECT is performed by a work and queue manager, all related server managers implicitly lose their connection to WLM. This also results in the related server managers losing their ability to process more requests using this service.

SRV_DISCONNECT_COND
Indicates that the calling server's connection to WLM is to be severed only if the caller has no more WLM enclaves that it is still

managing. A work and queue manager is still managing an enclave if it has yet to be serviced by a server manager. Once a server is disconnected from WLM, it can no longer use this service to process more requests for the application environment it had been connected to (using a call to the server_init() function). If a SRV_DISCONNECT_COND is performed by a work and queue manager, all related server managers implicitly lose their connection to WLM. This also results in the related server managers losing their ability to process more requests using this service.

***transclass**
Points to a NULL-terminated character string that represents the name of the transaction class to be associated with the work request. This parameter is only valid when the SRV_PUT_NEWWRK *fcncode* parameter value is specified. It should be NULL for the other *fcncode* parameter values. The character string can be up to 8 bytes in length.

***applenv**
Points to a NULL-terminated character string that contains the name of the application environment under which work requests are served. This parameter is valid for the set of SRV_PUT *fcncode* values, the SRV_TRANSFER_WRK *fcncode* value and the SRV_REFRESH_WRK *fcncode* value. It should be NULL for the other *fcncode* parameter values. The character string can be up to 32 bytes in length.

***classify**
Points to a character string that contains the classification information for the work request macro.

***appldatalen**
When one of the SRV_PUT or SRV_TRANSFER *fcncode* parameter values is specified this is a supplied parameter that points to an integer containing the length of the application data specified by the ***appldata* parameter. When the SRV_GET_WRK *fcncode* value is specified, this is an output parameter where the __server_pwu() function is to return the length of the application data associated with the obtained work request. **appldatalen* is only valid when one of the SRV_PUT, SRV_GET_WRK or SRV_TRANSFER *fcncode* parameter values is specified, it is ignored otherwise. The maximum length supported for the application data is 10 megabytes.

****appldata**
When one of the SRV_PUT or SRV_TRANSFER *fcncode* parameter values is specified this is a supplied parameter that points to the application data string. This application data allows the caller to uniquely identify the specific work the caller is requesting. When the SRV_GET_WRK *fcncode* value is specified, this is an output parameter where the __server_pwu() function is to return a pointer to the application data associated with the obtained work request. The returned data area will be an identical copy of the data area that was supplied on the corresponding __server_pwu() call to put the work request on a WLM work queue. ***appldata* is only valid when one of the SRV_PUT, SRV_GET_WRK or SRV_TRANSER *fcncode* parameter values is specified, it is ignored otherwise.

****fdlstruct**
When one of the SRV_PUT or SRV_TRANSFER *fcncode* parameter values is specified the ***fdlstruc* parameter is an input parameter that contains a pointer to a __srv_fd_list structure. The __srv_fd_list structure contains the following members:

```
int        fdcount         count of file descriptiors
int        flags           flag SRV_FDCLOSE
int        fd(SRV_FDS)     file descriptor list
```

The supplied __srv_fd_list structure contains the count of file descriptors to be propagated, followed by the list of file descriptors that are to be propagated to the process that calls server_pwu() to obtain the work request created by the call to this service. If the SRV_FDCLOSE flag is turned on in the flags field of the __srv_fd_list structure, all file descriptors in the list are closed in the calling process. If a NULL pointer is specified, no file descriptors are propagated. When the ***fdlstruc* parameter is used to propagate file descriptors, the caller must ensure that all of the file descriptors in the list are valid open file descriptors in the caller's process, and are not being closed during the processing of this service. If this is not the case, then this function cannot guarantee the proper propagation of the specified file descriptors. When the SRV_GET_WRK *fcncode* parameter value is specified the ***fdlstruc* parameter is an output parameter where the __server_pwu() function returns a pointer to the __srv_fd_list structure associated with the obtained work request. The returned __srv_fd_list structure will contain the count of file descriptors in the returned structure, followed by the list of remapped file descriptor values in the calling process of the file descriptors that were supplied in the __srv_fd_list structure on the corresponding __server_pwu() call to put the work request on a WLM work queue. The flags field in the returned __srv_fd_list structure will be NULL. The ***fdlstruc* parameter is only valid when one of the SRV_PUT, SRV_TRANSFER or SRV_GET_WRK *fcncode* parameter values are specified. It is ignored otherwise. The maximum number of file descriptors supported in the file descriptor list structure is 64.

A successful call to __server_pwu() for the SRV_PUT_NEWWRK *fcncode* not only creates a work request that is placed onto a WLM work queue, but it also creates a new WLM enclave for that work to run in when the work request is obtained. By comparison, the SRV_PUT_SUBWRK and SRV_TRANSFER_WRK *fcncodes*, queue work requests that are part of the existing WLM enclave of the calling thread.

A successful call to __server_pwu() for the SRV_GET_WRK *fcncode* not only results in the caller obtaining a work request from a WLM work queue associated with the caller's application environment, but also results in the calling thread being associated with the WLM enclave associated with the obtained work request.

Usage of the server_pwu function requires the calling address space to have successfully issued a call to the __server_init() function.

For the SRV_PUT_NEWWRK function to run successfully, the caller must have successfully issued a call to the __server_init() service for one of the following *managertype* parameter combinations:

- SRV_WORKMGR + SRV_QUEUEMGR
- SRV_WORKMGR + SRV_QUEUEMGR + SRV_SERVERMGR

For the SRV_PUT_SUBWRK and SRV_TRANSFER_WRK functions to run successfully, the caller must have successfully issued a call to the __server_init() service for one of the following *managertype* parameter combinations:

- SRV_WORKMGR + SRV_QUEUEMGR SRV_SERVERMGR
- SRV_SERVERMGR

For the SRV_GET_WRK, SRV_END_WRK and SRV_DEL_ENC functions to run successfully, the caller must have successfully issued a call to the __server_init() service for one of the following *managertype* parameter combinations:

- SRV_WORKMGR + SRV_QUEUEMGR SRV_SERVERMGR
- SRV_SERVERMGR

For the SRV_REFRESH_WRK function to run successfully, the caller must have successfully issued a call to the __server_init() service for any of the following *managertype* parameter combinations:

- SRV_WORK_MGR + SRV_QUEUE_MGR
- SRV_WORK_MGR + SRV_QUEUE_MGR + SRV_SERVER_MGR

Returned value

If successful, __server_pwu() returns 0.

If unsuccessful, __server_pwu() returns -1 and sets errno to one of the following values:

Error Code
Description

EAGAIN
The requested service could not be performed at the current time. Use __errno2() to obtain the reason code for the failure.

EFAULT
An argument of this service contained an address that was not accessible to the caller.

EINVAL
The *managertype* parameter contains a value that is not correct.

EMVSERR
A MVS environmental or internal error has occurred. Use __errno2() to obtain the exact reason for the failure.

EMVSWLMERROR
A WLM service failed. Use __errno2() to obtain the WLM service reason code for the failure.

Related information

- "sys/server.h" on page 74
- "__server_classify() — Set classify area field" on page 1509
- "__server_classify_create() — Create a classify area" on page 1512
- "__server_classify_destroy() — Delete a classify area" on page 1513
- "__server_classify_reset() — Reset a classify area to an initial state" on page 1514
- "__server_init() — Initialize server" on page 1515

__server_threads_query() — Query the number of threads

Standards

Standards / Extensions	C or C++	Dependencies
z/OS UNIX	both	OS/390 V2R10

Format

```
#include <sys/server.h>

int __server_threads_query(int *threads);
```

General description

Provides information about the number of threads a server should be using for this server environment. After a successful query, *threads* will contain the number of threads that the WorkLoad Manager (WLM) recommends for this address space.

Usage notes

This service is a privileged service that requires the caller to be authorized in one of the following ways:

- Have read access to the BPX.WLMSERVER FACILTY class profile
- Have a UID=0 when the BPX.WLMSERVER FACILTY class profile is not defined

Returned value

If successful, __server_threads_query() returns 0.

If unsuccessful, __server_threads_query() returns -1 and sets errno to one of the following values:

Error Code
Description

EINTR
The wait was interrupted by an unblocked, caught signal. No further waiting will occur for this call. __server_threads_query() can be reissued to begin waiting again.

EPERM
The caller is not permitted to perform the specified operation.

Related information

- "sys/server.h" on page 74

setbuf() — Control buffering

Standards

Standards / Extensions	C or C++	Dependencies
ISO C POSIX.1 XPG4 XPG4.2 C99 Single UNIX Specification, Version 3	both	

Format

```
#include <stdio.h>

void setbuf(FILE *__restrict__stream, char *__restrict__buffer);
```

General description

Controls buffering for the specified *stream*. The *stream* pointer must refer to an open file, and setbuf() must be the first operation on the stream.

If the *buffer* argument is NULL, the *stream* is unbuffered. If not, the buffering mode will be full buffer and the *buffer* must point to a character array of length at least `BUFSIZ`, which is the buffer size defined in the `stdio.h` header file. I/O functions use the *buffer*, which you specify here, for input/output buffering instead of the default system-allocated buffer for the given *stream*. If the buffer does not meet the requirements of the z/OS XL C/C++ product, the buffer is not used.

The setvbuf() function is more flexible than setbuf(), because you can specify the type of buffering and size of buffer.

Note: If you use setvbuf() or setbuf() to define your own buffer for a stream, you must ensure that the buffer is available the whole time that the stream associated with the buffer is in use.

For example, if the buffer is an automatic array (block scope) and is associated with the stream *s*, leaving the block causes the storage to be deallocated. I/O operations of stream *s* are prevented from using deallocated storage. Any operation on *s* would fail because the operation would attempt to access the nonexistent storage.

To ensure that the buffer is available throughout the life of a program, make the buffer a variable allocated at file scope. This can be achieved by using an identifier of type *array* declared at file scope, or by allocating storage (with malloc() or calloc()) and assigning the storage address to a pointer declared at file scope.

VSAM file types do not support unbuffered I/O, causing requests for unbuffered I/O to fail.

Returned value

setbuf() returns no values.

For details about errno values, and about buffers you may have set, see discussions about buffering in *z/OS XL C/C++ Programming Guide*.

Example

CELEBS01

```
/* CELEBS01

   This example opens the file myfile.dat for writing.
   It then calls the &setbuf. function to establish a buffer of
   length BUFSIZ.
   When string is written to the stream, the buffer buf is used
   and contains the string before it is flushed to the file.

 */
#include <stdio.h>

int main(void)
{
   char buf[BUFSIZ];
   char string[] = "hello world";
   FILE *stream;

   stream = fopen("myfile.dat", "wb,recfm=f");

   setbuf(stream,buf);       /* set up buffer */

   fwrite(string, sizeof(string), 1, stream);

   printf("%s\n",buf);       /* string is found in buf now */

   fclose(stream);           /* buffer is flushed out to myfile.dat */
}
```

Related information

- "stdio.h" on page 68
- "fclose() — Close file" on page 473
- "fflush() — Write buffer to file" on page 523
- "fopen() — Open a file" on page 565
- "setvbuf() — Control buffering" on page 1596

setcontext() — Restore user context

Standards

Standards / Extensions	C or C++	Dependencies
XPG4.2 Single UNIX Specification, Version 3	both	POSIX(ON)

Format

```
#define _XOPEN_SOURCE_EXTENDED 1
#include <ucontext.h>

int setcontext(const ucontext_t *ucp);
```

General description

The setcontext() function restores the user context pointed to by *ucp*. A successful call to setcontext() does not return; program execution resumes at the point specified by the *ucp* argument passed to setcontext(). The *ucp* argument should be created either by a prior call to getcontext(), or by being passed as an argument to a signal handler. If the *ucp* argument was created with getcontext(), program execution continues as if the corresponding call of getcontext() had just returned. If the *ucp* argument was modified with makecontext(), program execution continues with the function passed to makecontext(). When that function returns, the process continues as if after a call to setcontext() with the context pointed to by the **uc_link** member of the **ucontext_t** structure if it is not equal to 0. If the **uc_link** member of the **ucontext_t** structure pointed to by the *ucp* argument is equal to 0, then this context is the main context, and the process will exit when this context returns. The effects of passing a *ucp* argument obtained from any other source are undefined.

setcontext() is similar in some respects to siglongjmp() (and longjmp() and _longjmp()). The getcontext()–setcontext() pair, the sigsetjmp()–siglongjmp() pair, the setjmp()–longjmp() pair, and the _setjmp()–_longjmp() pair cannot be intermixed. A context saved by getcontext() should be restored only by setcontext().

Notes:

1. Some compatibility exists with siglongjmp(), so it is possible to use siglongjmp() from a signal handler to restore a context created with getcontext(), but it is not recommended.
2. If the *ucontext* that is input to setcontext() has not been modified by makecontext(), you must ensure that the function that calls getcontext() does not return before you call the corresponding setcontext() function. Calling setcontext() after the function calling getcontext() returns causes unpredictable program behavior.
3. If setcontext() is used to jump back into an XPLINK routine, any alloca() requests issued by the XPLINK routine after the earlier getcontext() was called and before setcontext() is called are backed out. All storage obtained by these alloca() requests is freed before the XPLINK routine is resumed.
4. If setcontext() is used to jump back into a non-XPLINK routine, alloca() requests made after getcontext() and before setcontext() are not backed out.
5. If *ucp* is pointing to a user context of a different execution stack from the current, the user context should be either a freshly modified one (by makecontext()) or the most recently saved one (by getcontext() or swapcontext()) when running on its stack.
6. If *ucp* is pointing to a user context of a different execution stack from the current, the current stack is never collapsed and any resource associated with it is never freed after setcontext() being called.

This function is supported only in a POSIX program.

The **<ucontext.h>** header file defines the **ucontext_t** type as a structure that includes the following members:

```
mcontext_t       uc_mcontext     A machine-specific representation
                                 of the saved context.
ucontext_t      *uc_link         Pointer to the context that will
                                 be resumed when this context returns.
sigset_t         uc_sigmask      The set of signals that are blocked
                                 when this context is active.
stack_t          uc_stack        The stack used by this context.
```

Special behavior for C++: If getcontext() and setcontext() are used to transfer control in a z/OS XL C++ program, the behavior in terms of the destruction of automatic objects is undefined. This applies to both z/OS XL C++ and z/OS XL C/C++ ILC modules. The use of getcontext() and setcontext() in conjunction with try(), catch(), and throw() is also undefined.

Do not issue getcontext() in a C++ constructor or destructor, since the saved context would not be usable in a subsequent setcontext() or swapcontext() after the constructor or destructor returns.

Special behavior for XPLINK-compiled C/C++: Restrictions concerning setjmp.h and ucontext.h:

1. All XPLINK programs compiled with the V2R10 or later C compilers that are to run with Language Environment V2R10 or later libraries and use the **jmp_buf**, **sigjmp_buf** or **ucontext_t** types must not be compiled with C headers from Language Environment V2R9 or earlier.
2. Non-XPLINK functions compiled with any level of Language Environment headers must not define **jmp_buf**, **sigjmp_buf** or **ucontext_t** data items and pass them to XPLINK functions that call getcontext(), longjmp(), _longjmp(), setjmp(), _setjmp(), setcontext(), sigsetjmp(), or swapcontext() with these passed-in data items.

3. When __XPLINK__ is defined, the Language Environment V2R10 and later headers define a larger **jmp_buf**, **sigjmp_buf** or **ucontext_t** area that is required by setjmp(), getcontext(), and related functions when they are called from a XPLINK routine. If __XPLINK__ is not defined, the Language Environment V2R10 and later headers define a shorter **jmp_buf**, **sigjmp_buf** or **ucontext_t** area. The Language Environment headers before V2R10 also define the shorter version of these data areas. If a XPLINK function calls setjmp(), getcontext() or similar functions with a short **jmp_buf**, **sigjmp_buf** or **ucontext_t** area, a storage overlay or program check may occur when the C library tries to store past the end of the passed-in (too short) data area.

Returned value

If successful, setcontext() does not return.

If unsuccessful, setcontext() returns -1.

There are no errno values defined.

Example

This example saves the context in main with the getcontext() statement. It then returns to that statement from the function *func* using the setcontext() statement. Since getcontext() always returns 0 if successful, the program uses the variable *x* to determine if getcontext() returns as a result of setcontext() or not.

```
/* This example shows the usage of getcontext() and setcontext(). */

#define _XOPEN_SOURCE_EXTENDED 1
#include <stdio.h>
#include <ucontext.h>

void func(void);

int  x = 0;
ucontext_t context, *cp = &context;

int main(void) {

  getcontext(cp);
  if (!x) {
    printf("getcontext has been called\n");
    func();
  }
  else {
    printf("setcontext has been called\n");
  }

}

void func(void) {

  x++;
  setcontext(cp);

}
```

Output

```
getcontext has been called
setcontext has been called
```

Related information

- “ucontext.h” on page 82
- “getcontext() — Get user context” on page 694
- “longjmp() — Restore stack environment” on page 1005
- “_longjmp() — Nonlocal goto” on page 1008
- “makecontext() — Modify user context” on page 1028
- “sigaction() — Examine or change a signal action” on page 1611
- “siglongjmp() — Restore the stack environment and signal mask” on page 1640
- “swapcontext() — Save and restore user context” on page 1792

setegid() — Set the effective group ID

Standards

Standards / Extensions	C or C++	Dependencies
POSIX.1a Single UNIX Specification, Version 3	both	

Format

```
#define _POSIX1_SOURCE 2
#include <unistd.h>

int setegid(gid_t gid);
```

General description

Sets the effective group ID (GID) of a process to *gid*, if *gid* is equal to the real GID or the saved set GID of the calling process, or if the process has appropriate privileges. The real GID, the saved set GID, and any supplementary GIDs are not changed.

Returned value

If successful, setegid() returns 0.

If unsuccessful, setegid() returns -1 and sets errno to one of the following values:

Error Code
Description

EINVAL
The value specified for *gid* is incorrect and is not supported by the implementation.

EPERM
The process does not have appropriate privileges, and *gid* does not match the real GID or the saved set GID.

Example

CELEBS02

```
/* CELEBS02

   This example changes your effective GID.

 */
#define _POSIX1_SOURCE 2
#include <unistd.h>
```

```
#include <stdio.h>

main() {
  printf("your effective group id is %d\n", (int) getegid());
  if (setegid(500) != 0)
    perror("setegid() error");
  else
    printf("your effective group id was changed to %d\n",
           (int) getegid());
}
```

Output

```
your effective group id is 512
your effective group id was changed to 500
```

Related information

- "unistd.h" on page 82
- "exec functions" on page 436
- "getegid() — Get the effective group ID" on page 703
- "getgid() — Get the real group ID" on page 708
- "setgid() — Set the group ID" on page 1537

setenv() — Add, delete, and change environment variables

Standards

Standards / Extensions	C or C++	Dependencies
POSIX.1a Single UNIX Specification, Version 3 Language Environment	both	

Format

POSIX - C only:

```
#define _POSIX1_SOURCE 2
#include <env.h>

int setenv(const char *var_name, const char *new_value, int change_flag)
```

Single UNIX Specification, Version 3:

```
#define _UNIX03_SOURCE
#include <stdlib.h>

int setenv(const char *var_name, const char *new_value, int change_flag)
```

Non-POSIX:

```
#include <stdlib.h>

int setenv(const char *var_name, const char *new_value, int change_flag)
```

General description

Adds, changes, and deletes environment variables.

To avoid infringing on the user's name space, the non-POSIX version of this function has two names. One name is prefixed with two underscore characters, and one name is not. The name without the prefix underscore characters is exposed only when you use LANGLVL(EXTENDED).

To use this function, you must either invoke the function using its external entry point name (that is, the name that begins with two underscore characters), or compile with LANGLVL(EXTENDED). When you use LANGLVL(EXTENDED) any relevant information in the header is also exposed.

var_name is a pointer to a character string that contains the name of the environment variable to be added, changed, or deleted. If setenv() is called with *var_name* containing an equal sign ('='), setenv() will fail, and errno will be set to indicate that an invalid argument was passed to the function.

new_value is a pointer to a character string that contains the value of the environment variable named in *var_name*. If *new_value* is a NULL pointer, it indicates that all occurrences of the environment variable named in *var_name* be deleted.

change_flag is a flag that can take any integer value:

Nonzero
Change the existing entry. If *var_name* has already been defined and exists in the environment variable table, its value *will* be updated with *new_value*. If *var_name* was previously undefined, it will be appended to the table.

0
Do not change the existing entry.

If *var_name* has already been defined and exists in the environment variable table, its value will *not* be updated with *new_value*. However, if *var_name* was previously undefined, it will be appended to the table.

Notes:

1. The value of the *change_flag* is irrelevant if *new_value*=`NULL`.
2. You should not define environment variables that begin with '_BPXK_' since they might conflict with variable names defined by z/OS UNIX services. setenv() uses the BPX1ENV callable service to pass environment variables that begin with '_BPXK_' to the kernel.

   Also, do not use '_EDC_' and '_CEE_'. They are used by the runtime library and the Language Environment.

Environment variables set with the setenv() function will only exist for the life of the program, and are not saved before program termination. Other ways to set environment variables are found in “Using Environment Variables” in *z/OS XL C/C++ Programming Guide*.

Special behavior for POSIX C: Under POSIX, setenv() is available if one of the following is true:

- Code is compiled with the compiler option LANGLV(ANSI), uses `#include <env.h>`, and has the POSIX feature tests turned on.
- Code is compiled with `LONGNAME` and prelinked with the OMVS option.

Returned value

If successful, setenv() returns 0.

If unsuccessful, setenv() returns -1 and sets errno to indicate the type of failure that occurred.

Error Code
Description

EINVAL

The name argument is a null pointer, points to an empty string, or points to a string containing an '=' character.

Note: Starting with z/OS V1.9, environment variable _EDC_SUSV3 can be used to control the behavior of setenv() with respect to setting EINVAL when var_name is a null pointer, points to an empty string or points to a string containing an '=' character. By default, setenv() will not set EINVAL for these conditions. When _EDC_SUSV3 is set to 1, setenv() will set errno to EINVAL if one of these conditions is true.

ENOMEM
Insufficient memory was available to add a variable or its value to the environment.

Example

CELEBS03

```
/* CELEBS03

   This example (program 1) sets the environment variable
   _EDC_ANSI_OPEN_DEFAULT.
   A child program (program 2) is then initiated via a system
   call.
   The example illustrates that environment variables are
   propagated forward to a child program, but not backward to
   the parent.

 */
#include <stdio.h>
#include <stdlib.h>

int main(void)
{
   char *x;

   /* set environment variable _EDC_ANSI_OPEN_DEFAULT to "Y" */
   setenv("_EDC_ANSI_OPEN_DEFAULT","Y",1);

   /* set x to the current value of the _EDC_ANSI_OPEN_DEFAULT*/
   x = getenv("_EDC_ANSI_OPEN_DEFAULT");

   printf("program1 _EDC_ANSI_OPEN_DEFAULT = %s\n",
      (x != NULL) ? x : "undefined");

   /* call the child program */
   system("program2");

   /* set x to the current value of the _EDC_ANSI_OPEN_DEFAULT*/
   x = getenv("_EDC_ANSI_OPEN_DEFAULT");

   printf("program1 _EDC_ANSI_OPEN_DEFAULT = %s\n",
      (x != NULL) ? x : "undefined");
}
```

CELEBS04

```
/* CELEBS04

   Program 2:
   A child program of CELEBS03, which is initiated via a system call.

 */
#include <stdio.h>
#include <stdlib.h>

int main(void)
{
   char *x;

   /* set x to the current value of the _EDC_ANSI_OPEN_DEFAULT*/
   x = getenv("_EDC_ANSI_OPEN_DEFAULT");

   printf("program2 _EDC_ANSI_OPEN_DEFAULT = %s\n",
      (x != NULL) ? x : "undefined");

   /* clear the Environment Variables Table */
   setenv("_EDC_ANSI_OPEN_DEFAULT", NULL, 1);

   /* set x to the current value of the _EDC_ANSI_OPEN_DEFAULT*/
   x = getenv("_EDC_ANSI_OPEN_DEFAULT");

   printf("program2 _EDC_ANSI_OPEN_DEFAULT = %s\n",
      (x != NULL) ? x : "undefined");
}
```

Output

```
program1 _EDC_ANSI_OPEN_DEFAULT = Y
program2 _EDC_ANSI_OPEN_DEFAULT = Y
program2 _EDC_ANSI_OPEN_DEFAULT = undefined
program1 _EDC_ANSI_OPEN_DEFAULT = Y
```

Related information

- See the topic about using environment variables in *z/OS XL C/C++ Programming Guide*.
- “stdlib.h” on page 70
- “clearenv() — Clear environment variables” on page 281
- “getenv() — Get value of environment variables” on page 704
- “__getenv() — Get an environment variable” on page 705
- “putenv() — Change or add an environment variable” on page 1348
- “system() — Execute a command” on page 1805

seteuid() — Set the effective user ID

Standards

Standards / Extensions	C or C++	Dependencies
POSIX.1a Single UNIX Specification, Version 3	both	

Format

```
#define _POSIX1_SOURCE 2
#include <unistd.h>

int seteuid(uid_t uid);
```

General description

Sets the effective user ID (UID) to *uid* if *uid* is equal to the real UID or the saved set user ID of the calling process, or if the process has appropriate privileges. The real UID and the saved set UID are not changed.

The seteuid() function is not supported from an address space running multiple processes, since it would cause all processes in the address space to have their security environment changed unexpectedly.

seteuid() can be used by daemon processes to change the identity of a process in order for the process to be used to run work on behalf of a user. In z/OS UNIX, changing the identify of a process is done by changing the real and effective UIDs and the auxiliary groups. In order to change the identity of the process on MVS completely, it is necessary to also change the MVS security environment. The identity change will only occur if the EUID value is specified, changing just the real UID will have no effect on the MVS environment.

The seteuid() function invokes MVS SAF services to change the MVS identity of the address space. The MVS identity that is used is determined as follows:

- If an MVS user ID is already known by the kernel from a previous call to a kernel function (for example, getpwnam()) and the UID for this user ID matches the UID specified on the seteuid() call, then this user ID is used.
- For nonzero target UIDs, if there is no saved user ID or the UID for the saved user ID does not match the UID requested on the seteuid() call, the seteuid() function queries the security database (for example, using getpwnam) to retrieve a user ID. The retrieved user ID is then used.
- If the target UID=0 and a user ID is not known, the seteuid() function always sets the MVS user ID to BPXROOT or the value specified on the SUPERUSER parm in sysparms. BPXROOT is set up during system initialization as a superuser with a UID=0. The BPXROOT user ID is not defined to the BPX.DAEMON FACILITY class profile. This special processing is necessary to prevent a superuser from gaining daemon authority.
- A nondaemon superuser that attempts to set a user ID to a daemon superuser UID fails with an EPERM.

When the MVS identity is changed, the auxiliary list of groups is also set to the list of groups for the new user ID.

If the seteuid() function is issued from multiple tasks within one address space, use synchronization to ensure that the seteuid() functions are not performed concurrently. The execution of seteuid() function concurrently within one address space can yield unpredictable results.

Returned value

If successful, seteuid() returns 0.

If unsuccessful, seteuid() returns -1 and sets errno to one of the following values:

Error Code
Description

EINVAL
The value specified for *uid* is incorrect and is not supported by the implementation.

EPERM
The process does not have appropriate privileges, and *uid* does not match the real UID or the saved set UID.

Example

CELEBS05

```
/* CELEBS05

   This example changes the effective UID.

 */
#define _POSIX1_SOURCE 2
#include <unistd.h>
#include <stdio.h>

main() {
  printf("your effective user id is %d\n", (int) geteuid());
  if (seteuid(25) != 0)
    perror("seteuid() error");
  else
    printf("your effective user id was changed to %d\n",
           (int) geteuid());
}
```

Output

```
your effective user id is 0
your effective user id was changed to 25
```

Related information

- “unistd.h” on page 82
- “geteuid() — Get the effective user ID” on page 707
- “getuid() — Get the real user ID” on page 791
- “setreuid() — Set real and effective user IDs” on page 1571
- “setuid() — Set the effective user ID” on page 1592

__set_exception_handler() — Register an exception handler routine

Standards

Standards / Extensions	C or C++	Dependencies
	both	

Format

```
#include <__le_api.h>

int  __set_exception_handler( void(*exception_handler) (struct __cib *, void *),
                              void * user_data);
```

General description

Restriction: This function is only valid for AMODE 64.

A nonstandard function that registers an 'Exception Handler' function for the current stack frame. Exception Handlers are used to process 'exceptions' at the thread level (unlike signal catchers which process signals at the process level).

Parameter
Description

exception_handle
Address of a function descriptor that is associated with an 'Exception Handler' function.

user_data
A user definable token that will be passed to the 'Exception Handler' function.

Exception Handlers are invoked for the following conditions:

Table 45. Invoked Exception Handlers

Exception	Feedback Code	Message Number	Resulting Signal
Operation	CEE341	CEE3201S	SIGILL
Privileged-operation	CEE342	CEE3202S	SIGILL
Execute	CEE343	CEE3203S	SIGILL
Protection	CEE344	CEE3204S	SIGSEGV
Addressing	CEE345	CEE3205S	SIGSEGV
Specification	CEE346	CEE3206S	SIGILL
Data	CEE347	CEE3207S	SIGFPE
Fixed-point overflow **Note:** Not processed in a C/C++ application.	CEE348	CEE3208S	SIGFPE
Fixed-point divide by zero	CEE349	CEE3209S	SIGFPE
Decimal overflow exception	CEE34A	CEE3210S	SIGFPE
Decimal divide by zero	CEE34B	CEE3211S	SIGFPE
Exponent overflow	CEE34C	CEE3212S	SIGFPE
Exponent underflow **Note:** Not processed in a C/C++ application.	CEE34D	CEE3213S	SIGFPE
Significance **Note:** Not processed in a C/C++ application.	CEE34E	CEE3214S	SIGFPE
Floating-point divide by zero	CEE34F	CEE3215S	SIGFPE
IEEE Binary Floating-Point inexact (truncated)	CEE34G	CEE3216S	SIGFPE
IEEE Binary Floating-Point inexact (incremented)	CEE34H	CEE3217S	SIGFPE
IEEE Binary Floating-Point exponent underflow	CEE34I	CEE3218S	SIGFPE
IEEE Binary Floating-Point exponent underflow inexact (truncated)	CEE34J	CEE3219S	SIGFPE
IEEE Binary Floating-Point exponent underflow inexact (incremented)	CEE34K	CEE3220S	SIGFPE

Table 45. Invoked Exception Handlers (continued)

Exception	Feedback Code	Message Number	Resulting Signal
IEEE Binary Floating-Point exponent overflow	CEE34L	CEE3221S	SIGFPE
IEEE Binary Floating-Point exponent overflow inexact (truncated)	CEE34M	CEE3222S	SIGFPE
IEEE Binary Floating-Point exponent overflow inexact (incremented)	CEE34N	CEE3223S	SIGFPE
IEEE Binary Floating-Point divide by zero	CEE34O	CEE3224S	SIGFPE
IEEE Binary Floating-Point invalid operation	CEE34P	CEE3225S	SIGFPE
Compare and Trap Data Exception	CEE352	CEE3234S	SIGFPE
Vector-processing exception of IEEE invalid operation	CEE354	CEE3236S	SIGFPE
Vector-processing exception of IEEE division-by-zero	CEE355	CEE3237S	SIGFPE
Vector-processing exception of IEEE exponent-overflow	CEE356	CEE3238S	SIGFPE
Vector-processing exception of IEEE exponent-underflow	CEE357	CEE3239S	SIGFPE
Vector-processing exception of IEEE inexact	CEE358	CEE3240S	SIGFPE
Retryable abend	CEE35I	CEE3250C	SIGABND

When one of the exceptions listed above occurs, on a specific thread with a registered Exception Handler, Language Environment will invoke the handler function with the following syntax:

```
void  exception_handler(struct __cib * cib, void * user_data);
```

Parameter
Description

cib Address of the Language Environment Condition Information Block (CIB).

user_data
A user definable token that was specified when the Exception Handler was registered.

An Exception Handler function should never return to Language Environment. It should terminate the thread with pthread_exit(), terminate the process with exit(), or resume execution at a predefined point with setjmp() and longjmp(). If the Exception Handler returns to Language Environment, the thread will be abnormally terminated.

- In a Posix environment only the thread is abnormally terminated, and the thread exit status is set to -1. Equivalent to:

  ```
  pthread_exit( (void *) -1);
  ```

- In a non-Posix environment the entire Language Environment (process as well as thread) is abnormally terminated. Equivalent to:

  ```
  exit(-1);
  ```

Returned value

If successful, __set_exception_handler() returns 0. Otherwise, -1 is returned and errno is set to indicate the error. The following is a possible value for errno:

- EINVAL — The Exception Handler is invalid.

Application usage

1. Multiple Exception Handlers may not be registered for a single stack frame. Only the last one registered is honored.
2. Exception Handlers may be nested, but they must be on different stack frames.
3. Once an Exception Handler is registered, it remains active across calls to nested functions, and will be automatically unregistered once the flow returns from the stack frame in which the call to __set_exception_handler() was invoked.
4. If an Exception Handler is registered, it remains active across subsequence function calls (nested function calls), unless one of the nested functions registers another exception handler. In which case the first exception handler is suspended.
5. Exception Handlers are automatically unregistered when a longjmp() returns to a stack frame earlier on the stack then the frame on which the Exception Handler was registered. All Exception Handlers that are associated with stack frames that are traversed as a result of a longjmp() are automatically unregistered.
6. When an Exception Handler is given control, it is disabled. Any other Exception Handler (that may have been previously registered) is not set active. In other words, when the Exception Handler is given control there is no Exception Handler active. It is suspended.
   - Any exception that occurs while the Exception Handler is executing, will be processed in the same way that Language Environment processes exceptions when no Exception Handlers are present.
   - If the Exception Handler needs to be able to handle exceptions that occur during the execution of the Exception Handler, the handler must invoke __set_exception_handler() to register another (or re-register the same) Exception Handler.
7. One and only one, Exception Handler will be invoked for a condition (the Handler that is active for the stack frame on which the condition (or exception) occurred).
8. If an Exception Handler exists and the condition is one of those listed above, all of the standard Language Environment condition processing is bypassed (including, when POSIX(ON), the mapping of the exception into a signal). Instead, the one active Exception Handler is given control, and it has one opportunity to 'handle' the exception. If it does not handle the exception then abnormal termination will occur.
9. Functions used to affect the processing of signals, have no impact on the processing of Exception Handlers. That is, blocking or ignoring SIGABND, SIGFPE, SIGILL, or SIGSEGV will not prevent Exception Handlers from getting control.

10. In order for Exception Handlers to work, the Language Environment 'TRAP' runtime option must be set on (i.e., TRAP(ON) or TRAP(ON, NOSPIE)).

Related information

- "exit() — End program" on page 443
- "longjmp() — Restore stack environment" on page 1005
- "_longjmp() — Nonlocal goto" on page 1008
- "pthread_exit() — Exit a thread" on page 1267
- "__reset_exception_handler() — Unregister an exception handler routine" on page 1444
- "setjmp() — Preserve stack environment" on page 1546
- "siglongjmp() — Restore the stack environment and signal mask" on page 1640
- "sigsetjmp() — Save stack environment and signal mask" on page 1659

setgid() — Set the group ID

Standards

Standards / Extensions	C or C++	Dependencies
POSIX.1 XPG4 XPG4.2 Single UNIX Specification, Version 3	both	

Format

```
#define _POSIX1_SOURCE 2
#include <unistd.h>

int setgid(gid_t gid);
```

General description

Sets one or more of the group IDs (GIDs) for the current process to *gid*.

If *gid* is the same as the process's real GID or the saved set-group-ID, setgid() always succeeds and sets the effective GID to gid.

If *gid* is not the same as the process's real GID, setgid() succeeds only if the process has appropriate privileges. If the process has such privileges, setgid() sets the real GID, the effective GID, and saved set GID to *gid*.

setgid() does not change any supplementary GIDs of the calling process.

Returned value

If successful, setgid() returns 0.

If unsuccessful, setgid() returns -1 and sets errno to one of the following values:

Error Code
Description

EINVAL
The value of *gid* is incorrect.

EPERM
The process does not have appropriate privileges to set the GID.

Example

CELEBS06

```
/* CELEBS06

   This example sets your GID.

 */
#define _POSIX_SOURCE 1
#include <unistd.h>
#include <stdio.h>

main() {
  printf("your group id is %d\n", (int) getgid());
  if (setgid(500) != 0)
    perror("setgid() error");
  else
    printf("your group id was changed to %d\n",
           (int) getgid());
}
```

Output

```
your group id is 512
your group id was changed to 500
```

Related information

- “sys/types.h” on page 75
- “unistd.h” on page 82
- “exec functions” on page 436
- “getegid() — Get the effective group ID” on page 703
- “getgid() — Get the real group ID” on page 708
- “setuid() — Set the effective user ID” on page 1592

setgrent() — Reset group database to first entry

The information for this function is included in “endgrent() — Group database entry functions” on page 420.

setgroups() — Set the supplementary group ID list for the process

Standards

Standards / Extensions	C or C++	Dependencies
z/OS UNIX	both	

Format

```
#define _OPEN_SYS
#include <sys/types.h>
#include <grp.h>

int setgroups(const int size, const gid_t list[]);
```

General description

setgroups() sets the supplementary group IDs for the process to the list provided in the *list* array. The argument *size* gives the number of `gid_t` elements in *list* array. The maximum number of supplementary groups for a strictly conforming program is **NGROUPS_MAX**, as defined in `<limits.h>` Or, refer to sysconf() (see "sysconf() — Determine system configuration options" on page 1798) for information on dynamically determining the number of supplementary groups allowed.

The caller of this function must be a superuser.

Returned value

If successful, setgroups() returns 0.

If unsuccessful, setgroups() returns -1 and sets errno to one of the following values:

Error Code
Description

EFAULT
The *list* and *size* specify an array that is partially or completely outside of addressable storage for the process.

EINVAL
The *size* parameter is greater than the maximum allowed.

EMVSERR
An MVS environmental or internal error occurred.

EMVSSAF2ERR
The Security Authorization Facility (SAF) had an error.

EPERM
The caller is not authorized, only authorized users are allowed to alter the supplementary group IDs list.

Related information

- "grp.h" on page 32
- "sys/types.h" on page 75
- "getgroups() — Get a list of supplementary group IDs" on page 714
- "initgroups() — Initialize the supplementary group ID list for the process" on page 866

sethostent() — Open the host information data set

Standards

Standards / Extensions	C or C++	Dependencies
XPG4.2 Single UNIX Specification, Version 3	both	

Format

X/Open:

```
#define _XOPEN_SOURCE_EXTENDED 1
#include <netdb.h>

void sethostent(int stayopen);
```

Berkeley sockets:

```
#define _OE_SOCKETS
#include <netdb.h>

void sethostent(int stayopen);
```

General description

The sethostent() function opens and rewinds the local host tables. If the *stayopen* flag is nonzero, the local host tables remain open after each call.

You can use the X_SITE environment variable to specify different local host tables and override those supplied by the z/OS global resolver during initialization.

Note: For more information on these local host tables or the environment variables, see *z/OS Communications Server: IP Configuration Guide, SC27-3650*.

Special behavior for C++: To use this function with C++, you must use the _XOPEN_SOURCE_EXTENDED 1 feature test macro.

Returned value

sethostent() returns no values.

Related information

- “netdb.h” on page 49
- “endhostent() — Close the host information data set” on page 421
- “endnetent() — Close network information data sets” on page 422
- “gethostbyaddr() — Get a host entry by address” on page 717
- “gethostbyname() — Get a host entry by name” on page 719
- “gethostent() — Get the next host entry” on page 721

setibmopt() — Set IBM TCP/IP image

Standards

Standards / Extensions	C or C++	Dependencies
z/OS UNIX	both	

Format

```
#define _OPEN_SYS_SOCK_EXT
#include <sys/socket.h>

int setibmopt(int cmd, struct ibm_tcpimage *bfrp);
```

General description

The setibmopt() function call is used to set TCP/IP options. Currently, the only supported command is IBMTCP_IMAGE which allows the setibmopt() to choose the active TCP/IP image stack the application will connect to.

To reset ibm_tcpimage to nothing chosen, set the *name* to all blanks.

The chosen transport is inherited over fork() and preserved over exec(). If this is not desired, the child process should call setibmopt() with a blank name to reset the TCP/IP image for the child.

Parameter
Description

cmd The value in *cmd* must be set to the command to be performed. Currently, only IBMTCP_IMAGE is supported and must be paired with the *bfrp* parameter as described.

bfrp The pointer to a ibm_tcpimage structure.

To set the TCP/IP image for a socket, the application should set values in the ibm_tpcimage structure as follows:

Element
Description

status 0 means is not known and need not be checked. Currently, this is the only value with meaning.

version
0 means the version is to be set on return if known.

name The name must be left justified, uppercase, padded with blanks, and be the name of an active TCP stack.

Returned value

If successful, setibmopt() returns 0.

If unsuccessful, setibmopt() returns -1 and sets errno to one of the following values:

Error Code
Description

EFAULT
Using the *bfrp* supplied would result in access of a storage location that is inaccessible.

EIBMBADTCPNAME
A name of a PFS was specified that either is not configured or is not a Sockets PFS.

EOPNOTSUPP
The *cmd* is a function that is not supported.

ENXIO
The name that was specified did not match an AF_INET socket stack, but Common Inet is not configured on this system. Because this system does not have multiple AF_INET socket transports configured, there is already a natural affinity to one single stack, and this failure may not be a problem for the application.

Related information

- “sys/socket.h” on page 74
- “__iptcpn() — Retrieve the resolver supplied jobname or user ID” on page 894

setibmsockopt() — Set IBM specific options associated with a socket

Standards

Standards / Extensions	C or C++	Dependencies
z/OS UNIX	both	

Format

```
#define _OPEN_SYS_SOCK_EXT
#include <sys/socket.h>

int setibmsockopt(int s, int level, int optname, char *optval, size_t optlen);
```

General description

These options are only valid on IBM systems and can be specified to allow improved processing of requests on sockets.

Parameter
Description

s The socket descriptor.

level The level for which the option is set.

optname
The name of a specified socket option. The socket option currently available is:

- SO_EioIfNewTP

optval The pointer to option data.

optlen The length of the option data.

Usage notes

The SO_EioIfNewTP option allows a socket application that has bound INADDRANY to be notified if a new common inet transport provider was activated after the socket was created. In order to activate this option, the option data should have a value of 1. To deactivate this option, supply a value of 0 for the option data. This option can be useful to a server that is listening - waiting for requests to come in from a number of sources. When a new transport provider is activated while this option is in effect, the application program will receive an EIO on the next accept, select or read request. Once this happens, the application should close the current socket and create a new one - thus enabling the socket to communicate with the new transport provider.

Returned value

If successful, setibmsockopt() returns 0.

If unsuccessful, setibmsockopt() returns -1 and sets errno to one of the following values:

Error Code
Description

EBADF
The *s* parameter is not a valid socket descriptor.

EFAULT
Using *optval* and *optlen* parameters would result in an attempt to access storage outside the caller's address space.

ENOPROTOOPT
The *optname* parameter is unrecognized. The *level* parameter is not SOL_SOCKET. The domain of the socket descriptor is not AF_INET. The socket descriptor is not a datagram type socket.

Related information

- “sys/socket.h” on page 74
- “fcntl() — Control open file descriptors” on page 474
- “getibmsockopt() — Get IBM specific options associated with a socket” on page 725
- “getsockopt() — Get the options associated with a socket” on page 777
- “ibmsflush() — Flush the application-side datagram queue” on page 824
- “setsockopt() — Set options associated with a socket” on page 1578

setipv4sourcefilter() — Set source filter

Standards

Standards / Extensions	C or C++	Dependencies
RFC3678	both	z/OS V1.9

Format

```
#define _OPEN_SYS_SOCK_EXT3
#include <netinet/in.h>

int setipv4sourcefilter(int s, struct in_addr interface, struct in_addr group,
                        uint32_t fmode, uint32_t numsrc, struct in_addr *slist);
```

General description

This function allows applications to set and replace the current multicast filtering state for a tuple consisting of socket, interface, and multicast group values.

A multicast filter is described by a filter mode, which is MCAST_INCLUDE or MCAST_EXCLUDE, and a list of source addresses which are filtered.

This function is IPv4-specific, must be used only on AF_INET sockets with an open socket of type SOCK_DGRAM or SOCK_RAW.

If the function is unable to obtain the required storage, control will not return to the caller. Instead the application will terminate due to an out of memory condition (if the reserve stack is available and the caller is not XPLINK), or it will terminate with an abend indicating that storage could not be obtained.

Argument
Description

s Identifies the socket.

interface
Holds the local IP address of the interface.

group Holds the IP multicast address of the group.

fmode Identifies the filter mode. The value of this field must be either MCAST_INCLUDE or MCAST_EXCLUDE, which are likewise defined in <netinet/in.h>.

numsrc
Holds the number of source addresses in the slist array.

slist Points to an array of IP addresses of sources to include or exclude depending on the filter mode.

Returned value

If successful, the function returns 0. Otherwise, it returns -1 and sets errno to one of the following values.

errno Description

EADDRNOTAVAIL
The specified interface address is incorrect for this host, or the specified interface address is not multicast capable.

EBADF
s is not a valid socket descriptor.

EINVAL
Interface or group is not a valid IPv4 address, or the socket s has already requested multicast setsockopt options.

ENOBUFS
The number of the source addresses exceeds the allowed limit.

EPROTOTYPE
The socket s is not of type SOCK_DGRAM or SOCK_RAW.

Related information

- “netinet/in.h” on page 53
- “getipv4sourcefilter() — Get source filter” on page 728

setitimer() — Set value of an interval timer

Standards

Standards / Extensions	C or C++	Dependencies
XPG4.2	both	POSIX(ON)

Format

```
#define _XOPEN_SOURCE_EXTENDED 1
#include <sys/time.h>

int setitimer(int which, struct itimerval *value, struct itimerval *ovalue);
```

General description

setitimer() sets the value of an interval timer. An interval timer is a timer which sends a signal after each repetition (interval) of time.

The *which* argument indicates what kind of time is being controlled. Values for *which* are:

ITIMER_REAL
This timer is marking real (clock) time. A SIGALRM signal is generated after each interval of time.

Note: alarm() also sets the real interval timer.

ITIMER_VIRTUAL
This timer is marking process virtual time. Process virtual time is the amount of time spent while executing in the process, and can be thought of as a CPU timer. A SIGVTALRM signal is generated after each interval of time.

ITIMER_PROF
This timer is marking process virtual time plus time spent while the system is running on behalf of the process. A SIGPROF signal is generated after each interval of time.

Note: In a multithreaded environment, each of the above timers is specific to a thread of execution for both the generation of the time interval and the measurement of time. For example, an a ITIMER_VIRTUAL timer will mark execution time for just the thread, not the entire process.

The *value* argument points to an `itimerval` structure containing the timer value to be set. The structure contains:

`it_interval`
timer interval

When `it_interval` is nonzero, it is used as the value which `it_value` is initialized to after each timer expiration. If `it_interval` is zero, the timer is disabled **after the next expiration**, subject to the value in `it_value`.

`it_value`
current timer value to be set

When `it_value` is nonzero, it is used as the initial value to establish the timer with, that is, the time to the next timer expiration. If `it_value` is zero, the timer is **immediately** disabled.

The *ovalue* argument points to an `itimerval` structure in which the current value of the timer is returned. If *ovalue* is a NULL pointer, the current timer value is not returned. The structure contains:

`it_interval`
current timer interval

`it_value`
current timer value

For both `itimerval` structures, each of the fields (`it_interval` and `it_value`) is a `timeval` structure, and contains:

`tv_sec` seconds since January 1, 1970 Coordinated Universal Time (UTC)

`tv_usec`
microseconds

Returned value

If successful, setitimer() returns 0, and if *ovalue* was non-NULL, *ovalue* points to the `itimerval` structure containing the old timer values.

If unsuccessful, setitimer() returns -1 and sets errno to one of the following values:

Error Code
Description

EINVAL
which is not a valid timer type, or the *value* argument has an incorrect (noncanonical) form. The `tv_seconds` field must be a nonnegative integer, and the `tv_usec` field must be a nonnegative integer in the range of 0-1,000,000.

Usage of the `ITIMER_PROF` timer generates a SIGPROF signal which may interrupt an in-progress function. Thus, programs using this timer may need to be able to restart an interrupted function.

Related information

- "sys/time.h" on page 75
- "alarm() — Set an alarm" on page 156
- "getitimer() — Get value of an interval timer" on page 729
- "gettimeofday(), gettimeofday64() — Get date and time" on page 790
- "sleep() — Suspend execution of a thread" on page 1678
- "ualarm() — Set the interval timer" on page 1939
- "usleep() — Suspend execution for an interval" on page 1966

setjmp() — Preserve stack environment

Standards

Standards / Extensions	C or C++	Dependencies
ISO C POSIX.1 XPG4 XPG4.2 C99 Single UNIX Specification, Version 3	both	

Format

```
#include <setjmp.h>

int setjmp(jmp_buf env);
```

General description

Saves a stack environment that can subsequently be restored by longjmp(). The setjmp() and longjmp() functions provide a way to perform a nonlocal `goto`. They are often used in signal handlers.

A call to setjmp() causes it to save the current stack environment in *env*. A subsequent call to longjmp() restores the saved environment and returns control to a point corresponding to the setjmp() call. The values of all variables, except register variables and nonvolatile automatic variables, accessible to the function receiving control, contain the values they had when longjmp() was called. The values of register variables are unpredictable. Nonvolatile *auto* variables that are changed between calls to setjmp() and longjmp() are also unpredictable.

An invocation of setjmp() must appear in one of the following contexts only:

- The entire controlling expression of a selection or iteration statement.
- One operand of a relational or equality operator with the other operand an integral constant expression, with the resulting expression being the entire controlling expression of a selection or iteration statement.
- The operand of a unary “!” operator with the resulting expression being the entire controlling expression of a selection or iteration.
- The entire expression of an expression statement (possibly cast to void).

Note: Ensure that the function that calls setjmp() does not return before you call the corresponding longjmp() function. Calling longjmp() after the function calling setjmp() returns causes unpredictable program behavior.

Special behavior for POSIX C: To save and restore a stack environment that includes a signal mask, use sigsetjmp() and siglongjmp(), instead of setjmp().

The sigsetjmp()—siglongjmp() pair, the setjmp()—longjmp() pair, the _setjmp()—_longjmp() pair, and the getcontext()—setcontext() pair *cannot* be intermixed. A stack environment saved by setjmp() can only be restored by longjmp().

Special behavior for C++: If setjmp() and longjmp() are used to transfer control in a z/OS XL C++ program, the behavior in terms of the destruction of automatic objects is undefined. This applies both to z/OS XL C++ and z/OS XL C/C++ ILC modules. The use of setjmp() and longjmp() in conjunction with try(), catch(), and throw() is also undefined.

Special behavior for XPG4.2: In a program that was compiled with the feature test macro _XOPEN_SOURCE_EXTENDED defined, another pair of functions, _setjmp()—_longjmp(), are available. On this implementation, these calls are functionally identical to setjmp()—longjmp(). Therefore it is possible, but not recommended, to intermix the setjmp()—longjmp() pair with the _setjmp()—_longjmp() pair.

Special behavior for XPLINK-compiled C++: Restrictions concerning setjmp.h and ucontext.h:

1. All XPLINK programs compiled with the V2R10 or later C compilers that are to run with Language Environment V2R10 or later libraries and use the **jmp_buf**, **sigjmp_buf** or **ucontext_t** types must not be compiled with C headers from Language Environment V2R9 or earlier.
2. Non-XPLINK functions compiled with any level of Language Environment headers must not define **jmp_buf**, **sigjmp_buf** or **ucontext_t** data items and pass them to XPLINK functions that call getcontext(), longjmp(), _longjmp(), setjmp(), _setjmp(), setcontext(), sigsetjmp(), or swapcontext() with these passed-in data items.
3. When __XPLINK__ is defined, the Language Environment V2R10 and later headers define a larger **jmp_buf**, **sigjmp_buf** or **ucontext_t** area that is required by setjmp(), getcontext(), and related functions when they are called from an XPLINK routine. If __XPLINK__ is not defined, the Language Environment V2R10 and later headers define a shorter **jmp_buf**, **sigjmp_buf** or **ucontext_t** area. The Language Environment headers before V2R10 also define the shorter version of these data areas. If an XPLINK function calls setjmp(), getcontext() or similar functions with a short **jmp_buf**, **sigjmp_buf** or **ucontext_t** area, a storage overlay or program check may occur when the C library tries to store past the end of the passed-in (too short) data area.

Returned value

setjmp() returns 0 after saving the stack environment.

If setjmp() returns as a result of a longjmp() call, it returns the *value* argument of longjmp(), or the value 1 if the *value* argument of longjmp() is equal to 0.

Example

This example stores the stack environment at the statement:

```
if(setjmp(mark) != 0) ...
```

When the system first performs the `if` statement, it saves the environment in *mark* and sets the condition to `FALSE` because setjmp() returns 0 when it saves the environment. The program prints the message: `setjmp has been called`.

The subsequent call to function *p* tests for a local error condition, which can cause it to perform the longjmp() function. Then control returns to the original setjmp() function using the environment saved in *mark*. This time the condition is `TRUE` because -1 is the returned value from the longjmp() function. The program then performs the statements in the block and prints: `longjmp has been called`. Finally, the program calls the `recover` function and exits.

```
/* This example shows the effect of having set the stack environment.  */
#include <stdio.h>
#include <setjmp.h>

jmp_buf mark;

void p(void);
void recover(void);

int main(void)
{
   if (setjmp(mark) != 0) {
      printf("longjmp has been called\n");
      recover();
      exit(1);
      }
   printf("setjmp has been called\n");
⋮
   p();
⋮
}

void p(void)
{
   int error = 0;
⋮
   error = 9;
⋮
   if (error != 0)
      longjmp(mark, -1);
⋮
}

void recover(void)
{
⋮
}
```

Related information

- "setjmp.h" on page 62
- "getcontext() — Get user context" on page 694
- "longjmp() — Restore stack environment" on page 1005
- "_longjmp() — Nonlocal goto" on page 1008
- "setcontext() — Restore user context" on page 1524
- "_setjmp() — Set jump point for a nonlocal goto"
- "sigsetjmp() — Save stack environment and signal mask" on page 1659
- "siglongjmp() — Restore the stack environment and signal mask" on page 1640
- "swapcontext() — Save and restore user context" on page 1792

_setjmp() — Set jump point for a nonlocal goto

Standards

Standards / Extensions	C or C++	Dependencies
XPG4.2 Single UNIX Specification, Version 3	both	

Format

```
#define _XOPEN_SOURCE_EXTENDED 1
#include <setjmp.h>

int _setjmp(jmp_buf env);
```

General description

The _setjmp() function saves a stack environment that can subsequently be restored by _longjmp(). The _setjmp() and _longjmp() functions provide a way to perform a nonlocal `goto`. They are often used in signal handlers.

A call to _setjmp() causes it to save the current stack environment in *env*. A subsequent call to _longjmp() restores the saved environment and returns control to a point corresponding to the _setjmp() call. The values of all variables, except register variables, and except nonvolatile automatic variables, accessible to the function receiving control contain the values they had when _longjmp() was called. The values of register variables are unpredictable. Nonvolatile *auto* variables that are changed between calls to _setjmp() and _longjmp() are also unpredictable.

An invocation of _setjmp() must appear in one of the following contexts only:

1. The entire controlling expression of a selection or iteration statement.
2. One operand of a relational or equality operator with the other operand an integral constant expression, with the resulting expression being the entire controlling expression of a selection or iteration statement.
3. The operand of a unary "!" operator with the resulting expression being the entire controlling expression of a selection or iteration.
4. The entire expression of an expression statement (possibly cast to void).

The X/Open standard states that _setjmp() and _longjmp() are functionally identical to longjmp() and setjmp(), respectively, with the addition restriction that _setjmp() and _longjmp() do not manipulate the signal mask. However, on this implementation longjmp() and setjmp() do not manipulate the signal mask. So on this implementation _setjmp() and _longjmp() are literally identical to longjmp() and setjmp(), respectively.

To save and restore a stack environment, including the current signal mask, use sigsetjmp() and siglongjmp() instead of _setjmp() and _longjmp(), or setjmp() and longjmp().

The _setjmp()—_longjmp() pair, the setjmp()—longjmp() pair, the sigsetjmp()—siglongjmp() pair, and the getcontext()—setcontext() pair cannot be intermixed. A stack environment saved by _setjmp() can be restored only by _longjmp().

Notes:

1. However, on this implementation, since the _setjmp()—_longjmp() pair are functionally identical to the setjmp()—longjmp() pair it is possible to intermix them, but it is not recommended.
2. Ensure that the function that calls _setjmp() does not return before you call the corresponding _longjmp() function. Calling _longjmp() after the function calling _setjmp() returns causes unpredictable program behavior.

Special behavior for C++: If _setjmp() and _longjmp() are used to transfer control in a z/OS XL C++ program, the behavior in terms of the destruction of automatic objects is undefined. This applies both to z/OS XL C++ and z/OS XL C/C++ ILC modules. The use of _setjmp() and _longjmp() in conjunction with try(), catch(), and throw() is also undefined.

Special behavior for XPLINK-compiled C++: Restrictions concerning setjmp.h and ucontext.h:

1. All XPLINK programs compiled with the V2R10 or later C compilers that are to run with Language Environment V2R10 or later libraries and use the **jmp_buf**, **sigjmp_buf** or **ucontext_t** types must not be compiled with C headers from Language Environment V2R9 or earlier.
2. Non-XPLINK functions compiled with any level of Language Environment headers must not define **jmp_buf**, **sigjmp_buf** or **ucontext_t** data items and pass them to XPLINK functions that call getcontext(), longjmp(), _longjmp(), setjmp(), _setjmp(), setcontext(), sigsetjmp(), or swapcontext() with these passed-in data items.
3. When __XPLINK__ is defined, the Language Environment V2R10 and later headers define a larger **jmp_buf**, **sigjmp_buf** or **ucontext_t** area that is required by setjmp(), getcontext(), and related functions when they are called from an XPLINK routine. If __XPLINK__ is not defined, the Language Environment V2R10 and later headers define a shorter **jmp_buf**, **sigjmp_buf** or **ucontext_t** area. The Language Environment headers before V2R10 also define the shorter version of these data areas. If an XPLINK function calls setjmp(), getcontext() or similar functions with a short **jmp_buf**, **sigjmp_buf** or **ucontext_t** area, a storage overlay or program check may occur when the C library tries to store past the end of the passed-in (too short) data area.

Returned value

_setjmp() returns 0 after saving the stack environment.

If _setjmp() returns as a result of a _longjmp() call, it returns the *value* argument of _longjmp(), or 1 if the *value* argument of _longjmp() was 0.

Related information

- “setjmp.h” on page 62
- “getcontext() — Get user context” on page 694

- "longjmp() — Restore stack environment" on page 1005
- "_longjmp() — Nonlocal goto" on page 1008
- "setcontext() — Restore user context" on page 1524
- "setjmp() — Preserve stack environment" on page 1546
- "siglongjmp() — Restore the stack environment and signal mask" on page 1640
- "sigsetjmp() — Save stack environment and signal mask" on page 1659
- "swapcontext() — Save and restore user context" on page 1792

setkey() — Set encoding key

Standards

Standards / Extensions	C or C++	Dependencies
XPG4 XPG4.2 Single UNIX Specification, Version 3	both	

Format

```
#define _XOPEN_SOURCE
#include <stdlib.h>

void setkey(const char *key);
```

General description

The setkey() function transforms the *key* argument array into data encryption keys which are used by the encrypt() function to encode blocks of data.

The *key* argument of setkey() is an array of length 64 bytes containing only the bytes with numerical value of 0 and 1. If this 64 byte array is divided into groups of 8, the low-order byte of each group is ignored. The setkey() function transforms the remaining 56 bytes, each with values 0 or 1, into 16 48-bit keys according to the Data Encryption Standard (DES) key algorithm.

Special behavior for z/OS UNIX Services: When setkey() is called from a thread, the array of 16 bit-bit keys produced by setkey() is unique to the thread. Thus, for each thread from which the encrypt() function is called by a threaded application, the setkey() function must first be called from each thread.

Returned value

setkey() returns no values.

Special behavior for z/OS UNIX Services: The setkey() function will fail if:

Error Code
Description

EINVAL
64 byte input array contains bytes with values other than 0x00 or 0x01.

ENOMEM
Unable to allocate storage for DES keys on thread from which setkey() invoked.

Note: Because setkey() returns no values, applications wishing to check for errors should set errno to 0, call setkey(), then test errno and, if it is nonzero, assume an error has occurred.

Related information

- "stdlib.h" on page 70
- "__cnvblk() — Convert block" on page 294
- "crypt() — String encoding function" on page 345
- "encrypt() — Encoding function" on page 419

setlocale() — Set locale

Standards

Standards / Extensions	C or C++	Dependencies
ISO C POSIX.1 XPG4 XPG4.2 SAA Language Environment z/OS UNIX C99 Single UNIX Specification, Version 3	both	

Format

```
#include <locale.h>

char *setlocale(int category, const char *locale);
```

General description

Sets, changes, or queries *locale* categories or groups of categories. It does this action according to values of the *locale* and *category* arguments.

A *locale* is the complete definition of the part of a user's program that depends on language and cultural conventions. You can accept the default value of *locale*, or you can set it to one of the supplied locales listed in the appendix , "Supplied Locales", in *z/OS XL C/C++ Programming Guide*. Some examples of the supplied locales are: "C", "POSIX", "SAA", "S370", "Fr_BE.IBM-1047", "En_GB.IBM-285", "En_US.IBM_1047", "Fr_BE.IBM-1148@euro", and "Fr_BE.IBM-1148".

Note that non-POSIX programs may exploit the POSIX style of locale support. This use of environment variables also applies to non-POSIX programs that use POSIX locale support.

Effect of setlocale() on Language Environment: The current locale set with the setlocale() function affects only some C library functions. (See Table 46 on page 1553). It does not affect the CEE locale set and query functions available under Language Environment and described in the *IBM Language Environment Programming Reference*.

The category argument

The *category* argument may be set to one of these values:

Table 46. Values for Category Arguments of setlocale()

Category	Purpose
LC_ALL	Specifies all categories associated with the program's locale.
LC_COLLATE	Defines the collation sequence, that is, the relative order of collation elements (characters and multicharacter collation elements) in the program's locale. The collation sequence definition is used by regular expression, pattern matching, and sorting functions. These string functions are affected by the defined collation sequence: strcoll(), strxfrm(), wcscoll(), and wcsxfrm(). LC_CTYPE, LC_COLLATE, and LC_SYNTAX should refer to the same locale. Changing just one of them may invalidate another.
LC_CTYPE	Defines character classification and case conversion for characters in the program's locale. Affects the behavior of character-handling functions defined in the ctype.h header file: csid(), isalnum(), isalpha(), isblank(), iswblank() iscntrl(), isdigit(), isgraph(), islower(), isprint(), ispunct(), isspace(), isupper(), iswalnum(), iswalpha(), iswcntrl(), iswctype(), iswdigit(), iswgraph(), iswlower(), iswprint(), iswpunct(), iswspace(), iswupper(), iswxdigit(), isxdigit(), tolower(), toupper(), towlower(), towupper(), wcsid(), and wctype(). Affects behavior of the printf() and scanf() families of functions: fprintf(), printf(), sprintf(), fscanf(), scanf(), and sscanf(). Affects the behavior of wide-character input/output functions: fgetwc(), fgetws(), getwc(), getwchar(), fputwc(), fputws(), putwc(), putwchar(), and ungetwc(). Affects the behavior of multibyte and wide-character functions: mblen(), mbtowc(), mbstowcs(), wctomb(), wcstombs(), mbrlen(), mbrtowc(), mbsrtowcs(), wcrtomb(), wcsrtombs(), wcswidth(), wcwidth(), wcstod(), wcstol(), and wcstoul(). LC_CTYPE, LC_COLLATE, and LC_SYNTAX should refer to the same locale. Changing just one of them may invalidate another.
LC_MESSAGES	Under z/OS XL C/C++ support, it affects the messages returned by the nl_langinfo() function and it also has an effect on rpmatch(). The LC_MESSAGES *category* will *not* affect the messages for the following functions: perror(), strerror(), and regerror(). In the locale of a C program running with POSIX(ON), it defines affirmative and negative response patterns.
LC_MONETARY	Affects monetary information returned by localeconv() and the strfmon() function. It defines the rules and symbols used to format monetary numeric information in the program's locale. The formatting rules and symbols are strings. localeconv() returns pointers to these strings with names found in the locale.h header file.

Table 46. Values for Category Arguments of setlocale() (continued)

Category	Purpose
LC_NUMERIC	Affects the decimal-point character for the formatted input/output and string conversion functions, and the non-monetary formatting information returned by the localeconv() function, specifically: • The printf() family of functions • The scanf() family of functions • strtod() • atof() The formatting rules and symbols are strings. localeconv() returns pointers to the strings with names found in the locale.h header file.
LC_TIME	Defines time and date format information in the program's locale used by the strftime(), strptime(), and wcsftime() functions.
LC_SYNTAX	Affects the behavior of functions that use encoded values to format characters: • printf() family of functions • scanf() family of functions • regcomp() • strfmon() LC_SYNTAX also affects values that may be retrieved using the getsyntx() function. LC_CTYPE, LC_COLLATE, and LC_SYNTAX should refer to the same locale. Changing just one of them may invalidate another.
LC_TOD	Affects the behavior of the functions related to time zone and Daylight Savings Time information in the program's locale, when time zone and Daylight Savings Time information is not defined by the TZ environment variable. This information is used by ctime(), localtime(), mktime(), and strftime().

For a POSIX program, the functions ctime(), localtime(), mktime(), setlocale(), and strftime() call the tzset() function to override LC_TOD category information when TZ is defined and valid.

The locale argument

Identifies the locale. For a list of locales provided by IBM, see the topic about supplied locales in *z/OS XL C/C++ Programming Guide*.

If the value of an environment variable is used, it must be a valid locale name. If this is the case, setlocale() sets the specified category to the named locale, and returns a string giving the name of the locale. Otherwise, setlocale() does not change the program's locale and returns a NULL pointer. Valid *category* names include names of locales provided by IBM. Also, names of locales, which are created using the z/OS XL C/C++ locale definition mechanism, are valid.

The null-string ("") locale value: If "" is specified, the locale-related environment variables are checked. If the locale name is not defined by the environment variables, the default is "S370" when running POSIX(OFF) and "C" when running POSIX(ON).

For both C and C++ languages, and whether you are using POSIX or not, if a program using POSIX-style locale support specifies "" for the value of *locale*, then setlocale() interrogates locale-related environment variables in the program's environment to find a locale name or names to use. The locale name is chosen according to the first of the following conditions that applies:

1. If the environment variable LC_ALL is defined and is not NULL, the value of LC_ALL is used. That value is applied to all categories.
2. If individual environmental variables are defined, then their values are used for the categories.
3. If the environment variable LANG is defined and is not NULL, the value of LANG is used.
4. If no non-NULL environment variable is present to supply a value, "C" is used.

If a program using POSIX-style locale support specifies LC_ALL for the value of *category* and "" for the value of *locale*, setlocale() searches environment variables in the way just described to obtain a locale name for each category. If all the locale names obtained identify valid locales, setlocale() sets each category to the appropriate locale and returns a string naming the locale associated with each category. Otherwise, setlocale() does not change the program's locale and returns a NULL pointer.

Default locale: The relationship between the POSIX C and SAA C locales is as follows.

Using C or C++ languages with the runtime option POSIX(OFF):

1. The SAA C locale definition is the default. "C", "SAA", and "S370" are synonyms for the SAA C locale definition, which is prebuilt into the library.

   The source file `EDC$SAAC LOCALE` is provided for reference, but cannot be used to alter the definition of this prebuilt locale.
2. Issuing `setlocale(`*`category`*`, "")` has the following effect:
   - Locale-related environment variables are checked to find the name of locales) to use to set the *category* specified. Querying the locale with `setlocale(`*`category`*`, NULL)` returns the name of the locales specified by the appropriate environment variables.
   - If no non-NULL environment variable is present, it is the equivalent of having issued `setlocale(`*`category`*`, "S370")`. That is, the locale chosen is the SAA C locale definition, and querying the locale with `setlocale(`*`category`*`, NULL)`, returns "S370" as the locale name.
3. If no setlocale() function is issued or `setlocale(LC_ALL, "C")` is used, then the locale chosen is the prebuilt SAA C locale, and querying the locale with `setlocale(`*`category`*`, NULL)`, returns "C" as the locale name.
4. For `setlocale(LC_ALL,"SAA")`, the locale chosen is the prebuilt SAA C locale, and querying the locale with `setlocale(`*`category`*`, NULL)`, returns "SAA" as the locale name.
5. For `setlocale(LC_ALL,"S370")`, the locale chosen is the prebuilt SAA C locale, and querying the locale with `setlocale(`*`category`*`, NULL)`, returns "S370" as the locale name.
6. For `setlocale(LC_ALL,"POSIX")`, the locale chosen is the prebuilt POSIX C locale, and querying the locale with `setlocale(`*`category`*`, NULL)`, returns "POSIX" as the locale name.

Using z/OS XL C with the runtime option POSIX(ON):

1. The POSIX C locale definition is the default. "C" and "POSIX" are synonyms for the POSIX C locale definition, which is prebuilt into the library.

   The source file `EDC$POSX LOCALE` is provided for reference, but cannot be used to alter the definition of this prebuilt locale.
2. Issuing `setlocale(`*`category`*`, "")` has the following effect:
   - Locale-related environment variables are checked to find the name of locales that can set the *category* specified. Querying the locale with `setlocale(`*`category`*`, NULL)` returns the name of the locale specified by the appropriate environment variables.
   - If no non-NULL environment variable is present, the result is equivalent to having issued `setlocale(`*`category`*`,"C")`. That is, the locale chosen is the POSIX C locale definition, and querying the locale with `setlocale(`*`category`*`, NULL)`, returns "C" as the locale name.
3. If no setlocale() function is issued or if `setlocale(LC_ALL, "C")` is used, the locale chosen is the prebuilt POSIX C locale. Querying the locale with `setlocale(`*`category`*`, NULL)` returns "C" as the locale name.
4. For `setlocale(LC_ALL,"POSIX")` the locale chosen is the prebuilt POSIX C locale. Querying the locale with `setlocale(`*`category`*`, NULL)` returns "POSIX" as the locale name.
5. For `setlocale(LC_ALL,"SAA")` the locale chosen is the prebuilt SAA C locale. Querying the locale with `setlocale(`*`category`*`, NULL)` returns "SAA" as the locale name.
6. For `setlocale(LC_ALL,"S370")` the locale chosen is the prebuilt SAA C locale. Querying the locale with `setlocale(`*`category`*`, NULL)` returns "S370" as the locale name.

The setlocale() function supports locales built by using the localedef utility, as well as locales built using the assembler language source and produced by the EDCLOC macro. Find more information about old format locales in "Internationalization: Locales and Character Sets", in *z/OS XL C/C++ Programming Guide*.

Special behavior for z/OS UNIX Services

The LOCPATH environment variable specifies a colon separated list of HFS directories. If LOCPATH is defined, setlocale() searches HFS directories in the order specified by LOCPATH for locale object files it requires. Locale object files in the HFS are produced by the localedef utility running under z/OS UNIX. If LOCPATH is not defined and setlocale() is called by a POSIX program, setlocale() looks in the default HFS locale directory, /usr/lib/nls/locale, for locale object files it requires. If setlocale() does not find a locale object it requires in the HFS, it converts the locale name to a PDS member name as described in *z/OS XL C/C++ Programming Guide* and searches locale PDS load libraries associated with the program calling setlocale().

Locale names may be file names, relative path names, or absolute path names. LOCPATH is used if file name rather than path name is specified. Also, // preceding a file name tells setlocale() to skip HFS search and to convert the name to a load module name of the form EDC$xxxx, and to search MVS load libraries for a member to load with this name. Also, // preceding a file name tells setlocale() to skip HFS search, to convert the name to a load module PDS name. XPLINK locale object PDS names begin with EDC. Non-XPLINK locale object load module names begin with CEH. See *z/OS XL C/C++ Programming Guide*, section titled Locale Naming Conventions for further information regarding locale object names.

All locales supplied by IBM come in two versions: non-XPLINK and XPLINK. The HFS-resident XPLINK locale objects are distinguished from their non-XPLINK versions by an ".xplink" suffix on the HFS path name. PDS-resident XPLINK locale objects are distinguished from their non-XPLINK versions by a "CEH" prefix. The non-XPLINK PDS-resident locale objects have a prefix of "EDC".

It is the convention to specify locales using the locale descriptive names as they are listed in Appendix D of *z/OS XL C/C++ Programming Guide*. The runtime loads the non-XPLINK or XPLINK locale as appropriate.

It is also possible to specify a locale's relative or full path name on the setlocale call. However, the runtime does nothing to ensure the locale is the appropriate version. Setlocale uses the locale object exactly as specified if it is a relative or fully qualified path name. For example, setlocale() will fail if it is given an XPLINK locale full path name but the application is a non-XPLINK application. Similarly, setlocale() will fail if it is given a non-XPLINK locale full path name but the application is an XPLINK application. These problems are avoided if the locale names are the descriptive locale names.

Invocation sequence for setlocale()

In all three variations of the setlocale() function call, a pointer to a string that represents the locale value is returned. Also, in all variations, if the value for either *category* or *locale* is invalid, setlocale() returns a NULL pointer and the operating environment is not changed.

Each variation causes a different function to be performed:

1. `setlocale(category, locale);`

   When an explicit locale is named, the *category* named in the call is set according to the named locale.

2. `setlocale(category, "");`

   When the locale argument of the setlocale() function is given as a NULL string (""), the setlocale() function sets the locale environment according to the environment variables. If these are not set, the default locale "S370" is used. This locale may be customized when the z/OS XL C/C++ product is installed. See "Using Environment Variables" in *z/OS XL C/C++ Programming Guide*.

   The environment variables are not currently supported under all z/OS XL C/C++ environments. The processing above will allow the setlocale() function to use the environment variables if they are available, and to use the "S370" locale otherwise.

3. `setlocale(category, (char *) 0);`

   When a NULL pointer is given as a locale, a pointer to a string that represents the current locale for the specified *category* is returned. The string has the property that if it were specified as the locale of a subsequent setlocale() call of the same *category*, the current locale would be restored. For example, the following sequence is effectively a no-op:

   ```
   setlocale(category, setlocale(category, (char *) 0));
   ```

When called with a NULL string (for example, `setlocale(LC_ALL,"")`, setlocale() determines the locale to be set, using the environment variables, and checking them in this order:

1. LC_ALL. If set, it specifies the name for all categories; it can override the values in the other environmental variables.
2. LC_COLLATE, LC_CTYPE, LC_MESSAGES, LC_MONETARY, LC_NUMERIC, LC_TIME, LC_SYNTAX, and LC_TOD. If set, these variables specify the locale name for the given *category*.
3. LANG.

The setlocale() function uses the getenv() function to retrieve the environment variables if the system supports the getenv() function. Under CICS it is not supported.

Querying the locale

When *locale* is set to a NULL pointer, setlocale() returns a string indicating the program's locale without changing it. This provides a means to query the program's current locale. To query the locale, give a NULL pointer as the second parameter. For example, to query all the categories of your locale, use a statement like the following:

```
char *string = setlocale(LC_ALL, NULL);
```

Returned value

If successful, setlocale() returns a pointer to the string associated with the specified *category* for the new *locale*. The string can be used on a subsequent call to restore that part of the program's locale.

Note: Because the string that a successful call to setlocale() points to may be overwritten by subsequent calls to the setlocale() function, you should copy the string if you plan to use it later.

If unsuccessful, setlocale() returns a NULL pointer and the program's locale is not changed.

If successful, setlocale() returns a string whose contents depend on the values of the *category* and *locale* arguments as shown in the following table.

Table 47. Return String as Determined by Category and Locale Values

Category Value	Locale Value	Return String
Specific category	NULL pointer	current-locale-name for category
	new-locale-name	new-locale-name for category
	"" (Null string)	If the environmental variables are set, new-locale-name: environment-variable-value or C. If the environmental variables are not set, and if non-POSIX Program, then S370 or SAA.

Table 47. Return String as Determined by Category and Locale Values (continued)

Category Value	Locale Value	Return String
LC_ALL	NULL pointer	One of these: • locale-name • locale-name-list: locale-name1, locale-name2, ..., if different names for one or more categories.
	new-locale-name	new-locale-name (same for all categories)
	"" (Null string)	One of these: • new-locale-name: environment-variable-value or C • locale-name-list: environment-variable-value-list if different names for one or more categories. If environmental variables are not set, and if non-POSIX program, then S370 (same for all categories).

If the string returned contains a locale name list, the names have the following order:

1. LC_COLLATE locale-name
2. LC_CTYPE locale-name
3. LC_MONETARY locale-name
4. LC_NUMERIC locale-name
5. LC_TIME locale-name
6. LC_TOD locale-name
7. LC_MESSAGES locale-name
8. LC_SYNTAX locale-name

If unsuccessful, setlocale() returns a NULL pointer and does not change the program's locale. Failure can result if:

- An incorrect *category* value is used.
- An incorrect *locale* value is used.
- The value of the environment variable used by setlocale() when the value of *locale* is "" is an undefined or incorrect locale name.

Note: If setlocale() is called and an application has called pthread_create() to create another thread, setlocale() returns a NULL pointer and does not change the current locale.

Example

CELEBS07

```
/* CELEBS07

   This example sets the locale of the program to be
   Fr_FR.IBM-1047 and prints the string that is associated with
   the locale.

 */
#include <stdio.h>
```

```
#include <locale.h>

char *string;

int main(void)
{
   string = setlocale(LC_ALL, "Fr_FR.IBM-1047");
   if (string != NULL)
      printf(" %s \n",string);
}
```

CELEBS08

```
/* CELEBS08

   This example uses &setenv. to set the value of the
   environment variable LC_TIME to FRAN, calls &setloc. to set
   all categories to default values, uses &setloc. to query all
   categories, and uses &printf. to print results.

 */
#include <stdio.h>
#include <stdlib.h>
#include <env.h>
#include <locale.h>

int main(void)
{

   char *string;
   setenv("LC_TIME", "FRAN", 1);
   setlocale(LC_ALL, "");
   string = setlocale(LC_ALL, NULL);
   printf("string = %s \n", string);
}
```

Output

If the example is run with POSIX(OFF), the result of printf() is:

```
string = "S370,S370,S370,S370,FRAN,S370,S370,S370"
```

If the example is run with POSIX(ON), the result of printf() is:

```
string = "C,C,C,C,FRAN,C,C,C"
```

Example

The following example shows euro currency support:

```
/* EUROSAMP
   This example sets the locale of the program to be
   Fr_BE.IBM-1148 and Fr_BE.IBM-1148@euro and prints
   the string associated with each locale.
*/

#include <stdio.h>
#include <locale.h>

int main(void)
{
   char *string;

   string = setlocale(LC_ALL,"Fr_BE.IBM-1148");
   if (string != NULL)
      printf("String = %s \n",string);
   string = setlocale(LC_ALL,"Fr_BE.IBM-1148@euro");
   if (string != NULL)
      printf("String = %s \n",string);
}
```

Output

```
String = Fr_BE.IBM-1148
String = Fr_BE.IBM-1148@euro
```

Related information

- “localdef.h” on page 40
- “locale.h” on page 40
- “getenv() — Get value of environment variables” on page 704
- “localeconv() — Query numeric conventions” on page 982
- “nl_langinfo() — Retrieve locale information” on page 1147

setlogmask() — Set the mask for the control log

Standards

Standards / Extensions	C or C++	Dependencies
XPG4.2 Single UNIX Specification, Version 3	both	

Format

```
#define _XOPEN_SOURCE_EXTENDED 1
#include <syslog.h>

int setlogmask(int maskpri);
```

General description

The setlogmask() function sets the log priority mask for the current process to *maskpri* and returns the previous mask. If the *maskpri* argument is 0 (zero), the current log mask is not modified. Calls by the current process to the syslog() function with a priority not set in *maskpri* are rejected. The mask for an individual priority *pri* is calculated by the macro `LOG_MASK(pri)` ; The mask for all priorities up to and including *toppri* is given by the macro `LOG_UPTO(toppri)` The default log mask allows all priorities to be logged.

Returned value

If successful, setlogmask() returns the value of the previous mask setting.

No errors are defined.

Related information

- “syslog.h” on page 72
- “closelog() — Close the control log” on page 292
- “openlog() — Open the system control log” on page 1162
- “syslog() — Send a message to the control log” on page 1803

setnetent() — Open the network information data set

Standards

Standards / Extensions	C or C++	Dependencies
XPG4.2 Single UNIX Specification, Version 3	both	

Format

X/Open:

```
#define _XOPEN_SOURCE_EXTENDED 1
#include <netdb.h>

void setnetent(int stayopen);
```

Berkeley sockets:

```
#define _OE_SOCKETS
#include <netdb.h>

void setnetent(int stayopen);
```

General description

The setnetent() function opens and rewinds the *tcpip*.HOSTS.ADDRINFO data set, which contains information about known networks. If the *stayopen* flag is nonzero, the *tcpip*.HOSTS.ADDRINFO remains open after each call to setnetent().

You can use the **X_ADDR** environment variable to specify a data set other than *tcpip*.HOSTS.ADDRINFO.

Note: For more information on these data sets and environment variables, *tcpip*.HOSTS.ADDRINFO, see *z/OS Communications Server: IP Configuration Guide, SC27-3650*.

Special behavior for C++: To use this function with C++, you must use the _XOPEN_SOURCE_EXTENDED 1 feature test macro.

Returned value

setnetent() returns no values.

Related information

- “netdb.h” on page 49
- “endhostent() — Close the host information data set” on page 421
- “endnetent() — Close network information data sets” on page 422
- “getnetbyaddr() — Get a network entry by address” on page 741
- “getnetbyname() — Get a network entry by name” on page 742
- “getnetent() — Get the next network entry” on page 744

set_new_handler() — Register a function for set_new_handler()

Standards

Standards / Extensions	C or C++	Dependencies
ISO/ANSI C++	C++ only	

Format

```
#include <new>

new_handler set_new_handler(new_handler ph) throw();
```

General description

The set_new_handler() function is part of the z/OS XL C++ error handling mechanism. If you have registered a new-handler function with set_new_handler(), that new-handler function will be called by the new operator if it is unable to allocate storage. If you have not registered a new-handler function, the default behavior is for the new operator to return NULL.

The argument supplied to set_new_handler() is of type new_handler as defined in the header <new> (that is, a pointer to a function with a void return type and no arguments).

For C++ applications that are compiled NOXPLINK, the variable containing the address of the new handler function is statically bound with the executable. This means that each executable has its own new handler function which is shared only by the other functions that are linkedited as part of that executable. This is true even if multiple threads are using that same executable. This means that you cannot issue a set_new_handler() from within a non-XPLINK DLL if the new handler function is to be invoked outside of that DLL.

For C++ applications that are compiled XPLINK, the new handler function is truly global, so the DLL restriction is lifted. In a multithreaded environment consisting of XPLINK executables, the new handler function created by a call to set_new_handler() still applies to all threads in the (POSIX) process.

The required behavior of a new handler is to perform one of the following operations:
- Make more storage available for allocation and then return.
- Call either abort() or exit(int).
- Throw an object of type bad_alloc.

Returned value

Returns a value of type new_handler. The function pointed to is the function that was previously called by the set_new_handler() function, or NULL if a new handler function was not established.

Refer to *z/OS XL C/C++ Language Reference* for more information about z/OS XL C++ error handling, including the new operator and the set_new_handler() functions.

Related information

- "new" on page 56

setpeer() — Preset the socket peer address

Standards

Standards / Extensions	C or C++	Dependencies
z/OS UNIX	both	

Format

X/Open:

```
#define _XOPEN_SOURCE_EXTENDED 1
#include <sys/socket.h>

int setpeer(int socket, struct sockaddr *address, int length, char *name);
```

Berkeley sockets:

```
#define _OE_SOCKETS
#include <sys/socket.h>

int setpeer(int socket, struct sockaddr *address, int length, char *name);
```

General description

The setpeer() function presets the peer address associated with a socket.

Note: Neither AF_INET, AF_UNIX, nor AF_INET6 support this function.

Parameter
Description

socket The socket descriptor.

address The address of the socket peer.

length The length of the socket address.

name The name of a field indicating the conditions of the peer request.

Special behavior for C++: To use this function with C++, you must use the _XOPEN_SOURCE_EXTENDED 1 feature test macro.

Returned value

setpeer() always returns -1.

Error Code
Description

EINVAL
The request is invalid or not supported.

Related information

- "sys/socket.h" on page 74

setpgid() — Set process group ID for job control

Standards

Standards / Extensions	C or C++	Dependencies
POSIX.1 XPG4 XPG4.2 Single UNIX Specification, Version 3	both	

Format

```
#define _POSIX_SOURCE
#include <unistd.h>

int setpgid(pid_t pid, pid_t pgid);
```

General description

Sets the process group ID (PGID) of a process within the session of the calling process, so you can reassign a process to a different process group, or start a new process group with the specified process as its group leader.

`pid_t` *pid* is the process ID (PID) of the process whose PGID you want to change. This must either be the caller of setpgid() or one of its children, and it must be in the caller's session. It cannot be the PID of a session leader. If *pid* is zero, the system uses the PID of the process calling setpgid().

`pid_t` *pgid* is the new PGID you want to assign to the process identified by *pid*. If *pgid* indicates an existing process group, it must be in the caller's session. If *pgid* is zero, the system uses the PID of the process indicated by *pid* as the ID for the new process group. The new group is created in the caller's session.

Returned value

If successful, setpgid() returns 0.

If unsuccessful, setpgid() returns -1 and sets errno to one of the following values:

Error Code
Description

EACCES
The value of *pid* matches the PID of a child of the calling process, but the child has successfully run one of the EXEC functions.

EINVAL
pgid is less than zero or has some other unsupported value.

EPERM
The caller cannot change the PGID of the specified process. Some possible reasons are:

- The specified process is a session leader.
- *pid* matches the PID of a child of the calling process, but the child is not in the same session as the caller.
- *pgid* does not match the PID of the process specified by *pid*, and it does not match the PGID of any other process in the caller's session.

ESRCH
pid does not match the PID of the calling process or any of its children.

Example

CELEBS09

```
/* CELEBS09

   This example sets the PGID.

 */
#define _POSIX_SOURCE
#include <unistd.h>
#include <sys/types.h>
#include <stdio.h>

main() {
  pid_t pid;
  int p1[2], p2[2];
  char c='?';

  if (pipe(p1) != 0)
    perror("pipe() #1 error");
  else if (pipe(p2) != 0)
    perror("pipe() #2 error");
  else
    if ((pid = fork()) == 0) {
      printf("child's process group id is %d\n", (int) getpgrp());
      write(p2[1], &c, 1);
      puts("child is waiting for parent to complete task");
      read(p1[0], &c, 1);
      printf("child's process group id is now %d\n", (int) getpgrp());
      exit(0);
    }
    else {
      printf("parent's process group id is %d\n", (int) getpgrp());
      read(p2[0], &c, 1);
      printf("parent is performing setpgid() on pid %d\n", (int) pid);
      if (setpgid(pid, 0) != 0)
        perror("setpgid() error");
      write(p1[1], &c, 1);
      printf("parent's process group id is now %d\n", (int) getpgrp());
      sleep(5);
    }
}
```

Output

```
parent's process group id is 5767174
child's process group id is 5767174
parent is performing setpgid() on pid 131084
parent's process group id is now 5767174
child is waiting for parent to complete task
child's process group id is now 131084
```

Related information

- “unistd.h” on page 82
- “exec functions” on page 436
- “getpgrp() — Get the process group ID” on page 751
- “setpgrp() — Set process group ID” on page 1567
- “setsid() — Create session, set process group ID” on page 1576
- “tcsetpgrp() — Set the foreground process group ID” on page 1858

setpgrp() — Set process group ID

Standards

Standards / Extensions	C or C++	Dependencies
XPG4.2 Single UNIX Specification, Version 3	both	

Format

```
#define _XOPEN_SOURCE_EXTENDED 1
#include <unistd.h>

pid_t setpgrp(void);
```

General description

If the calling process is not already a session leader, setpgrp() sets the process group ID of the calling process to the process ID of the calling process. If a new process group is created, it is created within the session of the calling process.

Returned value

If successful, setpgrp() returns the new process group ID.

If unsuccessful, setpgrp() returns -1 and sets errno to one of the following values:

Error Code
  Description

EPERM
  The calling process is a session leader.

Related information

- "unistd.h" on page 82
- "exec functions" on page 436
- "getpgrp() — Get the process group ID" on page 751
- "setpgid() — Set process group ID for job control" on page 1565
- "setsid() — Create session, set process group ID" on page 1576
- "tcsetpgrp() — Set the foreground process group ID" on page 1858

setpriority() — Set process scheduling priority

Standards

Standards / Extensions	C or C++	Dependencies
XPG4.2 Single UNIX Specification, Version 3	both	

Format

```
#define _XOPEN_SOURCE_EXTENDED 1
#include <sys/resource.h>

int setpriority(int which, id_t who, int priority);
```

General description

setpriority() sets the scheduling priority of a process, process group or user.

Processes are specified by the values of the *which* and *who* arguments. The *which* argument may be any one of the following set of symbols defined in the *sys/resource.h* include file:

PRIO_PROCESS
indicates that the *who* argument is to be interpreted as a process ID

PRIO_PGRP
indicates that the *who* argument is to be interpreted as a process group ID

PRIO_USER
indicates that the *who* argument is to be interpreted as a user ID

The *who* argument specifies the ID (process, process group, or user). A 0 (zero) value for the *who* argument specifies the current process, process group or user ID.

The *priority* argument specifies the scheduling priority. It is specified as a signed integer in the range, -20 to 19. Negative priorities cause more favorable scheduling. The default priority is 0. If the value specified to setrlimit() is less than the system's lowest supported priority value, the system's lowest supported value is used; if it is greater than the system's highest supported value, the system's highest supported value is used. The setting of a process's scheduling priority value has the equivalent effect on a process's nice value, since they both represent the process's relative CPU priority. For example, setting one's scheduling priority value to its maximum value (19) has the equivalent effect of increasing one's nice value to its maximum value ((2*NZERO)-1), and will be reflected on the nice(), getpriority() and setpriority() functions.

If more than one process is specified, setpriority() sets the priorities of all of the specified processes to the specified value.

Only a process with appropriate privilege can lower its priority.

Returned value

If successful, setpriority() returns 0.

If unsuccessful, setpriority() returns -1 and sets errno to one of the following values:

Error Code
Description

EACCES
The priority is being changed to a lower value and the current process does not have the appropriate privilege.

EINVAL
The symbol specified in the *which* argument was not recognized, or the value of the *who* argument is not a valid process ID, process group ID or user ID.

ENOSYS
The system does not support this function.

EPERM
A process was located, but neither the real nor effective user ID of the executing process match the effective user ID of the process whose priority is to be changed.

ESRCH
No process could be located using the *which* and *who* argument values specified.

Related information

- "sys/resource.h" on page 74
- "getpriority() — Get process scheduling priority" on page 755
- "nice() — Change priority of a process" on page 1145

setprotoent() — Open the protocol information data set

Standards

Standards / Extensions	C or C++	Dependencies
XPG4.2 Single UNIX Specification, Version 3	both	

Format

X/Open:

```
#define _XOPEN_SOURCE_EXTENDED 1
#include <netdb.h>

void setprotoent(int stayopen);
```

Berkeley sockets:

```
#define _OE_SOCKETS
#include <netdb.h>

void setprotoent(int stayopen);
```

General description

The setprotoent() function opens and rewinds the */etc/protocol* or the *tcpip*.ETC.PROTO data set. If the *stayopen* flag is nonzero, the */etc/protocol* or the *tcpip*.ETC.PROTO data set remains open after each call.

Special behavior for C++

To use this function with C++, you must use the _XOPEN_SOURCE_EXTENDED 1 feature test macro.

Returned value

setprotoent() returns no values.

Related information

- "netdb.h" on page 49
- "endprotoent() — Work with a protocol entry" on page 423
- "getprotobyname() — Get a protocol entry by name" on page 756

- "getprotobynumber() — Get a protocol entry by number" on page 758
- "getprotoent() — Get the next protocol entry" on page 759

setpwent() — Reset user database search

The information for this function is included in "endpwent() — User database functions" on page 423.

setregid() — Set real and effective group IDs

Standards

Standards / Extensions	C or C++	Dependencies
XPG4.2 Single UNIX Specification, Version 3	both	

Format

```
#define _XOPEN_SOURCE_EXTENDED 1
#include <unistd.h>

int setregid(gid_t rgid, gid_t egid);
```

General description

The setregid() function sets the real and/or effective GIDs for the calling process to the values specified by the input real and effective GID values. If a specified value is equal to -1, the corresponding real or effective GID of the calling process is left unchanged.

A process with appropriate privileges can set the real and effective GID to any valid GID value. An unprivileged process can only set the effective GID if the EGID argument is equal to either the real, effective, or saved GID of the process. An unprivileged process can only set the real GID if the RGID argument is equal to either the real, effective, or saved GID of the process.

If the setregid() function is issued from multiple tasks within one address space, use synchronization to ensure that the setregid() functions are not performed concurrently. The execution of setregid() function concurrently within one address space can yield unpredictable results.

The setregid() function does not change any supplementary GIDs of the calling process.

Returned value

If successful, setregid() returns 0.

If unsuccessful, neither of the group IDs will be changed, setregid() returns -1 and sets errno to one of the following values:

Error Code
Description

EINVAL
The value of the *rgid* or *egid* argument is invalid or out-of-range.

EMVSSAF2ERR
The SAF call IRRSSU00 incurred an error.

EPERM
The processes does not have appropriate privileges and a change other than changing the real group ID to the saved set-group-ID, or changing the effective group ID to the real group ID or the saved group ID, was requested.

Related information

- “unistd.h” on page 82
- “exec functions” on page 436
- “getuid() — Get the real user ID” on page 791
- “setreuid() — Set real and effective user IDs”
- “setuid() — Set the effective user ID” on page 1592

setreuid() — Set real and effective user IDs

Standards

Standards / Extensions	C or C++	Dependencies
XPG4.2 Single UNIX Specification, Version 3	both	

Format

```
#define _XOPEN_SOURCE_EXTENDED 1
#include <unistd.h>

int setreuid(uid_t ruid, uid_t euid);
```

General description

The setreuid() function sets the real and/or effective UIDs for the calling process to the values specified by the input real and effective UID values. If a specified value is equal to -1, the corresponding real or effective UID of the calling process is left unchanged.

A process with appropriate privileges can set the real and effective UID to any valid UID value. An unprivileged process can only set the effective UID if the EUID argument is equal to either the real, effective, or saved UID of the process. An unprivileged process can only set the real UID if the RUID argument is equal to either the real, effective, or saved UID of the process.

The setreuid() function is not supported from an address space running multiple processes, since it would cause all processes in the address space to have their security environment changed unexpectedly.

setreuid() can be used by daemon processes to change the identity of a process in order for the process to be used to run work on behalf of a user. In z/OS UNIX, changing the identify of a process is done by changing the real and effective UIDs and the auxiliary groups. In order to change the identity of the process on MVS completely, it is necessary to also change the MVS security environment. The identity change will only occur if the EUID value is specified, changing just the real UID will have no effect on the MVS environment.

The setreuid() function invokes MVS SAF services to change the MVS identity of the address space. The MVS identity that is used is determined as follows:

- If an MVS user ID is already known by the kernel from a previous call to a kernel function (for example, getpwnam()) and the UID for this user ID matches the UID specified on the setreuid() call, then this user ID is used.
- For nonzero target UIDs, if there is no saved user ID or the UID for the saved user ID does not match the UID requested on the setreuid() call, the setreuid() function queries the security database (for example, using getpwnam()) to retrieve a user ID. The retrieved user ID is then used.
- If the target UID=0 and a user ID is not known, the setreuid() function always sets the MVS user ID to BPXROOT or the value specified on the SUPERUSER parm in sysparms. BPXROOT is set up during system initialization as a superuser with a UID=0. The BPXROOT user ID is not defined to the BPX.DAEMON FACILITY class profile. This special processing is necessary to prevent a superuser from gaining daemon authority.
- A nondaemon superuser that attempts to set a user ID to a daemon superuser UID fails with an EPERM.

When the MVS identity is changed, the auxiliary list of groups is also set to the list of groups for the new user ID.

If the setreuid() function is issued from multiple tasks within one address space, use synchronization to ensure that the setreuid() functions are not performed concurrently. The execution of setreuid() function concurrently within one address space can yield unpredictable results.

Returned value

If successful, setreuid() returns 0.

If unsuccessful, neither of the group IDs will be changed, setreuid() returns -1, and sets errno to one of the following values:

Error Code
Description

EINVAL
The value of the *rgid* or *egid* argument is invalid or out-of-range.

EMVSSAF2ERR
The SAF call IRRSSU00 incurred an error.

EPERM
The processes does not have appropriate privileges and a change other than changing the real group ID to the saved set-group-ID, or changing the effective group ID to the real group ID or the saved group ID, was requested.

Related information

- “unistd.h” on page 82
- “exec functions” on page 436
- “getuid() — Get the real user ID” on page 791
- “seteuid() — Set the effective user ID” on page 1531
- “setuid() — Set the effective user ID” on page 1592

setrlimit() — Control maximum resource consumption

Standards

Standards / Extensions	C or C++	Dependencies
XPG4.2 Single UNIX Specification, Version 3	both	

Format

```
#define _XOPEN_SOURCE_EXTENDED 1
#include <sys/resource.h>

int setrlimit(int resource, const struct rlimit *rlp);
```

General description

The setrlimit() function sets resource limits for the calling process. A resource limit is a pair of values; one specifying the current (soft) limit, the other a maximum (hard) limit.

The soft limit may be modified to any value that is less than or equal to the hard limit. For certain *resource* values, (RLIMIT_CPU, RLIMIT_NOFILE, RLIMIT_AS), the soft limit cannot be set lower than the existing usage.

The hard limit may be lowered to any value that is greater than or equal to the soft limit. The hard limit can be raised only by a process which has superuser authority. Both the soft limit and hard limit can be changed by a single call to setrlimit().

The value RLIM_INFINITY defined in <sys/resource.h>, is considered to be larger than any other limit value. If a call to getrlimit() returns RLIM_INFINITY for a resource, it means the implementation does not enforce limits on that resource. Specifying RLIM_INFINITY as any resource limit values on a successful call to setrlimit() inhibits enforcement of that resource limit.

The *resource* argument specifies which resource to set the hard and/or soft limits for, and may be one of the following values:

RLIMIT_CORE
The maximum size of a dump of memory (in bytes) allowed for the process. A value of 0 (zero) prevents file creation. Dump file creation will stop at this limit. .

RLIMIT_CPU
The maximum amount of CPU time (in seconds) allowed for the process. If the limit is exceeded, a SIGXCPU signal is sent to the process and the process is granted a small CPU time extension to allow for signal generation and delivery. If the extension is used up, the process is terminated with a SIGKILL signal. An attempt to set the CPU limit lower than that already used will result in an EINVAL errno.

RLIMIT_DATA
The maximum size of the break value for the process, in bytes. In this implementation, this resource always has a hard and soft limit value of RLIM_INFINITY. A call to setrlimit() to set this resource to any value other than RLIM_INFINITY will fail with an errno of EINVAL.

RLIMIT_FSIZE
: The maximum file size (in bytes) allowed for the process. A value of 0 (zero) prevents file creation. If the size is exceeded, a SIGXFSZ signal is sent to the process. If the process is blocking, catching, or ignoring SIGXFSZ, continued attempts to increase the size of a file beyond the limit will fail with an errno of EFBIG.

RLIMIT_MEMLIMIT
: The maximum amount of usable storage above the 2 gigabyte bar (in 1 megabyte segments) that can be allocated. Any attempt to extend the usable amount of virtual storage above the 2 gigabyte bar fails.

RLIMIT_NOFILE
: The maximum number of open file descriptors allowed for the process. This number is one greater than the maximum value that may be assigned to a newly created descriptor. (That is, it is one-based.) Any function that attempts to create a new file descriptor beyond the limit will fail with an EMFILE errno. An attempt to set the open file descriptors limit lower than that already used will result in an EINVAL errno.

  Restrictions: This value may not exceed 524288.

RLIMIT_STACK
: The maximum size of the stack for a process, in bytes. Note that in z/OS UNIX services, the stack is a per-thread resource. In this implementation, this resource always has a hard and soft limit value of `RLIM_INFINITY`. A call to setrlimit() to set this resource to any value other than `RLIM_INFINITY` will fail with an errno of EINVAL.

RLIMIT_AS
: The maximum address space size for the process, in bytes. If the limit is exceeded, malloc() and mmap() functions will fail with an errno of ENOMEM. Also, automatic stack growth will fail.

The *rlp* argument points to a `rlimit` structure. This structure contains the following members:

rlim_cur
: The current (soft) limit

rlim_max
: The maximum (hard) limit

Refer to the <sys/resource.h> header for more detail.

The resource limit values are propagated across exec and fork.

Special behavior for z/OS UNIX Services: An exception exists for exec processing in conjunction with daemon support. If a daemon process invokes exec and it had previously invoked setuid() before exec, the `RLIMIT_CPU`, `RLIMIT_AS`, `RLIMIT_CORE`, `RLIMIT_FSIZE`, and `RLIMIT_NOFILE` limit values are set based on the limit values specified in the kernel parmlib member BPXPRMxx.

For processes which are not the only process within an address space, the `RLIMIT_CPU` and `RLIMIT_AS` limits are shared with all the processes within the address space. For `RLIMIT_CPU`, when the soft limit is exceeded, action will be taken on the first process within the address space. If the action is termination, all processes within the address space will be terminated.

In addition to the RLIMIT_CORE limit values, the dump file defaults are set by SYSMDUMP defaults. Refer to *z/OS MVS Initialization and Tuning Reference* for more information on setting up SYSMDUMP defaults using the IEADMR00 parmlib member.

Dumps of memory are taken in 4160 byte increments. Therefore, RLIMIT_CORE values affect the size of memory dumps in 4160 byte increments. For example, if the RLIMIT_CORE soft limit value is 4000, the dump will contain no data. If the RLIMIT_CORE soft limit value is 8000, the maximum size of a memory dump is 4160 bytes.

When setting RLIMIT_NOFILE, the hard limit cannot exceed the system defined limit of 524288.

Large file support for z/OS UNIX files: Large z/OS UNIX files are supported automatically for AMODE 64 C/C++ applications. AMODE 31 C/C++ applications must be compiled with the option LANGLVL(LONGLONG) and define the _LARGE_FILES feature test macro before any headers are included to enable this function to operate on z/OS UNIX files that are larger than 2 GB in size. File size and offset fields are enlarged to 63 bits in width. Therefore, any other function operating on the file is required to define the _LARGE_FILES feature test macro as well.

Returned value

If successful, setrlimit() returns 0.

If unsuccessful, setrlimit() returns -1 and sets errno to one of the following values:

Error Code
Description

EINVAL
An invalid *resource* was specified, or the soft limit to set exceeds the hard limit to set, the soft limit to set is below the current usage, or the resource does not allow any value other than RLIM_INFINITY.

EPERM
The limit specified to setrlimit() would have raised the maximum limit value, and the calling process does not have appropriate privileges.

Related information

- “stropts.h” on page 72
- “sys/resource.h” on page 74
- “brk() — Change space allocation” on page 217
- “fork() — Create a new process” on page 571
- “getdtablesize() — Get the file descriptor table size” on page 702
- “getrlimit() — Get current or maximum resource consumption” on page 765
- “malloc() — Reserve storage block” on page 1031
- “open() — Open a file” on page 1152
- “rexec() — Execute commands one at a time on a remote host” on page 1447
- “sigaltstack() — Set or get signal alternate stack context” on page 1630
- “sysconf() — Determine system configuration options” on page 1798
- “ulimit() — Get or set process file size limits” on page 1942

setservent() — Open the network services information data set

Standards

Standards / Extensions	C or C++	Dependencies
XPG4.2 Single UNIX Specification, Version 3	both	

Format

X/Open:

```
#define _XOPEN_SOURCE_EXTENDED 1
#include <netdb.h>

void setservent(int stayopen);
```

Berkeley sockets:

```
#define _OE_SOCKETS
#include <netdb.h>

void setservent(int stayopen);
```

General description

The setservent() function opens and rewinds the */etc/services* or the *tcpip*.ETC.SERVICES data set. For more information on */etc/services* or the *tcpip*.ETC.SERVICES data set, see *z/OS Communications Server: IP Configuration Guide*. If the *stayopen* flag is nonzero, the *tcpip*.ETC.SERVICES data set remains open after each call.

Special behavior for C++: To use this function with C++, you must use the _XOPEN_SOURCE_EXTENDED 1 feature test macro.

Returned value

setservent() returns no values.

Related information

- “netdb.h” on page 49
- “endservent() — Close network services information data sets” on page 424
- “getservbyname() — Get a server entry by name” on page 770
- “getservent() — Get the next service entry” on page 773

setsid() — Create session, set process group ID

Standards

Standards / Extensions	C or C++	Dependencies
POSIX.1 XPG4 XPG4.2 Single UNIX Specification, Version 3	both	

Format

```
#define _POSIX_SOURCE
#include <unistd.h>

pid_t setsid(void);
```

General description

Creates a new session with the calling process as its session leader. The caller becomes the process group leader of a new process group. The calling process must not be a process group leader already. The caller does not have a controlling terminal.

The process group ID (PGID) of the new process group is equal to the process ID (PID) of the caller. The caller starts as the only process in the new process group and in the new session.

Returned value

If successful, setsid() returns the value of the caller's new PGID.

If unsuccessful, setsid() returns -1 and sets errno to one of the following values:

Error Code
Description

EPERM
One of the following error conditions exists:
- The caller is already a process group leader.
- The caller's PID matches the PGID of some other process.

Example

CELEBS10

```
/* CELEBS10

   This example creates a new session.

 */
#define _POSIX_SOURCE
#include <unistd.h>
#include <sys/types.h>
#include <stdio.h>

main() {
  pid_t pid;
  int p[2];
  char c='?';

  if (pipe(p) != 0)
    perror("pipe() error");
  else
    if ((pid = fork()) == 0) {
      printf("child's process group id is %d\n", (int) getpgrp());
      write(p[1], &c, 1);
      setsid();
      printf("child's process group id is now %d\n", (int) getpgrp());
      exit(0);
    }
    else {
      printf("parent's process group id is %d\n", (int) getpgrp());
      read(p[0], &c, 1);
      sleep(5);
    }
}
```

Output

```
child's process group id is 262152
child's process group id is now 262150
parent's process group id is 262152
```

Related information

- "sys/types.h" on page 75
- "unistd.h" on page 82
- "exec functions" on page 436
- "_exit() — End a process and bypass the cleanup" on page 445
- "fork() — Create a new process" on page 571
- "getpid() — Get the process ID" on page 752
- "kill() — Send a signal to a process" on page 931
- "setpgid() — Set process group ID for job control" on page 1565
- "sigaction() — Examine or change a signal action" on page 1611

setsockopt() — Set options associated with a socket

Standards

Standards / Extensions	C or C++	Dependencies
XPG4.2 Single UNIX Specification, Version 3	both	

Format

X/Open:

```
#define _XOPEN_SOURCE_EXTENDED 1
#include <sys/socket.h>

int setsockopt(int socket, int level, int option_name,
               const void *option_value, socklen_t option_length);
```

Berkeley sockets:

```
#define _OE_SOCKETS
#include <sys/types.h>
#include <sys/socket.h>

int setsockopt(int socket, int level, int option_name,
               char *option_value, int *option_length);
```

IPv6: To include support for IPv6 socket options, add the following code:

```
#define _OPEN_SYS_SOCK_IPV6 1
#include <netinet/in.h>
```

ip_mreq structure: To include the ip_mreq structure in your program, add the following code:

```
#define _XOPEN_SOURCE  500
#include <netinet/in.h>
```

or

```
#define _OPEN_SYS_SOCK_EXT3
#include <netinet/in.h>
```

group_req structure: To include the group_req structure in your program, add the following code:

```
#define _OPEN_SYS_SOCK_EXT3
#include <netinet/in.h>
```

group_source_req structure: To include the group_source_req structure in your program, add the following code:

```
#define _OPEN_SYS_SOCK_EXT3
#include <netinet/in.h>
```

ipv6_mreq structure: To include the ipv6_mreq structure in your program, add the following code:

```
#define _OPEN_SYS_SOCK_IPV6
#include <netinet/in.h>
```

icmp6_filter structure: To include the icmp6_filter structure in your program, add the following code:

```
#define _OPEN_SYS_SOCK_IPV6
#include <netinet/icmp6.h>
```

General description

The setsockopt() function sets options associated with a socket. Options can exist at multiple protocol levels.

Parameter
Description

socket The socket descriptor.

level The level for which the option is being set.

option_name
The name of a specified socket option.

option_value
The pointer to option data.

option_length
The length of the option data.

When manipulating socket options, you must specify the level at which the option resides and the name of the option. To manipulate options at the socket level, the *level* parameter must be set to SOL_SOCKET as defined in sys/socket.h. To manipulate options at the IPv4 or IPv6 level, the *level* parameter must be set to IPPROTO_IP as defined in sys/socket.h or IPPROTO_IPV6 as defined in netinet/in.h. To manipulate options at any other level, such as the TCP level, supply the appropriate protocol number for the protocol controlling the option. The getprotobyname() call can be used to return the protocol number for a named protocol.

The *option_value* and *option_length* parameters are used to pass data used by the particular set command. The *option_value* parameter points to a buffer containing the data needed by the set command. The *option_value* parameter is optional and can be set to the NULL pointer, if data is not needed by the command. The *option_length* parameter must be set to the size of the data pointed to by *option_value*.

All of the socket-level options except SO_LINGER, SO_RCVTIMEO and SO_SNDTIMEO, expect *option_value* to point to an integer and *option_length* to be set to the size of an integer. When the integer is nonzero, the option is enabled. When it is zero, the option is disabled. The SO_LINGER option expects *option_value* to point to a **linger** structure, as defined in **sys/socket.h**. This structure is defined in the following example:

```
struct  linger
{
        int     l_onoff;               /* option on/off */
        int     l_linger;              /* linger time */
};
```

The *l_onoff* field is set to 0 if the SO_LINGER option is begin disabled. A nonzero value enables the option. The *l_linger* field specifies the amount of time to linger on close. The units of *l_linger* are seconds.

The following options are recognized at the IPv4 level:

Option Description

IP_ADD_MEMBERSHIP

(RAW and UDP) This option is used to join a multicast group on a specific interface (an interface has to be specified with this option). Only applications that want to receive multicast datagrams need to join multicast groups. Applications that only transmit will not need to do so.

The multicast IP address and the interface IP address will be passed in the following structure available in netinet/in.h:

```
struct ip_mreq
{
   struct in_addr imr_multiaddr; /* IP multicast addr of group */
   struct in_addr imr_interface; /* local IP addr of interface */
};
```

If INADDR_ANY is specified on the interface address of the mreq structure passed a default interface will be chosen as follows:

- If the group address specified in the mreq structure was specified on a GATEWAY statement use that interface.
- If 224.0.0.0 was specified on GATEWAY statement use that interface.
- If DEFAULTNET was specified and is multicast capable use that interface.

IP_ADD_SOURCE_MEMBERSHIP

(RAW and UDP) This option is used to join a source-specific multicast group specified by the ip_mreq_source structure. The ip_mreq_source structure is defined in netinet/in.h.

IP_BLOCK_SOURCE

(RAW and UDP) This option is used to block from a given source to a given multicast group (e.g., if the user "mutes" that source). The source multicast group is specified by the ip_mreq_source structure which is defined in netinet/in.h.

IP_DROP_MEMBERSHIP

(RAW and UDP) This option is used to leave a multicast group.

The multicast IP address and the interface IP address will be passed in the following structure available in netinet/in.h:

```
struct ip_mreq
{
   struct in_addr imr_multiaddr; /* IP multicast addr of group */
   struct in_addr imr_interface; /* local IP addr of interface */
};
```

If INADDR_ANY is specified on the interface address of the mreq structure passed the system will drop the first group that matches the group (class D) address without regard to the interface.

IP_DROP_SOURCE_MEMBERSHIP
(RAW and UDP) This option is used to leave a source-specific multicast group specified by the ip_mreq_source structure. The ip_mreq_source structure is defined in netinet/in.h.

IP_MULTICAST_IF
(RAW and UDP) Sets the interface for sending outbound multicast datagrams from this socket application. Multicast datagrams will be transmitted only on one interface at a time. An IP address is passed in an in_addr structure.

If INADDR_ANY is specified for the interface address passed a default interface will be chosen as follows:

- If 224.0.0.0 was specified on GATEWAY statement use that interface.
- If DEFAULTNET was specified and is multicast capable use that interface.

IP_MULTICAST_LOOP
(RAW and UDP) Enables or disables loopback of outgoing multicast datagrams. Default is enable. When it is enabled, multicast applications that have joined the outgoing multicast group can receive a copy of the multicast datagrams destined for that address/port pair. The loopback indicator is passed as an u_char. The value 0 is specified to disable loopback. The value 1 is specified to enable loopback.

IP_MULTICAST_TTL
(RAW and UDP) Sets the IP time-to-live of outgoing multicast datagrams. Default value is 1 (that is, multicast only to the local subnet). The TTL value is passed as an u_char.

IP_RECVPKINFO
(RAW and UDP) Enables or disables returning the destination IP address of an incoming packet and the interface over which the packet was received as IP_PKTINFO ancillary data on recvmsg() function calls. The option value is specified as an int. A nonzero value enables the option; 0 disables the option.

This option is protected by the _OPEN_SYS_SOCK_EXT4 feature test macro.

IP_UNBLOCK_SOURCE
(RAW and UDP) This option is used to undo the operation performed with the IP_BLOCK_SOURCE option (e.g., if the user "mutes" that source). The source group is specified by the ip_mreq_source structure which is defined in netinet/in.h.

MCAST_BLOCK_SOURCE
(RAW and UDP) This option is used to block data from a given source to a given group (e.g., if the user "mutes" that source). The source is specified by the group_source_req structure which is defined in netinet/in.h.

MCAST_JOIN_GROUP
(RAW and UDP) This option is used to join an any-source group. The group is specified by the group_req structure. The group_req structure is defined in netinet/in.h.

MCAST_JOIN_SOURCE_GROUP
(RAW and UDP) This option is used to join a source-specific group. The source is specified by the group_source_req structure which is defined in netinet/in.h.

MCAST_LEAVE_GROUP
(RAW and UDP) This option is used to to leave an any-source group. The group is specified by the group_req structure. The group_req structure is defined in netinet/in.h.

MCAST_LEAVE_SOURCE_GROUP
(RAW and UDP) This option is used to leave a source-specific group. The source is specified by the group_source_req structure which is defined in netinet/in.h.

MCAST_UNBLOCK_SOURCE
(RAW and UDP) This option is used to undo the operation performed with the MCAST_BLOCK_SOURCE option (e.g., if the user then "unmutes" the source). The source is specified by the group_source_req structure which is defined in netinet/in.h.

The following options are recognized at IPv6 level:

Option Description

IPV6_ADDR_PREFERENCES
(TCP and UDP) Used to set the source address selection preference flags for a given socket. The socket option value (optval) is a 32-bit unsigned integer argument. The argument consists of a number of flags where each flag indicates a source address selection preference. These flags indicate the application's preferences for a source address, but will be ignored by the TCP stack if an IP address with the preferred address attributes is not available. For example, a preference flag of IPV6_PREFER_SRC_TMP tells the stack that the application would prefer to use a temporary IPv6 source address rather than a public source address. You can combine multiple flags with logical OR to express multiple preferences as long as the flags are not contradictory. The constants for the flag bit values are defined in netinet/in.h.

IPV6_CHECKSUM
(RAW) For a RAW (non-ICMPv6) socket, this option instructs the kernel to compute and store a checksum for output and verifies the received checksum on input. This prevents applications from having to perform source address selection on the packets sent. This option specifies an integer value into the user data where the checksum is located. This option can be disabled by specifying an option value of -1.

IPV6_DONTFRAG
(RAW and UDP) This option turns off the automatic inserting of a fragment header in the packet for UDP and raw sockets.

IPV6_DSTOPTS
(RAW and UDP) The application can remove any sticky destination options header by calling setsockopt() for this option with a zero option length.

IPV6_HOPOPTS
(RAW and UDP) The application can remove any sticky hop-by-hop options header by calling setsockopt() for this option with a zero option length.

IPV6_JOIN_GROUP
(RAW and UDP) Controls the receipt of multicast packets by joining the multicast group specified by the ipv6_mreq structure that is passed. The ipv6_mreq structure is defined in netinet/in.h.

IPV6_LEAVE_GROUP
(RAW and UDP) Controls the receipt of multicast packets by leaving the multicast group specified by the ipv6_mreq structure that is passed. The ipv6_mreq structure is defined in netinet/in.h.

IPV6_MULTICAST_HOPS
(RAW and UDP) Sets the hop limit for outgoing multicast packets. The hop limit value is passed as an int.

IPV6_MULTICAST_IF
(RAW and UDP) Sets the interface for outgoing multicast packets. An interface index is used to specify the interface. It is passed as an u_int.

IPV6_MULTICAST_LOOP
(RAW and UDP) If a multicast datagram is sent to a group to which the sending host itself belongs (on the outgoing interface), a copy of the datagram is looped back by the IP layer for local delivery if this option is set to one. If this option is set to zero, a copy is not looped back. Other option values return an errno of EINVAL. The default is one (loopback). The option value is passed as an int.

IPV6_NEXTHOP
(RAW and UDP) Specifies the next hop for the datagram as a socket address structure.

IPV6_RECVDSTOPTS
(RAW and UDP) To receive destination options header this option must be enabled.

IPV6_RECVHOPLIMIT
(RAW, TCP, and UDP) When this option is enabled, the received hop limit from an incoming packet will be returned as IPV6_HOPLIMIT ancillary data on recvmsg() function calls. The option value is specified as an int. A nonzero value enables the option, zero disables the option.

IPV6_RECVHOPOPTS
(RAW and UDP) To receive a hop-by-hop options header this option must be enabled.

IPV6_RECVPATHMTU
(RAW and UDP) Enables the receipt of IPV6_PATHMTU ancillary data on recvmsg() function calls.

IPV6_RECVPKTINFO
(RAW and UDP) Enables or disables returning the destination IP address of an incoming packet and the interface over which the packet was received as IPV6_PKTINFO ancillary data on recvmsg() function calls. The option value is specified as an int. A nonzero value enables the option; 0 disables the option.

IPV6_RECVRTHDR
(RAW and UDP) To receive a routing header this option must be enabled.

IPV6_RECVTCLASS
(RAW, TCP, and UDP) To receive the traffic class this option must be enabled.

IPV6_RTHDR
(RAW and UDP) The application can remove any sticky routing header by calling setsockopt() for this option with a zero option length.

IPV6_RTHDRDSTOPTS
(RAW and UDP) The application can remove any sticky destination options header by calling setsockopt() for this option with a zero option length.

IPV6_TCLASS
(RAW, TCP, and UDP) To specify the traffic class value this option must be enabled.

IPV6_UNICAST_HOPS
(RAW and UDP) Used to control hop limit in outgoing unicast IPv6 packets. The hop limit value is passed as an int.

IPV6_USE_MIN_MTU
(RAW, TCP, and UDP) Indicates whether the IP layer will use the minimu MTU size (1280) for sending packets, bypassing path MTU discovery. The option value is passed as an int. A value of -1 causes the default values for unicast (disabled) and multicast (enabled) destinations to be used. A value of 0 disables this option for unicast and multicast destinations. A value of 1 enables this option for unicast and multicast destinations and the minimum MTU size will be used.

IPV6_V6ONLY
(RAW. TCP, and UDP) Used to determine whether a socket is restricted to IPv6 communications only. The default setting is off. The option value is passed as an int. A non-zero value means the option is enabled (socket can only be used for IPv6 communications). 0 means the option is disabled.

Note: To use these options, you must use the feature test macro #define _OPEN_SYS_SOCK_IPV6.

The following options are recognized at the ICMPv6 level:

Option Description

ICMP6_FILTER
(RAW) Used to filter ICMPv6 messages. The option value is passed as an icmp6_filter structure. The icmp6_filter structure is defined in netinet/icmp6.h.

The following options are recognized at the socket level:

Option Description

SO_BROADCAST
Toggles the ability to broadcast messages. If enabled, this option allows the application program to send broadcast messages over *socket*, if the interface specified in the destination supports broadcasting of packets. This option has no meaning for stream sockets.

SO_DEBUG
Turns on recording of debugging information. This option enables or disables debugging in the underlying protocol modules. This option takes an int value.

SO_KEEPALIVE
Toggles the TCP keep-alive mechanism for a stream socket. When activated, the keep-alive mechanism periodically sends a packet on an otherwise idle connection. If the remote TCP does not respond to the packet or to retransmissions of the packet, the connection is terminated with the error ETIMEDOUT.

SO_LINGER
Lingers on close if data is present. When this option is enabled and there is unsent data present when close() is called, the calling application program is blocked during the close() call, until the data is transmitted or the connection has timed out. If this option is disabled, the TCP/IP address space waits to try to send the data. Although the data transfer is usually successful, it cannot be guaranteed, because the TCP/IP address space waits only a finite amount of time trying to send the data. The close() call returns without blocking the caller. This option has meaning only for stream sockets.

SO_OOBINLINE
Toggles the reception of out-of-band data. When this option is enabled, it causes out-of-band data to be placed in the normal data input queue as it is received, making it available to recv(), recvfrom(), and recvmsg() without having to specify the MSG_OOB flag in those calls. When this option is disabled, it causes out-of-band data to be placed in the priority data input queue as it is received, making it available to recv(), recvfrom(), and recvmsg() only by specifying the MSG_OOB flag in those calls. This option has meaning only for stream sockets.

_SO_PROPAGATEUSERID
Toggles propagating a user ID (UID) over an AF_UNIX stream socket. When enabled, user (UID) information is extracted from the system when the connect() function is invoked. Then, when the accept() function is invoked, the accepter assumes the identity of the connecter until the accepted socket is closed.

SO_RCVBUF
Sets receive buffer size. This option takes an int value. The value cannot be set after the socket is connected.

SO_RCVTIMEO

Sets the timeout specifying the maximum amount of time an input function waits until it completes.

It accepts a *timeval* structure with the number of seconds and microseconds specifying the limit on how long to wait for an input operation to complete. The *timeval* structure contains the number of seconds and microseconds specified as fullword binary numbers. The seconds can be a value in the range from 0 to 2,678,400 (equal to 31 days), and the microseconds can be a value in the range from 0 to 1,000,000 (equal to 1 second). While the *timeval* structure can be specified using mircosecond granularity, the internal TCP/IP timers used to implement this function have a granularity of approximately 100 milliseconds.

If a receive operation has blocked for this much time without receiving additional data, it returns with a partial count or errno set to EWOULDBLOCK if no data is received. The default for this option is zero, which indicates that a receive operation does not time out.

SO_REUSEADDR

Toggles local address reuse. When enabled, this option allows local addresses that are already in use to be bound. This alters the normal algorithm used in the bind() call.

The system checks at connect time to ensure that the local address and port do not have the same foreign address and port. The error EADDRINUSE is returned if the association already exists.

After the SO_REUSEADDR option is active, the following situation is supported:

A server can bind() the same port multiple times as long as every invocation uses a different local IP address and the wildcard address INADDR_ANY is used only one time per port.

SO_SNDBUF

Sets send buffer size. This option takes an int value. The value cannot be set after the socket is connected.

SO_SNDTIMEO

Sets the timeout value specifying the amount of time that an output function blocks due to flow control preventing data from being sent.

It accepts a *timeval* structure with the number of seconds and microseconds specifying the limit on how long to wait for an input operation to complete. The *timeval* structure contains the number of seconds and microseconds specified as fullword binary numbers. The seconds can be a value in the range from 0 to 2,678,400 (equal to 31 days), and the microseconds can be a value in the range from 0 to 1,000,000 (equal to 1 second). While the timeval structure can be specified using mircosecond granularity, the internal TCP/IP timers used to implement this function have a granularity of approximately 100 milliseconds.

If a send operation has blocked for this time, it returns with a partial count or with errno set to EWOULDBLOCK if no data is sent. The default for this option is zero, which indicates that a send operation does not time out.

SO_SECINFO

Toggles receiving security information. When enabled on an AF_UNIX UDP socket, the recvmsg() function will return security information about the sender of each datagram as ancillary data. This information contains the sender's user ID, UID, GID, and job name and it is mapped by the secsinfo structure in sys/socket.h.

Note: To use these options, you must use the Feature Test Macro #define _OPEN_SYS_SOCK_IPV6.

Special behavior for C++: To use this function with C++, you must use the _XOPEN_SOURCE_EXTENDED 1 feature test macro.

Returned value

If successful, setsockopt() returns 0.

If unsuccessful, setsockopt() returns -1 and sets errno to one of the following values:

Error Code
Description

EADDRNOTAVAIL
The ipi6_addr is not available for use on the ipi6_ifindex interface or the tuple consisting of socket, interface, and multicast group values does not exist..

EBADF
The *socket* parameter is not a valid socket descriptor.

EDOM
The send or receive timeout values exceed 31 days.

EFAULT
Using *option_value* and *option_length* parameters would result in an attempt to access storage outside the caller's address space.

EHOSTUNREACH
No route to the destination exists over the interface specified by ifi6_ifindex.

EINVAL
The specified option is invalid at the specified socket level or the socket has been shut down.

ENETDOWN
The interface specified by ipi6_ifindex is not enabled for IPv6 use.

ENOBUFS
Insufficient system resources are available to complete the call or a maximum of 64 source filters can be specified per multicast group, interface pair.

ENOPROTOOPT
The *option_name* parameter is unrecognized, or the *level* parameter is not SOL_SOCKET.

ENOSYS
The function is not implemented. You attempted to use a function that is not yet available.

ENOTSOCK
The descriptor is for a file, not for a socket.

ENXIO
The interface specified by ipi6_ifindex does not exist.

Example

The following are examples of the setsockopt() call. See "getsockopt() — Get the options associated with a socket" on page 777 for examples of how the getsockopt() options set are queried.

```
int rc;
int s;
int option_value;
struct linger l;
int setsockopt(int s, int level, int option_name,
char *option_value,
     int option_len);
```

```
⋮
/* I want out of band data in the normal input queue */
option_value = 1;
rc = setsockopt(s, SOL_SOCKET, SO_OOBINLINE,
(char *) &option_value, sizeof(int));

⋮
/* I want to linger on close */
l.l_onoff  = 1;
l.l_linger = 100;
rc = setsockopt(s, SOL_SOCKET, SO_LINGER,
(char *) &l, sizeof(l));
```

Related information

- “netinet/in.h” on page 53
- “sys/socket.h” on page 74
- “sys/types.h” on page 75
- “fcntl() — Control open file descriptors” on page 474
- “getprotobyname() — Get a protocol entry by name” on page 756
- “getsockopt() — Get the options associated with a socket” on page 777
- “ioctl() — Control device” on page 869
- “recvmsg() — Receive messages on a socket and store in an array of message headers” on page 1405
- “sendmsg() — Send messages on a socket” on page 1500
- “socket() — Create a socket” on page 1688
- For more information about IPv4 socket options, see *z/OS Communications Server: IP Sockets Application Programming Interface Guide and Reference*.
- For more information about IPv6 socket options, see *z/OS Communications Server: IPv6 Network and Appl Design Guide*.

setsourcefilter() — Set source filter

Standards

Standards / Extensions	C or C++	Dependencies
RFC3678	both	z/OS V1.9

Format

```
#define _XOPEN_SYS_SOCK_EXT3
#include <netinet/in.h>

int setsourcefilter(int s, uint32_t interface, struct sockaddr *group,
 socklen_t grouplen, uint32_t fmode, uint32_t numsrc,
 struct sockaddr_storage *slist);
```

General description

This function allow applications to set and replace the current multicast filtering state for a tuple consisting of socket, interface, and multicast group values.

A multicast filter is described by a filter mode, which is MCAST_INCLUDE or MCAST_EXCLUDE, and a list of source addresses which are filtered.

This function is protocol-independent. It can be on either AF_INET or AF_INET6 sockets of the type SOCK_DGRAM or SOCK_RAW.

If the function is unable to obtain the required storage, control will not return to the caller. Instead the application will terminate due to an out of memory condition (if the reserve stack is available and the caller is not XPLINK), or it will terminate with an abend indicating that storage could not be obtained.

Argument
Description

s Identifies the socket.

interface
Holds the local the index of the interface.

group Points to either a sockaddr_in structure for IPv4 or a sockaddr_in6 structure for IPv6 that holds the IP multicast address of the group.

grouplen
Gives the length of the sockaddr_in or sockaddr_in6 structure.

fmode Identifies the filter mode. The value of this field will be either MCAST_INCLUDE or MCAST_EXCLUDE, which are likewise defined in <netinet/in.h>.

numsrc
Holds the number of source addresses in the slist array.

slist Points to an array of IP addresses of sources to include or exclude depending on the filter mode.

Returned value

If successful, the function returns 0. Otherwise, it returns -1 and sets errno to one of the following values.

errno Description

EBADF
s is not a valid socket descriptor.

EAFNOSUPPORT
The address family of the input sockaddr is not AF_INET or AF_INET6.

EPROTOTYPE
The socket s is not of type SOCK_DGRAM or SOCK_RAW.

EINVAL
Interface or group is not a valid address, or the socket s has already requested multicast setsockopt options (refer to z/OS Communications Server: IP Sockets Application Programming Interface Guide and Reference for details.) Or if the group address family is AF_INET and grouplen is not at least size of sockaddr_in or if the group address family is AF_INET6 and grouplen is not at least size of sockaddr_in6 or if grouplen is not at least size of sockaddr_in.

ENOBUFS
The number of the source addresses exceeds the allowed limit.

Related information

- “netinet/in.h” on page 53
- “getsourcefilter() — Get source filter” on page 785

setstate() — Change generator for random()

Standards

Standards / Extensions	C or C++	Dependencies
XPG4.2 Single UNIX Specification, Version 3	both	

Format

```
#define _XOPEN_SOURCE_EXTENDED 1
#include <stdlib.h>

char *setstate(const char *state);
```

General description

The setstate() function allows switching between state arrays used by the random() function once a state has been initialized. The array defined by the *state* argument is used for further random-number generation by the calling thread until initstate() is called or setstate() is called again. The setstate() function returns a pointer to the previous state array.

After initialization, a state array can be restarted at a different point by calling setstate() with the desired state, followed by srandom() with the desired seed.

Returned value

If successful, setstate() returns a pointer to the previous state array.

If unsuccessful, setstate() returns a NULL pointer. The function will fail and write a message to standard error if it detects that the state information has been damaged.

Related information

- “stdlib.h” on page 70
- “drand48() — Pseudo-random number generator” on page 402
- “initstate() — Initialize generator for random()” on page 867
- “rand() — Generate random number” on page 1374
- “rand_r() — Pseudo-random number generator” on page 1375
- “random() — A better random-number generator” on page 1375
- “srandom() — Use seed to initialize generator for random()” on page 1718

set_terminate() — Register a function for terminate()

Standards

Standards / Extensions	C or C++	Dependencies
ANSI/ISO C++	C++ only	

Format

```
#include <exception>

terminate_handler set_terminate(terminate_handler ph) throw();
```

General description

The set_terminate() function is part of the z/OS XL C++ error handling mechanism. The argument supplied to set_terminate() is of type terminate_handler as defined in the header <exception> (that is, a pointer to a function with a void return type and no arguments). The function specified will be called by the terminate() function.

Note that the function registered for terminate() must terminate execution of the program without returning to its caller(). If set_terminate() has not yet been called, then terminate() calls a system-defined default terminate handler, which calls abort().

In a multithreaded environment, the terminate function created by the issuance of a set_terminate() call applies to all threads in the (POSIX) process. If a thread throws an exception which is not caught by that thread of execution, then terminate() is called. The default terminate() action calls abort() which by default cause a SIGABRT signal. If there is no signal handler, then SIGABRT terminates the process. You can override this with a thread-level termination by supplying a function which invokes pthread_exit() as a terminate function. This terminates the thread but not the process.

Returned value

set_terminate() returns the address of the previous terminate_handler.

Refer to *z/OS XL C/C++ Language Reference* for more information about z/OS XL C++ exception handling, including the set_terminate() function.

Related information

- "exception" on page 26
- "terminate() — Terminate after failures in C++ error handling" on page 1868

_SET_THLIIPADDR() — Set the client's IP address

Standards

Standards / Extensions	C or C++	Dependencies
z/OS UNIX	both	

Format

```
#include <sys/__ussos.h>

int _SET_THLIIPADDR(ln, ipaddr);
```

General description

The _SET_THLIIPADDR() macro provides a way for daemons to set a client's IP address.

_SET_THLIIPADDR() takes the following arguments:

ln The length of the IP address as specified by *ipaddr*. The IP address length can be between 1 and 16 inclusive. The argument is specified as an unsigned int.

ipaddr Pointer to the IP address.

Usage notes

The intent of the _SET_THLIIPADDR() macro is to provide a way for daemons to set the IP address of a client for Security Authorization Facility (SAF) exits when performing security related functions.

Restrictions

Results are unpredictable if _SET_THLIIPADDR() is issued outside of the z/OS UNIX environment.

Returned value

If the client's IP address is set, _SET_THLIIPADDR() returns nonzero.

_SET_THLIIPADDR() returns 0 and does not set the IP address of the client when:
- The base level of z/OS UNIX is not OS/390 R5.
- The setting of the IP address is not supported.
- The length of the IP address is less then 1 or greater than 16.

Related information
- “sys/__ussos.h” on page 77

setuid() — Set the effective user ID

Standards

Standards / Extensions	C or C++	Dependencies
POSIX.1 XPG4 XPG4.2 Single UNIX Specification, Version 3	both	

Format

```
#define _POSIX_SOURCE
#include <unistd.h>

int setuid(uid_t uid);
```

General description

Sets the real, effective, or saved set user IDs (UIDs) for the current process to *uid*.

If *uid* is the same as the real UID or the saved set-user-ID of the process, setuid() always succeeds and sets the effective UID. the real user ID and saved set-user-ID will remain unchanged.

The setuid() function will not affect the supplementary group list in any way.

If *uid* is not the same as the real UID of the process, setuid() succeeds only if the process has appropriate privileges. If the process has such privileges, setuid() sets the real group ID (UID), effective UID, and saved set UID to *uid*.

The setuid() function is not supported from an address space running multiple processes, since it would cause all processes in the address space to have their security environment changed unexpectedly.

setuid() can be used by daemon processes to change the identity of a process in order for the process to be used to run work on behalf of a user. In z/OS UNIX, changing the identify of a process is done by changing the real and effective UIDs and the auxiliary groups. In order to change the identity of the process on MVS completely, it is necessary to also change the MVS security environment. The identity change will only occur if the EUID value is specified, changing just the real UID will have no effect on the MVS environment.

The setuid() function invokes MVS SAF services to change the MVS identity of the address space. The MVS identity that is used is determined as follows:

- If an MVS user ID is already known by the kernel from a previous call to a kernel function (for example, getpwnam()) and the UID for this user ID matches the UID specified on the setuid() call, then this user ID is used.
- For nonzero target UIDs, if there is no saved user ID or the UID for the saved user ID does not match the UID requested on the setuid() call, the setuid() function queries the security database (for example, using getpwnam()) to retrieve a user ID. The retrieved user ID is then used.
- If the target UID is 0 and a user ID is not known, the setuid() function always sets the MVS user ID to BPXROOT or the value specified on the SUPERUSER parm in sysparms. BPXROOT is set up during system initialization as a superuser with a UID=0. The BPXROOT user ID is not defined to the BPX.DAEMON FACILITY class profile. This special processing is necessary to prevent a superuser from gaining daemon authority.

  Note: When running under UID=0, some servers will issue setuid(0) in order to test whether they are running UID=0. The problem with this is that the setuid function will change the userid to BPXROOT which will likely cause the daemon to fail on subsequent function requests.
- When changing from a nonzero UID to a UID=0, the MVS user ID is not changed. When using the *su* shell command without specifying user name to become a superuser, the new shell retains the original MVS user ID.
- A nondaemon superuser that attempts to set a user ID to a daemon superuser UID fails with an EPERM.

When the MVS identity is changed, the daemon must make a call to initgroups() to set the auxiliary list of groups to the list of groups for the new user ID.

If the setuid() function is issued from multiple tasks within one address space, use synchronization to ensure that the setuid() functions are not performed concurrently. The execution of setuid() function concurrently within one address space can yield unpredictable results.

Returned value

If successful, setuid() returns 0.

If unsuccessful, setuid() returns -1 and sets errno to one of the following values:

Error Code
Description

EAGAIN
The process is currently not able to change UIDs.

EINVAL
The value of *uid* is incorrect.

EPERM
The process does not have appropriate privileges to set the UID to *uid*.

Example

CELEBS11

```
/* CELEBS11

   This example changes the effective UID.

 */
#define _POSIX_SOURCE
#include <sys/types.h>
#include <stdio.h>
#include <unistd.h>

main() {
  printf("prior to setuid(), uid=%d, effective uid=%d\n",
         (int) getuid(), (int) geteuid());
  if (setuid(25) != 0)
    perror("setuid() error");
  else
    printf("after setuid(),    uid=%d, effective uid=%d\n",
           (int) getuid(), (int) geteuid());
}
```

Output

```
before setuid(), uid=0, effective uid=0
after setuid(),    uid=25, effective uid=25
```

Related information

- "sys/types.h" on page 75
- "unistd.h" on page 82
- "exec functions" on page 436
- "geteuid() — Get the effective user ID" on page 707
- "getuid() — Get the real user ID" on page 791
- "seteuid() — Set the effective user ID" on page 1531
- "setgid() — Set the group ID" on page 1537
- "setreuid() — Set real and effective user IDs" on page 1571

set_unexpected() — Register a function for unexpected()

Standards

Standards / Extensions	C or C++	Dependencies
ANSI/ISO C++	C++ only	

Format

```
#include <exception>

unexpected_handler set_unexpected(unexpected_handler ph) throw();
```

General description

The set_unexpected() function is part of the z/OS XL C++ error handling mechanism. The argument supplied to set_unexpected() is of type unexpected_handler as defined in the header <exception> (that is, a pointer to a function with a void return type and no arguments). The function specified will be called by the unexpected() function.

Note that the function registered for unexpected() must not return to its caller. It may terminate execution by:

- Throwing an object of a type listed in the exception specification (or an object of any type if the unexpected handler is called directly by the program).
- Throwing an object of type bad_exception.
- Calling terminate(), abort(), or exit(int).

If set_unexpected() has not yet been called, then unexpected() calls terminate().

In a multithreaded environment, the unexpected() function created by the issuance of a set_unexpected() call applies to all threads in the (POSIX) process.

Returned value

set_unexpected() returns the address of the previous unexpected_handler.

Refer to *z/OS XL C/C++ Language Reference* for more information about z/OS XL C++ exception handling, including the set_unexpected() function.

Related information

- "exception" on page 26
- "unexpected() — Handle exception not listed in exception specification" on page 1957

setutxent() — Reset to start of utmpx database

Standards

Standards / Extensions	C or C++	Dependencies
XPG4.2 Single UNIX Specification, Version 3	both	

Format

```
#define _XOPEN_SOURCE_EXTENDED 1
#include <utmpx.h>

void setutxent(void);
```

General description

The setutxent() function resets the input to the beginning of the utmpx database opened by previous calls to getutxid(), getutxent(), getutxline(), or pututxline() calls from the current thread. This should be done before each getutxid() and getutxline() search for a new entry if it is desired that the entire database be examined.

Because the setutxent() function processes thread-specific data the setutxent() function can be used safely from a multithreaded application. If multiple threads in the same process open the database, then each thread opens the database with a different file descriptor. The thread's database file descriptor is closed when the calling thread terminates or the endutxent() function is called by the calling thread.

Returned value

setutxent() returns no values.

Related information

- "utmpx.h" on page 84
- "endutxent() — Close the utmpx database" on page 425
- "getutxent() — Read next entry in utmpx database" on page 793
- "getutxline() — Search by line utmpx database" on page 796
- "getutxid() — Search by ID utmpx database" on page 795
- "pututxline() — Write entry to utmpx database" on page 1354
- "__utmpxname() — Change the utmpx database name" on page 1971

setvbuf() — Control buffering

Standards

Standards / Extensions	C or C++	Dependencies
ISO C POSIX.1 XPG4 XPG4.2 C99 Single UNIX Specification, Version 3	both	

Format

```
#include <stdio.h>

int setvbuf(FILE * __restrict__stream, char * __restrict__buf, int type, size_t size);
```

General description

Controls the buffering strategy and buffer size for a specified stream. The *stream* pointer must refer to an open file, and setvbuf() must be the first operation on the file.

To provide an ASCII input/output format for applications using this function, define the feature test macro __LIBASCII as described in topic 2.1.

The location pointed to by *buf* designates an area that you provide that the z/OS XL C/C++ Runtime Library may choose to use as a buffer for the stream. A *buf* value of NULL indicates that no such area is supplied and that the z/OS XL C/C++ Runtime Library is to assume responsibility for managing its own buffers for the stream. If you supply a buffer, it must exist until the stream is closed.

If *type* is `_IOFBF` or `_IOLBF`, *size* is the size of the supplied buffer. If *buf* is NULL, the C library will take *size* as the suggested size for its own buffer. If *type* is `_IONBF`, both *buf* and *size* are ignored. Unbuffered I/O is allowed for memory files and hierarchical file system (HFS) files. However, it is not permitted for Hiperspace memory files. If the size of the supplied buffer for hiperspace memory files is greater than 4k, only the first 4k of the buffer will be used.

Value **Meaning**

_IONBF No buffer is used.

_IOFBF Full buffering is used for input and output. Use *buf* as the buffer and *size* as the size of the buffer.

_IOLBF Line buffering is used for text stream I/O and terminal I/O. The buffer is flushed when a newline character is used (text stream), when the buffer is full, or when input is requested (terminal). The value for size must be greater than 0.

The value for *size* must be greater than 0.

Note: If you use setvbuf() or setbuf() to define your own buffer for a stream, you must ensure that either the buffer is available after program termination, or the stream is closed or flushed, before you call exit(). This can be done by defining the array with file scope or by dynamically allocating the storage for the array using malloc().

For example, if the buffer is declared within the scope of a function block, the *stream* must be closed before the function is terminated. This prevents the storage allocated to the buffer from being freed.

Returned value

If successful, even if it chooses not to use your buffer. setvbuf() returns 0.

If an invalid value was specified in the parameter list, or if the request cannot be performed, setvbuf() returns nonzero.

Example

```
/* This example sets up a buffer of buf for stream1 and specifies that
   input from stream2 is to be unbuffered.
 */
#include <stdio.h>
#define  BUF_SIZE  1024

char buf[BUF_SIZE];

int main(void)
{
   FILE *stream1, *stream2;

   stream1 = fopen("myfile1.dat", "r");
   stream2 = fopen("myfile2.dat", "r");

   /* stream1 uses a user-assigned buffer of BUF_SIZE bytes */
   if (setvbuf(stream1, buf, _IOFBF, sizeof(buf)) != 0)
      printf("Incorrect type or size of buffer 1");

   /* stream2 is unbuffered                              */
   if (setvbuf(stream2, NULL, _IONBF, 0) != 0)
      printf("Incorrect type or size of buffer 2");
⋮
}
```

Related information

- One of the section s about I/O Operations in *z/OS XL C/C++ Programming Guide*.
- "stdio.h" on page 68
- "fclose() — Close file" on page 473
- "fflush() — Write buffer to file" on page 523
- "fopen() — Open a file" on page 565
- "setbuf() — Control buffering" on page 1522

shmat() — Shared memory attach operation

Standards

Standards / Extensions	C or C++	Dependencies
XPG4 XPG4.2 Single UNIX Specification, Version 3	both	

Format

```
#define _XOPEN_SOURCE
#include <sys/shm.h>

void  *shmat(int shmid, const void *shmaddr, int shmflg);
```

General description

The shmat() function attaches the shared memory segment associated with the shared memory identifier, *shmid*, to the address space of the calling process. The segment is attached at the address specified by one of the following criteria:

- If *shmaddr* is a NULL pointer, the segment is attached at the first available address as selected by the system.
- If *shmaddr* is not a NULL pointer, and the flag, **SHM_RND** was specified, the segment is attached at the address given by (*shmaddr*-((prtdiff_t)*shmaddr* %SHMLBA)) where % is the 'C' language remainder operator.
- If *shmaddr* is not a NULL pointer, and the flag, **SHM_RND** was not specified, the segment is attached at the address given by *shmaddr*.
- The segment is attached for reading if the flag, **SHM_RDONLY**, is specified with *shmflg* and the calling process has read permission. If the flag is not set and the process has both read and write permission, the segment is attached for reading and writing.

  The first attach of newly created __IPC_MEGA segment, as well as subsequent attaches, will have write access to the segment, regardless of the SHM_RDONLY option.
- All attaches to an **__IPC_MEGA** shared memory segment have the same Write or Read access authority. If a segment is enabled for writes then all attaches have the ability to read and write to the segment. If the segment is disabled for writes, then all attaches have the ability to read from the segment and cannot write to the segment

  The first attach of newly created **__IPC_MEGA** segment, as well as subsequent attaches, will have write access to the segment, regardless of the SHM_RDONLY option. Write/Read access can be changed by the shmctl() function, Shared Memory Control Operations.

An __IPC_MEGA shared memory segment is attached as follows:

- If shmaddr is zero and __IPC_MEGA segment, then the segment will be attached at the first available address selected by the system on a segment boundary.
- If shmaddr is not zero and SHM_RND is specified and __IPC_MEGA segment, the segment address will be truncated to the segment boundary (last 20 bits zero).
- If shmaddr is not zero and SHM_RND is not specified and __IPC_MEGA segment, the segment address must be a megabyte multiple (segment boundary).

Returned value

If successful, shmat() increments the value of `shm_nattach` in the data structure associated with the shared memory ID of the attached shared memory segment and returns the segment's starting address.

If unsuccessful, shmat() returns -1 and sets errno to one of the following values:

Error Code
Description

EACCES
Operation permission is denied to the calling process.

EINVAL
The value of *shmid* is not a valid shared memory identifier; the *shmaddr* is not a NULL pointer and the value of (*shmaddr*-((ptrdiff_t)*shmaddr* %SHMLBA)) is an illegal address for attaching shared memory segments; or the *shmaddr* is not a NULL pointer, **SHM_RND** was specified, and the value of *shmaddr* is an illegal address for attaching shared memory segments.

The shared memory address, *shmaddr, is not zero, is not on a megabyte boundary, and SHM_RND was not specified.

EMFILE
The number of shared memory segments attached to the calling process would exceed the system-imposed limit.

ENOMEM
The available data space is not large enough to accommodate the shared memory segment.

Related information

- "sys/shm.h" on page 74
- "exit() — End program" on page 443
- "_exit() — End a process and bypass the cleanup" on page 445
- "fork() — Create a new process" on page 571
- "rexec() — Execute commands one at a time on a remote host" on page 1447
- "shmctl() — Shared memory control operations" on page 1600
- "shmdt() — Shared memory detach operation" on page 1601
- "shmget() — Get a shared memory segment" on page 1602

shmctl() — Shared memory control operations

Standards

Standards / Extensions	C or C++	Dependencies
XPG4 XPG4.2 Single UNIX Specification, Version 3	both	

Format

```
#define _XOPEN_SOURCE
#include <sys/shm.h>

int shmctl(int shmid, int cmd, struct shmid_ds *buf);
```

General description

The shmctl() function provides a variety of shared memory control operations on the shared memory segment identified by the argument, *shmid*.

The argument *cmd* specifies the shared memory control operation and may be any of the following values:

IPC_STAT
This command obtains status information for the shared memory segment specified by the shared memory identifier, *shmid*. It places the current value of each member of the `shmid_ds` data structure associated with *shmid* into the structure pointed to by *buf*. The contents of this structure is defined in <sys/shm.h>. This command requires read permission.

IPC_SET
Set the value of the following members of the `shmid_ds` data structure associated with *shmid* to the corresponding value in the structure pointed to by *buf*:

shm_perm.uid
shm_perm.gid
shm_perm.mode (only the low-order 9 bits)

This command can only be executed by a process that has an effective user ID equal to either that of a process with appropriate privileges or to the value of `shm_perm.cuid` or `shm_perm.uid` in the `shmid_ds` data structure associated with *shmid*.

Using the IPC_SET function to change the IPC_MODE for an __IPC_MEGA shared memory segment will have an immediate effect on all attaches to the target segment. That is, the read and write access of all current attachers is immediately affected by the permissions specified in the new IPC_MODE. To determine how the new mode affects access, you must consider the effect of all three parts of the mode field (the owner permissions, group permissions and other permissions). If all three read and all three write permissions in the new mode are set off, then the access for all attachors is changed to read. If any of the three read permission bits is set on but the corresponding write permission bit is off, then the access for all attachors is changed to read. Otherwise, the access of all attachors is changed to write.

IPC_RMID
Remove the shared memory identified specified by *shmid* from the system and destroy the shared memory segment and `shmid_ds` data structure associated with *shmid*. This command can only be executed by a process that has an effective user ID equal to either that of a process with appropriate privileges or to the value of `shm_perm.cuid` or `shm_perm.uid` in the `shmid_ds` data structure associated with *shmid*. The remove will be completed asynchronous to the return from the shmctl() function, when the last attachment is detached. When **IPC_RMID** is processed, no further attaches will be allowed.

The IPC_STAT option of shmctl() returns a structure named shmid_ds (mapped in shm.h). shmid_ds contains status information for the requested shared memory segment.

As part of its own dump priority support, the USS Kernel will be adding a new field to shmid_ds. This field will contain the dump priority of the requested shared memory segment. The new field will be added to the shmid_ds structure in shm.h.

Returned value

If successful, shmctl() returns 0.

If unsuccessful, shmctl() returns -1 and sets errno to one of the following values:

Error Code
Description

EACCES
The argument *cmd* is equal to **IPC_STAT** but the calling process does not have read permission.

EINVAL
The value of *shmid* is not a valid shared memory identifier or the value of *cmd* is not a valid command.

EPERM
The argument *cmd* is equal to either **IPC_RMID** or **IPC_SET** and the effective user ID of the calling process is not equal to that of a process with appropriate privileges and it is not equal to the value of `shm_perm.cuid` or `shm_perm.uid` in the data structure associated with *shmid*.

Related information

- “sys/shm.h” on page 74
- “sys/ipc.h” on page 73
- “shmat() — Shared memory attach operation” on page 1598
- “shmdt() — Shared memory detach operation”
- “shmget() — Get a shared memory segment” on page 1602

shmdt() — Shared memory detach operation

Standards

Standards / Extensions	C or C++	Dependencies
XPG4 XPG4.2 Single UNIX Specification, Version 3	both	

Format

```
#define _XOPEN_SOURCE
#include <sys/shm.h>

int shmdt(const void *shmaddr);
```

General description

The shmdt() function detaches from the calling process's address space the shared memory segment located at the address specified by the argument *shmaddr*.

Storage in the user address space for a segment with the __IPC_SHAREAS attribute is not cleaned up unless the segment is no longer attached to by other processes in the address space.

Returned value

If successful, shmdt() decrements the value of `shm_nattach` in the data structure associated with the shared memory ID of the attached shared memory segment and returns 0.

If unsuccessful, shmdt() returns -1 and sets errno to one of the following values:

Error Code
  Description

EINVAL
  The value of *shmaddr* is not the data segment start address of a shared memory segment.

Related information

- "sys/shm.h" on page 74
- "exit() — End program" on page 443
- "_exit() — End a process and bypass the cleanup" on page 445
- "fork() — Create a new process" on page 571
- "rexec() — Execute commands one at a time on a remote host" on page 1447
- "shmat() — Shared memory attach operation" on page 1598
- "shmctl() — Shared memory control operations" on page 1600
- "shmget() — Get a shared memory segment"

shmget() — Get a shared memory segment

Standards

Standards / Extensions	C or C++	Dependencies
XPG4 XPG4.2 Single UNIX Specification, Version 3	both	

Format

```
#define _XOPEN_SOURCE
#include <sys/shm.h>

int shmget(key_t key, size_t size, int shmflg);
```

General description

The shmget() function returns the shared memory identifier associated with *key*.

A shared memory identifier, associated data structure and shared memory segment of at least *size* bytes, see <sys/shm.h>, are created for *key* if one of the following is true:

1. Argument *key* has a value of **IPC_PRIVATE**
2. Argument *key* does not already have a shared memory identifier associated with it and the flag **IPC_CREAT** was specified

Specify __IPC_MEGA to request segment level sharing. The resulting shared memory segment will be allocated in units of segments instead of units of pages. The shared memory size parameter still reflects the number of bytes required but must be in megabyte multiples. A shared memory size parameter of 0 or one which is not a megabyte multiple will result in the request failing.

The first shmget to define the shared memory segment determines whether the segment has the __IPC_MEGA attribute or not. Subsequent shmgets, those that use existing shared memory segments, will use the __IPC_MEGA attribute defined by that segment. The __IPC_MEGA option will have no effect for these shmgets and will be ignored.

Specification of the __IPC_MEGA option for large segments will result in significant real storage savings and reduced ESQA usage, especially as the number of shares increases.

Valid values for the argument *shmflg* include any combination of the following constants defined in <sys/ipc.h> and <sys/modes.h>:

__IPC_SHAREAS
This flag enables the sharing of the same storage area from multiple processes in the same address space. When specified by an AMODE 31 application, this flag is only honored when __IPC_MEGA is also specified, otherwise it is ignored. When specified by an AMODE 64 application, this flag is honored for any type of shared memory segment that is obtained above the bar.

__IPC_BELOWBAR
Forces the memory object to be allocated from below the 2 gigabyte address range. This can be used to allow AMODE 64 applications to share objects with non-AMODE 64 applications. This option is mutually exclusive with the __IPC_GIGA option. If a 31-bit application specifies this option, then the request will be failed with EINVAL.

IPC_CREAT
Create a shared memory segment if the *key* specified does not already have an associated ID. **IPC_CREAT** is ignored when **IPC_PRIVATE** is specified.

IPC_EXCL
Causes the shmget() function to fail if the *key* specified has an associated ID. **IPC_EXCL** is ignored when **IPC_CREAT** is not specified or **IPC_PRIVATE** is specified.

__IPC_GIGA
Requests a shared memory segment with a size in gigabyte multiples. Use of this option requires that the size parameter be specified as a gigabyte

multiple. Failure to use a gigabyte multiple will result in a failure. [EINVAL] This option is mutually exclusive with the **__IPC_BELOWBAR** and **__IPC_MEGA** options.

__IPC_MEGA
Requests a shared memory segment with the size in megabyte multiples. Use of this option requires that the size parameter, size_t, be in a megabyte multiple. The __IPC_MEGA option is required to create the shared memory segment but the **__IPC_MEGA** option is not required to acquire access to a previously defined/created shared memory segment that has the __IPC_MEGA attribute. When specified by an AMODE 64 application, option __IPC_BELOWBAR is implied and megaroo sharing will be in effect. This option is mutually exclusive with the __IPC_GIGA option.

S_IRGRP
Permits read access when the effective group ID of the caller matches either `shm_perm.cgid` or `shm_perm.gid`.

S_IROTH
Permits other read access.

S_IRUSR
Permits read access when the effective user ID of the caller matches either `shm_perm.cuid` or `shm_perm.uid`.

S_IWGRP
Permits write access when the effective group ID of the caller matches either `shm_perm.cgid` or `shm_perm.gid`.

S_IWOTH
Permits other write access.

S_IWUSR
Permits write access when the effective user ID of the caller matches either `shm_perm.cuid` or `shm_perm.uid`.

When a shared memory segment associated with argument *key* already exists, setting **IPC_EXCL** and **IPC_CREAT** in argument *shmflg* will force shmget() to fail.

The following fields are initialized when a `shmid_ds` data structure is created:

- The fields `shm_perm.cuid` and `shm_perm.uid` are set equal to the effective user ID of the calling process
- The fields `shm_perm.cgid` and `sem_perm.gid` are set equal to the effective group ID of the calling process
- The low-order 9 bits of `shm_perm.mode` are set to the value in the low-order 9 bits of *shmflg*
- The field `shm_segsz` is set equal to the value of the argument *size*
- The field `shm_lpid`, `shm_nattach`, `shm_atime`, and `shm_dtime` are set equal to zero
- The value of `shm_ctime` is set equal to the current time

Usage notes

- Shared memory segments created with __IPC_MEGA will show this bit in S_MODE byte returned with *w_getipc*.

Special behavior for AMODE 64: Applications will not be allowed to change the address to which a shared memory segment allocated is attached, when it resides above the 2 gigabyte address range. The size parameter is rounded up to a megabyte multiple for AMODE 64 users.

Returned value

If successful, shmget() returns a nonnegative integer, namely a shared memory identifier.

If unsuccessful, shmget() returns -1 and sets errno to one of the following values:

Error Code
  Description

EACCES
  A shared memory identifier exists for the argument *key*, but operation permission as specified by the low-order 9 bits of *shmflg* could not be granted

EEXIST
  A shared memory identifier exists for the argument *key* and both **IPC_CREAT** and **IPC_EXCL** are specified in *shmflg*

EINVAL
  A shared memory identifier does not exist for the argument *key* specified and the value of argument *size* is less than the system-imposed minimum or greater than the system-imposed maximum.

  OR a shared memory identifier exists for the argument *key*, but the size of the segment associated with it is less that specified by argument *size*.

  OR __IPC_MEGA is specified and the segment size, size_t, is not in megabyte multiples.

ENOENT
  A shared memory identifier does not exist for the argument, *key*, and **IPC_CREAT** is not specified.

ENOMEM
  A shared memory identifier and associated shared memory segment are to be created but the amount of available system storage was insufficient to fill the request.

ENOSPC
  A shared memory identifier is to be created but the system-imposed limit on the maximum number of allocated shared memory identifiers, system-wide, would be exceeded.

When *shmflg* equals 0, the following applies:

- If a shared memory identifier has already been created with *key* earlier, and the calling process of this shmget() has read and/or write permissions to it, then shmget() returns the associated shared memory identifier.
- If a shared memory identifier has already been created with *key* earlier, and the calling process of this shmget() does not have read and/or write permissions to it, then shmget() returns-1 and sets errno to EACCES.
- If a shared memory identifier has not been created with *key* earlier, then shmget() returns -1 and sets errno to ENOENT.

Related information

- “sys/ipc.h” on page 73
- “sys/shm.h” on page 74
- “sys/types.h” on page 75
- “ftok() — Generate an interprocess communication (IPC) key” on page 662
- “shmat() — Shared memory attach operation” on page 1598

- “shmctl() — Shared memory control operations” on page 1600
- “shmdt() — Shared memory detach operation” on page 1601

shutdown() — Shut down all or part of a duplex connection

Standards

Standards / Extensions	C or C++	Dependencies
XPG4.2 Single UNIX Specification, Version 3	both	

Format

X/Open:

```
#define _XOPEN_SOURCE_EXTENDED 1
#include <sys/socket.h>

int shutdown(int socket, int how);
```

Berkeley sockets:

```
#define _OE_SOCKETS
#include <sys/socket.h>

long shutdown(int *s, int how);
```

General description

The shutdown() function shuts down all or part of a duplex connection.

Parameter
Description

socket The socket descriptor.

how The condition of the shutdown. The values 0, 1, or 2 set the condition. *how* sets the condition for shutting down the connection to the socket indicated by *socket*.

how can have a value of:
- **SHUT_RD**, which ends communication from the socket indicated by *socket*.
- **SHUT_WR**, which ends communication to the socket indicated by *socket*.
- **SHUT_RDWR**, which ends communication both to and from the socket indicated by *socket*.

Note: You should issue a shutdown() call before you issue a close() call for a socket.

Special behavior for C++: To use this function with C++, you must use the _XOPEN_SOURCE_EXTENDED 1 feature test macro.

Returned value

If successful, shutdown() returns 0.

If unsuccessful, shutdown() returns -1 and sets errno to one of the following values:

Error Code
Description

EBADF
socket is not a valid socket descriptor.

EINVAL
The *how* parameter was not set to one of the valid values.

ENOBUFS
Insufficient system resources are available to complete the call.

ENOTCONN
The socket is not connected.

ENOTSOCK
The descriptor is for a file, not for a socket.

Related information

- "sys/socket.h" on page 74
- "accept() — Accept a new connection on a socket" on page 105
- "close() — Close a file" on page 288
- "connect() — Connect a socket" on page 308
- "socket() — Create a socket" on page 1688

__shutdown_registration() — Register OMVS shutdown options

Standards

Standards / Extensions	C or C++	Dependencies
z/OS UNIX	both	z/OS V1R3

Format

```
#define _OPEN_SYS
#include <signal.h>

int __shutdown_registration(int regtype, int regscope, int regoptions);
```

General description

The __shutdown_registration() function is used to register OMVS Shutdown characteristics for the process. The process can be registered as one of the following:

- a shutdown blocking process
- a permanent process
- a shutdown notification process

These types are mutually exclusive. A shutdown blocking process will prevent OMVS shutdown from proceeding until it either de-registers as a blocking process or ends. A permanent process will survive across an OMVS shutdown. Most OMVS process attributes will be checkpointed during the shutdown and restored during the Restart. A shutdown notification process will be informed when an OMVS shutdown is initiated. It neither blocks Shutdown nor survives shutdown. Blocking and permanent processes can also register to be informed when OMVS shutdown is initiated. For more information on OMVS shutdown see *z/OS UNIX System Services Planning, GA32-0884*.

Registration can be done for the invoking process only, or for all of the tasks in the job. The process can also modify the behavior of OMVS requests issued by permanent processes while OMVS shutdown and restart is in progress.

Checkpointed permanent process attributes include the following:
- process user and group identity
- process, session and process group identities
- process file mode creation mask
- zombie child processes
- signal registration, signal actions, signal mask data and pending signals
- current working directory
- open file and socket descriptors

Zombie child process ending status is checkpointed so that the permanent process can retrieve it after the restart.

If the current working directory path name cannot be resolved after restart, then the current working directory is set to a dummy root which will cause relative path name lookup to fail.

All of the checkpointed file descriptors will be marked invalid after restart, and any I/O requests other than close() will cause EIO errors.

Timer events are not checkpointed. Timer events which expire before the restart completes are lost. Timer events which have not expired after restart is complete will still be in effect.

Non-checkpointable permanent process attributes include the following:
- semaphores
- shared library programs
- __map() shared memory blocks
- message queues
- memory mapped files
- all other UNIX System Services resources

If any non-checkpointable resources are being used by a permanent process, the shutdown request will fail.

Registration is at the process level, not the thread level. For multithreaded applications, the SIGDANGER signal is sent to the process and not to any particular thread.

If a process is registered as a blocking process or a permanent process, the process must de-register before attempting to register with a different registration type. For example, a blocking process must de-register as a blocking process before attempting to register as a permanent process.

Registration remains in effect for the life of the process, or until the process de-registers. Registration remains in place across an exec() syscall because the new program image runs in the same process. Registration does not propagate to child processes as a result of fork() and spawn() syscalls.

regtype defines the type of registration. The possible values are listed below. These values are all mutually exclusive.

regtype
  Description

_SDR_BLOCKING
  The process will prevent OMVS Shutdown from proceeding for as long as it the process remains registered as a blocking process. If the process exits or de-registers as a blocking process then OMVS Shutdown can proceed.

_SDR_PERMANENT
  The process will not be terminated during OMVS Shutdown and Restart processing.

_SDR_NOBLOCKING
  The process will no longer block an OMVS Shutdown.

_SDR_NOPERMANENT
  The process will no longer be a permanent process.

_SDR_NOTIFY
  The process will be notified by SIGDANGER signal delivery once OMVS Shutdown is initiated.

_SDR_NONOTIFY
  The process will no longer be notified by SIGDANGER signal delivery if OMVS Shutdown is initiated.

The _SDR_BLOCKING and _SDR_PERMANENT registrations are restricted. The invoker must meet one of the following criteria in order for these two registration types to succeed:

- The calling address space is a system started task address space.
- The caller is running authorized (APF Authorized, System Key (0-7) or Supervisor State).
- The caller is a privileged UNIX process. It must either have a superuser identity or have read permission to BPX.SHUTDOWN.

regscope defines the registration scope. The possible values are listed below. The two values are mutually exclusive.

regscope
  Definition

_SDR_REGJOB
  All the processes in the Job are registered.

_SDR_REGPROCESS
  Only the calling process is registered.

regoptions defines various options for the registered process. The possible values are listed below. Multiple options may be specified by or'ing the values together. The default behavior for kernel calls issued by permanent processes while z/OS UNIX is not up is to fail the request with errno set to EMVSERR and the reason code (__errno2() value) set to JrKernelReady. Those kernel calls which which do not return a return code will end with an EC6 abend and reason code xxxx8039.

regoptions
  Definition

_SDR_NOOPTIONS
No options are requested. This request code is not valid for _SDR_NOTIFY registration.

_SDR_BLOCKSYSCALLS
Kernel calls issued from permanent processes while OMVS is not up will hang, and return to the caller once z/OS UNIX System Services is back up. This request is mutually exclusive with _SDR_ABENDSYSCALLS, and is valid only for permanent process registration.

_SDR_ABENDSYSCALLS
Kernel calls issued from permanent processes while OMVS is not up will ABEND. This request is mutually exclusive with _SDR_BLOCKSYSCALLS, and is valid only for permanent process registration.

_SDR_SENDSIGDANGER
Kernel sends SIGDANGER signal to the process when OMVS Shutdown is initiated. This option MUST be specified on _SDR_NOTIFY registration. This option may be specified for _SDR_BLOCKING and _SDR_PERMANENT registration. It may be combined with either _SDR_BLOCK_SYSCALLS or _SDR_ABENDSYSCALLS on _SDR_PERMANENT registration.

Returned value

If successful, __shutdown_registration() returns zero. the service completes without error, otherwise it returns

There are no documented errnos for this function.

If unsuccessful, __shutdown_registration() returns -1 and sets errno and __errno2() to indicate the cause of the failure. The _errno2() values are documented as reason codes in *z/OS UNIX System Services Messages and Codes*.

The values of errno are:

Error Code
Description

EINVAL
Failed for one of the following reasons:

- The callable service is rejected because the job step process must be registered before registering a lower process of the job step process.
- The request to register a blocking process or job, or a request to register a permanent process or job cannot be performed as a shutdown is currently in progress.
- The request to register a blocking process or job, or a request to register a permanent process or job cannot be performed as the job can not be de-registered while a lowerprocess is still registered.
- The request to deregister a blocking process or job, or a request to deregister a permanent process or job cannot be performed because the job or the current process is not registered.
- One of the parameters was invalid.

EPERM
Failed for one of the following reasons:

- Invoker does not have superuser or equivalent authority.

- Caller must be given read permission to BPX.SHUTDOWN facility class profile in order to use __shutdown_registration() successfully.

EMVSSAF2ERR
Internal Security product error. Hexadecimal Reason code value contains the two byte security product return code xx and reason code yy.

Example

```
/*
Register the process as a blocking process and request notification
of shutdown initiation by way of SIGDANGER signal.
*/
#define _OPEN_SYS
#include <signal.h>
...
if (-1 == (rc = __shutdown_registration(_SDR_ BLOCKING, _SDR_REGPROCESS,
         _SDR_SENDSIGDANGER)))
     printf("Error during __shutdown_registration errno=%d,
         errno2=0x%08x\n", errno, __errno2())

/*
Register the process as a permanent process and don't ask for
SIGDANGER signals.
*/
#define _OPEN_SYS
#include <sys>
if (-1 == (rc = __shutdown_registration(_SDR_PERMANENT, _SDR_REGPROCESS,
         _SDR_NOOPTIONS)))
     printf("Error during __shutdown_registration errno=%d,
         errno2=0x%08x\n", errno, __errno2())
```

Related information

- “signal.h” on page 63

sigaction() — Examine or change a signal action

Standards

Standards / Extensions	C or C++	Dependencies
POSIX.1 XPG4 XPG4.2 Single UNIX Specification, Version 3	both	POSIX(ON)

Format

```
#define _POSIX_SOURCE
#include <signal.h>

int sigaction(int sig, const struct sigaction *__restrict__ new,
              struct sigaction *__restrict__ old);
```

General description

Examines and changes the action associated with a specific signal.

`int` *sig* is the number of a recognized signal. sigaction() examines and sets the action to be associated with this signal. See Table 48 on page 1612 for the values of *sig*, as well as the signals supported by z/OS UNIX services. The *sig* argument must be one of the macros defined in the signal.h header file.

`const struct sigaction` **new* may be a NULL pointer. If so, sigaction() merely determines the action currently defined to handle *sig*. It does not change this action. If *new* is not NULL, it should point to a `sigaction` structure. The action specified in this structure becomes the new action associated with *sig*.

`struct sigaction` **old* points to a memory location where sigaction() can store a `sigaction` structure. sigaction() uses this memory location to store a `sigaction` structure describing the action currently associated with *sig*. *old* can also be a NULL pointer, in which case sigaction() does not store this information.

This function is supported only in a POSIX program.

Special behavior for C++:

- The behavior when mixing signal-handling with C++ exception handling is undefined. Also, the use of signal-handling with constructors and destructors is undefined.
- C++ and C language linkage conventions are incompatible, and therefore sigaction() cannot receive C++ function pointers. If you attempt to pass a C++ function pointer to sigaction(), the compiler will flag it as an error. Therefore, to use the sigaction() function in the C++ language, you must ensure that signal handler routines established have C linkage, by declaring them as `extern "C"`.

Signals: Table 48 lists signal values and their default action and meaning.

Table 48. Signal values and signals supported by z/OS UNIX services

Value	Default Action	Meaning
SIGABND	1	Abend.
SIGABRT	1	Abnormal termination (sent by abort()).
SIGALRM	1	A timeout signal (sent by alarm()).
SIGBUS	1	Bus error (available only when running on MVS 5.2 or higher).
SIGFPE	1	Arithmetic exceptions that are not masked, for example, overflow, division by zero, and incorrect operation.
SIGHUP	1	A controlling terminal is suspended, or the controlling process ended.
SIGILL	1	Detection of an incorrect function image.
SIGINT	1	Interactive attention.
SIGKILL	1	A termination signal that cannot be caught or ignored.
SIGPIPE	1	A write to a pipe that is not being read.
SIGPOLL	1	Pollable event occurred (available only when running on MVS 5.2 or higher).
SIGPROF	1	Profiling timer expired (available only when running on MVS 5.2 or higher).
SIGQUIT	1	A quit signal for a terminal.
SIGSEGV	1	Incorrect access to memory.
SIGSYS	1	Bad system call issued (available only when running on MVS 5.2 or higher).
SIGTERM	1	Termination request sent to the program.
SIGTRAP	1	Internal for use by dbx or ptrace.

Table 48. Signal values and signals supported by z/OS UNIX services (continued)

Value	Default Action	Meaning
SIGURG	2	High bandwidth data is available at a socket (available only when running on MVS 5.2 or higher).
SIGUSR1	1	Intended for use by user applications.
SIGUSR2	1	Intended for use by user applications.
SIGVTALRM	1	Virtual timer has expired (available only when running on MVS 5.2 or higher).
SIGXCPU	1	CPU time limit exceeded (available only when running on MVS 5.2 or higher). If a process runs out of CPU time and SIGXCPU is caught or ignored, a SIGKILL is generated.
SIGXFSZ	1	File size limit exceeded.
SIGCHLD	2	An ended or stopped child process (SIGCLD is an alias name for this signal).
SIGIO	2	Completion of input or output.
SIGIOERR	2	A serious I/O error was detected.
SIGWINCH	2	Window size has changed (available only when running on MVS 5.2 or higher).
SIGSTOP	3	A stop signal that cannot be caught or ignored.
SIGTSTP	3	A stop signal for a terminal.
SIGTTIN	3	A background process attempted to read from a controlling terminal.
SIGTTOU	3	A background process attempted to write to a controlling terminal.
SIGCONT	4	If stopped, continue.

The **Default Actions** in Table 48 on page 1612 are:

1 Normal termination of the process.

2 Ignore the signal.

3 Stop the process.

4 Continue the process if it is currently stopped. Otherwise, ignore the signal.

If the main program abends in a way that is not caught or handled by the operating system or application, z/OS UNIX terminates the running application with a KILL -9. If z/OS UNIX gets control in EOT or EOM and the terminating status has not been set, z/OS UNIX sets it to appear as if a KILL -9 occurred.

If a signal catcher for a SIGABND, SIGFPE, SIGILL or SIGSEGV signal runs as a result of a program check or an ABEND, and the signal catcher executes a RETURN statement, the process will be terminated.

sigaction structure: The `sigaction` structure is defined as follows:

```
struct sigaction {
     void      (*sa_handler)(int);
     sigset_t   sa_mask;
     int        sa_flags;
     void      (*sa_sigaction)(int, siginfo_t *, void *);
};
```

The following are members of the structure:

`void (*)(int) sa_handler`
A pointer to the function assigned to handle the signal. The value of this member can also be SIG_DFL (indicating the default action) or SIG_IGN (indicating that the signal is to be ignored).

Special behavior for XPG4.2: This member and `sa_sigaction` are mutually exclusive of each other. When the SA_SIGINFO flag is set in `sa_flags` then `sa_sigaction` is used. Otherwise, `sa_handler` is used.

`sigset_t sa_mask`
A signal set identifies a set of signals that are to be added to the signal mask of the calling process before the signal-handling function sa_handler or **sa_sigaction** (in XPG4.2) is invoked. For more on signal sets, see "sigemptyset() — Initialize a signal mask to exclude all signals" on page 1633. You cannot use this mechanism to block SIGKILL, SIGSTOP, or SIGTRACE. If sa_mask includes these signals, they will simply be ignored; sigaction() will not return an error.

sa_mask must be set by using one or more of the signal set manipulation functions: sigemptyset(), sigfillset(), sigaddset(), or sigdelset()

`int sa_flags`
A collection of flag bits that affect the behavior of signals. The following flag bits can be set in sa_flags:

_SA_IGNORE
This bit is output only and cannot be specified by the application. The handler value will be saved and returned on subsequent calls, but the signal is ignored.

SA_NOCLDSTOP
Tells the system not to issue a SIGCHLD signal when child processes stop. This is relevant only when the *sig* argument of sigaction() is SIGCHLD.

SA_NOCLDWAIT
Tells the system not to create 'zombie' processes when a child process dies. This is relevant only when the *sig* argument of sigaction() is SIGCHLD. If the calling process subsequently waits for its children, and the process has no unwaited for children that were transformed into zombie processes, it will block until all of its children terminate. The wait(), waitid(), or waitpid() will fail and set errno to **ECHILD**.

SA_NODEFER
Tells the system to bypass automatically blocking this signal when invoking a signal handler function.

_SA_OLD_STYLE
Tells the C runtime library to use ANSI signal delivery rules, instead of POSIX rules. It is supported for compatibility with applications that use signal() to set signal action. (See "signal() — Handle interrupts" on page 1643.) For a description of ANSI and POSIX.1 signal delivery rules, find "Handling Error Conditions and Signals" in *z/OS XL C/C++ Programming Guide*.

SA_ONSTACK
Tells the system to use the alternate signal stack (see "sigaltstack() — Set or get signal alternate stack context" on page 1630 or

"sigstack() — Set or get signal stack context" on page 1662) when invoking a signal handler function. If an alternate signal stack has not been declared, the signal handler function will be invoked with the current stack.

SA_RESETHAND

Tells the system to reset the signal's action to SIG_DFL and clear the SA_SIGINFO flag before invoking a signal handler function (Note: SIGILL and SIGTRAP cannot be automatically reset when delivered. However, no error will be generated should this situation exist). Otherwise, the disposition of the signal will not be modified on entry to the signal handler.

In addition, if this flag is set, sigaction() behaves as if the SA_NODEFER flag were also set.

SA_RESTART

Tells the system to restart certain library functions if they should be interrupted by a signal. The functions that this restartability applies to are all of those that are defined as interruptible by signals and set errno to **EINTR** (except pause(), sigpause(), and sigsuspend()).

Table 49 lists functions that are restartable if interrupted by a signal.

Table 49. Functions that are restartable if interrupted by a signal

accept()
catclose()
catgets()
chmod()
chown()
close()
closedir()
connect()
creat()
dup2()
endgrent()
fchmod()
fchown()
fclose()
fcntl()
fflush()
fgetc()
fgetwc()
fopen()
fputc()
fputwc()
freopen()
fseek()
fstatvfs()
fsync()
ftruncate()
getgrgid()
getgrnam()
getmsg()
getpass()
getpwnam()
getpwuid()
ioctl()
lchown()
lockf()
mkfifo()
msgrcv()
msgxrcv()
msgsnd()
open()
poll()
putmsg()
read()
readv()
recv()
recvfrom()
recvmsg()
select()
semop()
send()
sendmsg()
sendto()
statvfs()
tcdrain()
tcflow()
tcflush()
tcgetattr()
tcgetpgrp()
tcsendbreak()
tcsetattr()
tcsetpgrp()
tmpfile()
umount()
wait()
waitid()
waitpid()
write()

SA_SIGINFO

Tells the system to use the signal action specified by `sa_sigaction` instead of `sa_handler`.

When this flag is off and the action is to catch the signal, the signal handler function specified by `sa_handler` is invoked as:

void function(int *signo*);

Where *signo* is the only argument to the signal handler and it specifies the type of signal that has caused the signal handler function to be invoked.

When this flag is on and the action is to catch the signal, the signal handler function specified by `sa_sigaction` is invoked as:

```
void   function(int signo, siginfo_t *info, void *context);
```

Where two additional arguments are passed to the signal handler function. If the second argument is not a NULL pointer, it will point to an object of type `siginfo_t` which provides additional information about the source of the signal. A `siginfo_t` object is a structure contains the following members:

si_signo
Contains the system-generated signal number

si_errno
Contains the implementation-specific error information (it is not used on this implementation)

si_code
Contains a code identifying the cause of the signal (refer to the <signal.h> include file for a list of these codes and for their meanings, see Table 50 on page 1645).

If `si_signo` contains SIGPOLL then `si_code` can be set to SI_ASYNCIO. Otherwise, if the value of si_code is less than or equal to zero then the signal was generated by another process and the `si_pid` and `si_uid` members respectively indicate the process ID and the real user ID of the sender of this signal.

If the value of `si_code` is less than or equal to zero, then the signal was generated by another process and the `si_pid` and `si_uid` members respectively indicate the process ID and the real user ID of the sender of this signal.

si_pid If the value of `si_code` is less than or equal to zero, then this member will indicate the process ID of the sender of this signal. Otherwise, this member is meaningless.

si_uid If the value of `si_code` is less than or equal to zero, then this member will indicate the real user ID of the sender of this signal. Otherwise, this member is meaningless.

si_value
If `si_code` is SI_ASYNCIO the si_value contains the application specified value. Otherwise, the contents of si_value are undefined

The third argument will point to an object of type `ucontext_t` (refer to the <ucontext.h> include file for a description of the contents of this object).

Note: The remaining flag bits are reserved for system use. There is no guarantee that the integer value of "int sa_flags" will be the same upon return from sigaction(). However, all flag bits defined above will remain unchanged.

`void (*)(int, siginfo_t *, void *) sa_sigaction`

A pointer to the function assigned to handle the signal, or SIG_DFL, or SIG_IGN. This function will be invoked passing three parameters. The first is of type 'int' that contains the signal type for which this function is being invoked. The second is of type 'pointer to siginfo_t' where the siginfo_t contain additional information about the source of the signal. The third is of type 'pointer to void' but will actually point to a ucontext_t containing the context information at the time of the signal interrupt.

Notes:

1. The user must cast SIG_IGN or SIG_DFL to match the sa_sigaction definition. (indicating that the signal is to be ignored).
2. **Special behavior for XPG4.2:** This member and `sa_handler` are mutually exclusive of each other. When the SA_SIGINFO flag is set in `sa_flags` then `sa_sigaction` is used. Otherwise, `sa_handler` is used.

When a signal handler installed by sigaction(), with the _SA_OLD_STYLE flag set off, catches a signal, the system calculates a new signal mask by taking the union of the current signal mask, the signals specified by sa_mask, and the signal that was just caught (if the SA_NODEFER flag is not set). This new mask stays in effect until the signal handler returns, or sigprocmask(), sigsuspend(), siglongjmp(), sighold(), sigpause(), or sigrelse() is called. When the signal handler ends, the original signal mask is restored.

After an action has been specified for a particular signal, using sigaction() or signal(), it remains installed until it is explicitly changed with another call to sigaction(), signal(), one of the exec functions, bsd_signal(), sigignore(), sigset(), or until the SA_RESETHAND flag causes it to be reset to SIG_DFL.

After an action has been specified for a particular signal, using sigaction() with the _SA_OLD_STYLE flag not set, it remains installed until it is explicitly changed with another call to sigaction(), signal(), or one of the exec functions.

After an action has been specified for a particular signal, using sigaction() with the _SA_OLD_STYLE flag set or using signal(), it remains installed until it is explicitly changed with another call to sigaction(), signal(), or one of the exec functions, or a signal catcher is driven, where it will be reset to SIG_DFL.

Successful setting of signal action to SIG_IGN for a signal that is pending causes the pending signal to be discarded, whether or not it is blocked. This provides the ability to discard signals that are found to be blocked and pending by sigpending().

Special behavior for XPG4.2:

- If a process sets the action of the SIGCHLD signal to SIG_IGN, child processes of the calling process will not be transformed into 'zombie' processes when they terminate. If the calling process subsequently waits for its children, and the process has no unwaited for children that were transformed into 'zombie' processes, it will block until all of its children terminate. The wait(), waitid(), or waitpid() function will fail and set errno to **ECHILD**.
- If the **SA_SIGINFO** flag is set, the signal-catching function specified by `sa_sigaction` is invoked as:

  void *function*(int *signo*, siginfo_t **info*, void **context*);

Where *function* is the specified signal-catching function, *signo* is the signal number of the signal being delivered, *info* points to an object of type `siginfo_t` associated with the signal being delivered, and *context* points to an object of type `ucontext_t`.

Considerations for asynchronous signal-catching functions: Some of the functions have been restricted to be serially reusable with respect to asynchronous signals. That is, the library will not allow an asynchronous signal to interrupt the execution of one of these functions until it has completed.

This restriction needs to be taken into consideration when a signal-catching function is invoked asynchronously because it causes the behavior of some of the library functions to become unpredictable.

Thus, when you are producing a strictly compliant POSIX C or X/Open application, only the following functions should be assumed to be reentrant with respect to asynchronous signals. Use only these functions in your signal-catching functions:

access()
alarm()
cfgetispeed()
cfgetospeed()
cfsetispeed()
cfsetospeed()
chdir()
chmod()
chown()
close()
creat()
dup()
dup2()
execle()
execve()
_exit()
fcntl()
fork()
fstat()
getegid()
geteuid()
getgid()
getgroups()
getpgrp()
getpid()
getppid()
getuid()
kill()
link()
lseek()
mkdir()
mkfifo()
open()
pathconf()
pause()
pipe()
pthread_cond_broadcast()
pthread_cond_signal()
pthread_mutex_trylock()
read()
rename()
rmdir()
setgid()
setpgid()
setsid()
setuid()
sigaction()
sigaddset()
sigdelset()
sigemptyset()
sigfillset()
sigismember()
sigpending()
sigprocmask()
sigsuspend()
sleep()
stat()
sysconf()
tcdrain()
tcflow()
tcflush()
tcgetattr()
tcgetpgrp()
tcsendbreak()
tcsetattr()
tcsetpgrp()
time()
times()
umask()
uname()
unlink()
utime()
wait()
waitpid()
write()

Special behavior for XPG4.2: Adds the following functions to the list of functions above that may be used in signal-catching functions in strictly compliant X/Open applications:

- fpathconf()
- raise()
- signal()

The macro versions of getc() and putc() are not reentrant, even though the library versions of these functions are.

For nonportable POSIX applications, most of the library functions can be used in a signal-catching function. However, do not use the following functions:
- getenv()
- getgrent()
- getgrgid()
- getgrnam()
- getpwent()
- getpwnam()
- getpwuid()
- ttyname()

Special behavior for XPLINK-compiled C++: Restrictions concerning setjmp.h and ucontext.h:

1. All XPLINK programs compiled with the V2R10 or later C compilers that are to run with Language Environment V2R10 or later libraries and use the **jmp_buf**, **sigjmp_buf** or **ucontext_t** types must not be compiled with C headers from Language Environment V2R9 or earlier.
2. Non-XPLINK functions compiled with any level of Language Environment headers must not define **jmp_buf**, **sigjmp_buf** or **ucontext_t** data items and pass them to XPLINK functions that call getcontext(), longjmp(), _longjmp(), setjmp(), _setjmp(), setcontext(), sigsetjmp(), or swapcontext() with these passed-in data items.
3. When __XPLINK__ is defined, the Language Environment V2R10 and later headers define a larger **jmp_buf**, **sigjmp_buf** or **ucontext_t** area that is required by setjmp(), getcontext(), and related functions when they are called from an XPLINK routine. If __XPLINK__ is not defined, the Language Environment V2R10 and later headers define a shorter **jmp_buf**, **sigjmp_buf** or **ucontext_t** area. The Language Environment headers before V2R10 also define the shorter version of these data areas. If an XPLINK function calls setjmp(), getcontext() or similar functions with a short **jmp_buf**, **sigjmp_buf** or **ucontext_t** area, a storage overlay or program check may occur when the C library tries to store past the end of the passed-in (too short) data area.
4. The sigaction() function supersedes the signal() interface, and should be the preferred usage. In particular, sigaction() and signal() must not be used in the same process to control the same signal.

Usage notes

The use of the `SIGTHSTOP` and `SIGTHCONT` signal is not supported with this function.

Returned value

If successful, sigaction() returns 0.

If unsuccessful, no new signal handler is installed, sigaction() returns -1, and sets errno to one of the following values:

Error Code
Description

EINVAL
The value of *sig* is not a valid signal for one of the following reasons:
- The *sig* is not recognized.
- The process tried to ignore a signal that cannot be ignored.
- The process tried to catch a signal that cannot be caught.

The default action for SIGCHILD and SIGIO is for the signal to be ignored. A sigaction() to set the action to SIG_IGN for SIGIO will result in an error, with errno equal to EINVAL.

Example

CELEBS13

```
/* CELEBS13

   The first part of this example determines whether the SIGCHLD
   signal is currently being ignored.
   With a NULL pointer for the new argument, the current signal
   handler action is not changed.

 */
#define _POSIX_SOURCE
#define _XOPEN_SOURCE_EXTENDED 1
#include <stdio.h>
#include <signal.h>

void main(void) {
struct sigaction info;

if (sigaction(SIGCHLD,NULL,&info) != -1)
    if (info.sa_handler == SIG_IGN)
        printf("SIGCHLD being ignored.\n");
    else if (info.sa_handler == SIG_DFL)
            printf("SIGCHLD being defaulted.\n");
}
```

CELEBS14

```
/* CELEBS14

   This fragment initializes a sigaction structure to specify
   mysig as a signal handler and then sets the signal handler
   for SIGCHLD.
   Information on the previous signal handler for SIGCHLD is
   stored in info.

 */
#define _XOPEN_SOURCE_EXTENDED 1
#include <signal.h>
#include <stdio.h>
void mysig(int a) { printf("In mysig\n"); }

void main(void) {
   struct sigaction info, newhandler;

   if (sigaction(SIGCHLD,NULL,&info) != -1)
       if (info.sa_handler == SIG_IGN)
           printf("SIGCHLD being ignored.\n");
       else if(info.sa_handler == SIG_DFL)
           printf("SIGCHLD being defaulted.\n");

   newhandler.sa_handler = &mysig;
   sigemptyset(&(newhandler.sa_mask));
   newhandler.sa_flags = 0;
   if (sigaction(SIGCHLD,&newhandler,&info) != -1)
       printf("New handler set.\n"); }
```

Related information

- “signal.h” on page 63
- “alarm() — Set an alarm” on page 156
- “bsd_signal() — BSD version of signal()” on page 219

- “exec functions” on page 436
- “getcontext() — Get user context” on page 694
- “kill() — Send a signal to a process” on page 931
- “makecontext() — Modify user context” on page 1028
- “raise() — Raise signal” on page 1371
- “setcontext() — Restore user context” on page 1524
- “__sigactionset() — Examine or change signal actions”
- “sigaddset() — Add a signal to the signal mask” on page 1628
- “sigaltstack() — Set or get signal alternate stack context” on page 1630
- “sigdelset() — Delete a signal from the signal mask” on page 1632
- “sigemptyset() — Initialize a signal mask to exclude all signals” on page 1633
- “sigfillset() — Initialize a signal mask to include all signals” on page 1635
- “sigignore() — Set disposition to ignore a signal” on page 1637
- “siginterrupt() — Allow signals to interrupt functions” on page 1638
- “siglongjmp() — Restore the stack environment and signal mask” on page 1640
- “signal() — Handle interrupts” on page 1643
- “sigprocmask() — Examine or change a thread” on page 1651
- “sigset() — Change a signal action or a thread” on page 1656
- “sigstack() — Set or get signal stack context” on page 1662
- “sigsuspend() — Change mask and suspend the thread” on page 1663
- “swapcontext() — Save and restore user context” on page 1792
- “wait() — Wait for a child process to end” on page 1992
- “wait3() — Wait for child process to change state” on page 2000

__sigactionset() — Examine or change signal actions

Standards

Standards / Extensions	C or C++	Dependencies
z/OS UNIX	both	POSIX(ON) OS/390 V2R6

Format

```
#define _XOPEN_SOURCE_EXTENDED 1
#include <signal.h>

int __sigactionset(size_t newct, const __sigactionset_t new[],
                   size_t *oldct, __sigactionset_t old[],
                   int options);
```

General description

Examines and changes the actions associated with one or more signals. This function is equivalent to using sigaction() one or more times.

The parameters are:

size_t *newct*
: *newct* is the number of `__sigactionset_t` structures to be processed in the *new* array. The value of *newct* must be from 0 to 64. If this parameter is 0, the *new* parameter is ignored, and may be NULL. If the *newct* parameter is not 0, *new* must be an array containing at least *newct* `__sigactionset_t` structures.

const __sigactionset_t *new*[]
: *new* is an optional array of `__sigactionset_t` structures. When *newct* is 0, *new* may be NULL, and no signal actions will be changed.

When *newct* is not 0, the data in the *new* array of `__sigactionset_t` structures will cause the actions associated with one or more signals to be changed. The system will change the signal actions as if sigaction() were called multiple times. The first *newct* `__sigactionset_t` structures in the *new* array are processed in order, and may cause the actions for one or more signals to be set. For each array entry, the effect is the same as calling sigaction() once for each signal whose bit is on in the `__sa_signals` signal set. The fields `__sa_handler`, `__sa_mask`, `__sa_flags`, and `__sa_sigaction` correspond to the `sa_handler`, `sa_mask`, `sa_flags`, and `sa_sigaction` fields in the `sigaction` structure for sigaction().

If a signal appears in more than one `__sa_signals` signal set in the *new* array, the last action specified for that signal will be in effect when __sigactionset() returns. If all bits in all `__sa_signals` signal sets in the *new* parameter are off, no signal actions will be changed.

size_t*oldct*

oldct is both an input and output parameter. It points to a word containing the number of output entries allowed, used, or needed in the *old* array.

When __sigactionset() is called, **oldct* is the maximum number of `__sigactionset_t` structures in the *old* array that the system can fill in. The value of **oldct* must be from 0 to 64. If this parameter is 0, the *old* parameter is ignored, and may be NULL. If the **oldct* parameter is not 0, *old* must be an array of `__sigactionset_t` structures that the system can fill in. The number of array entries in *old* must be at least **oldct*. If not 0, **oldct* must be large enough to allow the system to pass back all the unique actions currently associated with all signals. If **oldct* is not large enough, __sigactionset() will fail and the errno will be set to ENOMEM.

If __sigactionset() returns with no error and **oldct* was not 0, **oldct* is set to the number of `__sigactionset_t` array entries in *old* that are filled in. If **oldct* was too small, causing an `ENOMEM` error, **oldct* is set to the number of `__sigactionset_t` structures the system would need in order to fill in all the distinct current signal actions. If **oldct* was 0 when __sigactionset() was called, it is not updated.

__sigactionset_t *old***[]**

old is an optional array of `__sigactionset_t` structures.

When **oldct* is not 0, the structures in the *old* array will be filled in with the signal actions currently in effect before any changes are made. The `__sigactionset_t` structure entries in *old* are filled in with all the distinct signal actions currently in effect, starting with the first array entry. Each `__sigactionset_t` structure in the array will contain information about one or more signals. Bits in the `__sa_signals` signal set in each array entry will indicate which signals that entry applies to. The system will try to use as few array entries as possible when passing back the different signal actions. The signal actions for `SIGKILL`, `SIGSTOP`, or `SIGTRACE` will not be returned.

The output information in each array entry is similar to that returned from sigaction(). In the `__sigactionset_t` structure, the fields `__sa_handler`, `__sa_mask`, `__sa_flags`, and `__sa_sigaction` correspond to the `sa_handler`, `sa_mask`, `sa_flags`, and `sa_sigaction` fields in the `sigaction` structure filled in by sigaction(). The signal action as described by these fields applies to all signals whose bits are on in the `__sa_signals` signal set in the array entry.

If *old* is not large enough to contain information about all distinct signal actions currently in effect, __sigactionset() fails, and `ENOMEM` is returned. There is no way to obtain the current signal actions for a specified subset of signals.

When *old* is NULL, the system does not return any information about the current signal actions.

`int` *options*

options is a collection of flag bits that affects the operation __sigactionset(). The following flag bit can be set in *options*:

`__SSET_IGINVALID`

Tells the system to ignore invalid bits in the `__sa_signals` field in all `__sigactionset_t` array entries in the *new* parameter. Also, the system will ignore attempts to set `SIGKILL`, `SIGSTOP` or `SIGTRACE` to an action other than `SIG_DFL`, or `SIGIO` to `SIG_IGN`.

If this option bit is off, the system will fail the __sigactionset() request if any invalid bits are found in any `__sa_signals` signal set in any *new* array entry. Also, __sigactionset() will fail if an attempt it made to set `SIGKILL`, `SIGSTOP`, or `SIGTRACE` to something other than `SIG_DFL`, or to set `SIGIO` to `SIG_IGN`.

This function is supported only in a POSIX(ON) program.

Special behavior for C++: The behavior when mixing signal-handling with C++ exception handling is undefined. Also, the use of signal-handling with constructors and destructors is undefined.

C++ and C language linkage conventions are incompatible, and therefore __sigactionset() cannot receive C++ function pointers. If you attempt to pass a C++ function pointer to __sigactionset(), the compiler will flag it as an error. Therefore to use the __sigactionset() function in the C++ language, you must ensure that signal handler routines have C linkage, by declaring them as extern "C".

Special behavior for XPLINK-compiled C++: Restrictions concerning setjmp.h and ucontext.h:

1. All XPLINK programs compiled with the V2R10 or later C compilers that are to run with Language Environment V2R10 or later libraries and use the **jmp_buf**, **sigjmp_buf** or **ucontext_t** types must not be compiled with C headers from Language Environment V2R9 or earlier.
2. Non-XPLINK functions compiled with any level of Language Environment headers must not define **jmp_buf**, **sigjmp_buf** or **ucontext_t** data items and pass them to XPLINK functions that call getcontext(), longjmp(), _longjmp(), setjmp(), _setjmp(), setcontext(), sigsetjmp(), or swapcontext() with these passed-in data items.
3. When __XPLINK__ is defined, the Language Environment V2R10 and later headers define a larger **jmp_buf**, **sigjmp_buf** or **ucontext_t** area that is required by setjmp(), getcontext(), and related functions when they are called from an XPLINK routine. If __XPLINK__ is not defined, the Language Environment V2R10 and later headers define a shorter **jmp_buf**, **sigjmp_buf** or **ucontext_t** area. The Language Environment headers before V2R10 also define the shorter version of these data areas. If an XPLINK function calls setjmp(), getcontext() or similar functions with a short **jmp_buf**, **sigjmp_buf** or **ucontext_t** area, a storage overlay or program check may occur when the C library tries to store past the end of the passed-in (too short) data area.

Usage note

The use of the SIGTHSTOP and SIGTHCONT signal is not supported with this function.

__sigactionset_t type

The __sigactionset_t type is defined as follows:

```
typedef struct __sigactionset_s
{
  sigset_t   __sa_signals;
  int        __sa_flags;
  void     (*__sa_handler)(int);
  sigset_t   __sa_mask;

  void     (*__sa_sigaction)(int, siginfo_t *, void *);
} __sigactionset_t;
```

The following are members of the structure:

sigset_t __sa_signals

This is a signal set. It contains the signals whose actions are described by the other members in this structure. For more information on signal sets, see "sigemptyset() — Initialize a signal mask to exclude all signals" on page 1633.

In the *new* array of _sigactionset_t structures, the caller sets bits in this signal set. The signal action for each signal in the signal set will be set as described by the other members of the structure.

__sa_signals must be set using one or more of the signal set manipulation functions: sigaddset(), sigdelset(), sigemptyset(), or sigfillset().

In the *old* array of __sigactionset_t structures, the system sets the bits in the __sa_signals field. The current signal action for each member of the signal set is described by the other members of the structure. All signals in the set have the same signal action.

int __sa_flags

A collection of flag bits that affect the behavior of the specified signal.

The flag bits in the __sa_flags field are the same as those in the sa_flags member of the sigaction structure. See "sigaction() — Examine or change a signal action" on page 1611 for a detailed description of these flag bits.

void (* __sa_handler)(int)

A pointer to the function assigned to handle the signals in the __sa_signals signal set. This function will be invoked passing one parameter of type int that contains the signal type for which this function is being invoked. The value of this member can also be SIG_DFL (indicating the default action) or SIG_IGN (indicating that the signal is to be ignored).

Note: This member and __sa_sigaction are mutually exclusive. When the SA_SIGINFO flag is set in __sa_flags, __sa__sigaction is used. Otherwise, __sa_handler is used.

sigset_t __sa_mask

This signal set identifies a set of signals that are to be added to the signal mask of the calling thread before the signal-handling function __sa_handler or __sa_sigaction is invoked. For more information on signal sets, see "sigemptyset() — Initialize a signal mask to exclude all signals" on page 1633. You cannot use this mechanism to block SIGKILL, SIGSTOP, or SIGTRACE. If __sa_mask includes these signals, they will simply be ignored; __sigactionset() will not return an error.

__sa_mask must be set by using one or more of the signal set manipulation functions: sigaddset(), sigdelset(), sigemptyset(), or sigfillset().

void (*_sa_sigaction)(int, siginfo_t *, void *)
A pointer to the function assigned to handle the signal, or SIG_DFL, or SIG_IGN. This function will be invoked passing three parameters. The first is of type int that contains the signal type for which this function is being invoked. The second is of type siginfo_t* where the siginfo_t contain additional information about the source of the signal. The third is of type void* but will actually point to a ucontext_t containing the context information at the of time the signal interrupt.

Notes:

1. The user must cast SIG_DFL or SIG_IGN to match the __sa_sigaction definition.
2. This member and sa_handler are mutually exclusive. When the SA_SIGINFO flag is set in __sa_flags, __sa_sigaction is used. Otherwise, __sa_handler is used.

When a signal handler installed by __sigactionset(), with the _SA_OLD_STYLE flag set off, catches a signal, the system calculates a new signal mask by taking the union of the current signal mask at the time of the signal interrupt, the signals specified by __sa_mask, and the signal that was just caught (if the SA_NODEFER flag is not set). This new mask stays in effect until the signal handler returns, or sigprocmask(), sigsuspend(), siglongjmp(), sighold(), sigpause(), or sigrelse() is called. When the signal handler ends, the original signal mask is restored.

After an action has been specified for a particular signal, using __sigactionset() with the _SA_OLD_STYLE flag not set, it remains installed until it is explicitly changed with another call to __sigactionset(), sigaction(), signal(), bsd_signal(), sigset(), sigignore(), one of the exec functions, or until the SA_RESETHAND flag causes it to be reset to SIG_DFL.

After an action has been specified for a particular signal, using __sigactionset() with the _SA_OLD_STYLE flag set or using signal(), it remains installed until it is explicitly changed with another call to __sigactionset(), sigaction(), bsd_signal(), sigset(), signal(), sigignore(), one of the exec functions, or a signal catcher is driven, where it will be reset to SIG_DFL.

Successful setting of a signal action to SIG_IGN for a signal that is pending causes the pending signal to be discarded, whether or not it is blocked. This provides the ability to discard signals that are found to be blocked and pending by sigpending(). A signal is discarded across a call to __sigactionset() if any __sigactionset_t structure in the *new* array causes the action for that signal to be set to SIG_IGN. This happens even if a later __sigactionset_t structure in the *new* array sets the signal action to something other than SIG_IGN before __sigactionset() returns.

If a process sets the action of the SIGCHLD signal to SIG_IGN, child processes of the calling process will not be transformed into zombie processes when they terminate. If the calling process subsequently waits for its children, and the process has no unwaited from children that were transformed into zombie processes, it will block until all of its children terminate. The wait(), waitid(), or waitpid() function will fail and set errno to ECHILD.

If the SA_SIGINFO flag is set, the signal catching function specified by __sa_sigaction is invoked as:

```
void function(int signo, siginfo_t *info, void * context);
```

where *function* is the specified signal-catching function, *signo* is the signal number of the signal being delivered, *info* points to an object of type siginfo_t associated with the signal being delivered, and *context* points to an object of type ucontext_t.

For a signal catcher that has been loaded by fetch() or fetchep(), the address returned by __sigactionset() in the __sa_handler or __sa_sigaction fields may be different than the value originally passed in to sigaction() or __sigactionset() (when the signal action was first set). This signal catcher address can be passed in again to sigaction() or __sigactionset() to reestablish the same signal catcher. The effect will be similar to passing in the original catcher address obtained from fetch() or fetchep(). However, this address should not be used for any other purpose, such as directly calling the signal catcher. Always use the original address obtained from fetch() or fetchep() when calling the catcher directly.

Considerations for asynchronous signal-catching functions

Some of the functions have been restricted to be serially reusable with respect to asynchronous signals. For more information on these functions, see “sigaction() — Examine or change a signal action” on page 1611.

Returned value

If successful, __sigactionset() returns 0.

If unsuccessful, no signal actions are changed, __sigactionset() returns -1 and sets errno to one of the following values:

Error Code
Description

EINVAL
This error can occur if:

- An unsupported signal bit was on in the __sa_signals signal set in the *new* parameter. This error will not be reported if the __SSET_IGINVALID flag is set in *options*. To obtain more information in this case, use __errno2().
- An attempt was made to set the signal action for SIGSTOP, SIGKILL, or SIGTRACE to something other than SIG_DFL. This error will not be reported if the __SSET_IGINVALID flag is set in *options*. To obtain more information in this case, use __errno2().
- The *newct* or *oldct* parameters are not in the range from 0 to 64.
- *newct* was not 0 and *new* was NULL.
- *oldct* was not 0 and *old* was NULL.

EMVSERR
An MVS environmental or internal error has occurred. Use __errno2() to obtain more information about this error.

ENOMEM

The input value in *oldct* was not 0, and was too small to let the system pass back all distinct current signal actions. When __sigactionset() returns, **oldct* will be set to the number of array entries needed by the system.

Example

```
/*
 *  Note: This is just a code fragment
 */

void catch_sigchld(int, siginfo_t *, void *);

 ...

 __sigactionset_t new[2], old[64];
 int              options;
 int              rc;
 size_t           oldct = 64;


/*
 *   Set SIGUSR1 and SIGUSR2 to SIG_IGN
 *   Set SIGCHLD to new-style catcher catch_sigchld()
 *   Save original signal setup in variable old
 */

 bzero(new, sizeof new);

 (void)sigemptyset(&(new[0].__sa_signals)         );
 (void)sigaddset  (&(new[0].__sa_signals), SIGUSR1);
 (void)sigaddset  (&(new[0].__sa_signals), SIGUSR2);

 (void)sigemptyset(&(new[1].__sa_signals)         );
 (void)sigaddset  (&(new[1].__sa_signals), SIGCHLD);

 new[0].__sa_handler   = SIG_IGN;
 new[1].__sa_sigaction = &catch_sigchld;
 new[1].__sa_flags     = SA_SIGINFO;

 rc = __sigactionset((size_t)2, new, &oldct, old, __SSET_IGINVALID);
```

Related information

- "signal.h" on page 63
- "alarm() — Set an alarm" on page 156
- "bsd_signal() — BSD version of signal()" on page 219
- "exec functions" on page 436
- "getcontext() — Get user context" on page 694
- "kill() — Send a signal to a process" on page 931
- "makecontext() — Modify user context" on page 1028
- "raise() — Raise signal" on page 1371
- "setcontext() — Restore user context" on page 1524
- "sigaction() — Examine or change a signal action" on page 1611
- "sigaddset() — Add a signal to the signal mask" on page 1628
- "sigaltstack() — Set or get signal alternate stack context" on page 1630
- "sigdelset() — Delete a signal from the signal mask" on page 1632
- "sigemptyset() — Initialize a signal mask to exclude all signals" on page 1633
- "sigfillset() — Initialize a signal mask to include all signals" on page 1635
- "sigignore() — Set disposition to ignore a signal" on page 1637
- "siginterrupt() — Allow signals to interrupt functions" on page 1638
- "siglongjmp() — Restore the stack environment and signal mask" on page 1640
- "signal() — Handle interrupts" on page 1643

- “sigprocmask() — Examine or change a thread” on page 1651
- “sigset() — Change a signal action or a thread” on page 1656
- “sigstack() — Set or get signal stack context” on page 1662
- “sigsuspend() — Change mask and suspend the thread” on page 1663
- “swapcontext() — Save and restore user context” on page 1792
- “wait() — Wait for a child process to end” on page 1992
- “wait3() — Wait for child process to change state” on page 2000

sigaddset() — Add a signal to the signal mask

Standards

Standards / Extensions	C or C++	Dependencies
POSIX.1 XPG4 XPG4.2 Single UNIX Specification, Version 3	both	

Format

```
#define _POSIX_SOURCE
#include <signal.h>

int sigaddset(sigset_t *set, int signal);
```

General description

Adds a signal to the set of signals already recorded in *set*.

sigaddset() is part of a family of functions that manipulate signal sets. *Signal sets* are data objects that let a process keep track of groups of signals. For example, a process can create one signal set to record which signals it is blocking, and another signal set to record which signals are pending. In general, signal sets are used to manipulate groups of signals used by other functions (such as sigprocmask()) or to examine signal sets returned by other functions (such as sigpending()).

Applications should call either sigemptyset() or sigfillset() at least once for each object of type sigset_t prior to any other use of that object. If such an object is not initialized in this way, but is nonetheless supplied as an argument to any of pthread_sigmask(), sigaction(), sigaddset(), sigdelset(), sigismember(), sigpending(), sigprocmask(), sigsuspend(), sigtimedwait(), sigwait(), or sigwaitinfo(), the results are undefined.

Usage note

The use of the SIGTHSTOP and SIGTHCONT signal is not supported with this function.

Returned value

If the signal is successfully added to the signal set, sigaddset() returns 0.

If *signal* is not supported, sigaddset() returns -1 and sets errno to EINVAL.

Example

CELEBS15

```
/* CELEBS15

   This example adds a set of signals.

 */
#define _POSIX_SOURCE
#include <stdio.h>
#include <signal.h>
#include <unistd.h>

void catcher(int signum) {
  puts("catcher() has gained control");
}

main() {
  struct   sigaction sact;
  sigset_t sigset;

  sigemptyset(&sact.sa_mask);
  sact.sa_flags = 0;
  sact.sa_handler = catcher;
  sigaction(SIGUSR1, &sact, NULL);

  puts("before first kill()");
  kill(getpid(), SIGUSR1);
  puts("before second kill()");

  sigemptyset(&sigset);
  sigaddset(&sigset, SIGUSR1);
  sigprocmask(SIG_SETMASK, &sigset, NULL);

  kill(getpid(), SIGUSR1);
  puts("after second kill()");
}
```

Output

```
before first kill()
catcher() has gained control
before second kill()
after second kill()
```

Related information

- "signal.h" on page 63
- "sigaction() — Examine or change a signal action" on page 1611
- "sigdelset() — Delete a signal from the signal mask" on page 1632
- "sigemptyset() — Initialize a signal mask to exclude all signals" on page 1633
- "sigfillset() — Initialize a signal mask to include all signals" on page 1635
- "sighold() — Add a signal to a thread" on page 1636
- "sigismember() — Test if a signal is in a signal mask" on page 1638
- "signal() — Handle interrupts" on page 1643
- "sigpause() — Unblock a signal and wait for a signal" on page 1649
- "sigpending() — Examine pending signals" on page 1650
- "sigprocmask() — Examine or change a thread" on page 1651
- "sigset() — Change a signal action or a thread" on page 1656
- "sigsuspend() — Change mask and suspend the thread" on page 1663

sigaltstack() — Set or get signal alternate stack context

Standards

Standards / Extensions	C or C++	Dependencies
XPG4.2 Single UNIX Specification, Version 3	both	POSIX(ON)

Format

```
#define _XOPEN_SOURCE_EXTENDED 1
#include <signal.h>

int sigaltstack(const stack_t *__restrict__ ss, stack_t *__restrict__ oss);
```

General description

The sigaltstack() function allows a thread to define and examine the state of an alternate stack for signal handlers. Signals that have been explicitly declared to execute on the alternate stack will be delivered on the alternate stack.

Note: To explicitly declare that a signal catcher is to run on the alternate signal stack, the SA_ONSTACK flag must be set in the sa_flags when the signal action is set using sigaction().

If *ss* is not a NULL pointer, it points to a `stack_t` structure that specifies the alternate signal stack that will take effect upon return from sigaltstack(). The `ss_flags` member specifies the new stack state. If it is set to SS_DISABLE, the stack is disabled and `ss_sp` and `ss_size` are ignored. Otherwise the stack will be enabled, and the `ss_sp` and `ss_size` members specify the new address and size of the stack.

AMODE 64 considerations: Storage for this stack must be above the 2GB bar. It may not be storage acquired with the __malloc24() or __malloc31() functions.

The range of addresses starting at `ss_sp`, up to but not including `ss_sp` + `ss_size`, is available to the implementation for use as the stack. This interface makes no assumptions regarding which end is the stack base and in which direction the stack grows as items are pushed.

If *oss* is not a NULL pointer, on successful completion it will point to a `stack_t` structure that specifies the alternate signal stack that was in effect before the call to sigaltstack(). The `ss_sp` and `ss_size` members specify the address and size of that stack. The `ss_flags` member specifies the stack's state, and may contain one of the following values:

SS_ONSTACK
The thread is currently executing on the alternate signal stack. Attempts to modify the alternate signal stack while the thread is executing on it fails. This flag must not be modified by threads.

SS_DISABLE
The alternate signal stack is currently disabled.

The value SIGSTKSZ is a system default specifying the number of bytes that would be used to cover the usual case when manually allocating an alternate stack area. The value `MINSIGSTKSZ` is defined to be the minimum stack size for a signal handler. In computing an alternate signal stack size, a program should add that amount to its stack requirements to allow for the system implementation overhead. The constants SS_ONSTACK, SS_DISABLE, SIGSTKSZ, and MINSIGSTKSZ are defined in <signal.h>.

After a successful call to one of the exec functions, there are no alternate signal stacks in the new process image.

Notes:

1. If a signal handler is enabled to run on an alternate stack, then all functions called by that signal handler must be compiled with the same linkage. For example, if the signal handler is compiled with XPLINK, then all functions it calls must also be compiled XPLINK. Since only one alternate stack can be supplied, no mixing of linkages (which would require both upward and downward-growing alternate stacks) is allowed. The type of stack created will be based on the attributes of the signal handler to be given control. If the signal handler has been compiled with XPLINK, then a downward-growing stack will be created in the alternate stack, including, in AMODE 31, using enough storage in the user stack to create a 4k read-only guard page (aligned on a 4k boundary).
2. If a new signal is received while a signal handler is running on an alternate stack, and that new signal specified a signal handler that also runs on the alternate stack, then both signal handlers must have been compiled with the same linkage (XPLINK versus non-XPLINK).

Returned value

If successful, sigaltstack() returns 0.

If unsuccessful, sigaltstack() returns -1 and sets errno to one of the following values:

Error Code
Description

EINVAL
ss argument is not a NULL pointer, and the `ss_flags` member pointed to by *ss* contains flags other than SS_DISABLE.

ENOMEM
The size of the alternate stack area is less than MINSIGSTKSZ.

EPERM
An attempt was made to modify an active stack.

Related information

- “signal.h” on page 63
- “sigaction() — Examine or change a signal action” on page 1611
- “sigsetjmp() — Save stack environment and signal mask” on page 1659
- “sigstack() — Set or get signal stack context” on page 1662

sigdelset() — Delete a signal from the signal mask

Standards

Standards / Extensions	C or C++	Dependencies
POSIX.1 XPG4 XPG4.2 Single UNIX Specification, Version 3	both	

Format

```
#define _POSIX_SOURCE
#include <signal.h>

int sigdelset(sigset_t *set, int signal);
```

General description

Removes the specified *signal* from the list of signals recorded in *set*.

The sigdelset() function is part of a family of functions that manipulate signal sets. *Signal sets* are data objects that let a process keep track of groups of signals. For example, a process can create one signal set to record which signals it is blocking, and another signal set to record which signals are pending. In general, signal sets are used to manipulate groups of signals used by other functions (such as sigprocmask()) or to examine signal sets returned by other functions (such as sigpending()).

Applications should call either sigemptyset() or sigfillset() at least once for each object of type sigset_t prior to any other use of that object. If such an object is not initialized in this way, but is nonetheless supplied as an argument to any of pthread_sigmask(), sigaction(), sigaddset(), sigdelset(), sigismember(), sigpending(), sigprocmask(), sigsuspend(), sigtimedwait(), sigwait(), or sigwaitinfo(), the results are undefined.

Usage note

The use of the SIGTHSTOP and SIGTHCONT signal is not supported with this function.

Returned value

If the signal is successfully deleted from the signal set, sigdelset() returns 0.

If *signal* is not supported, sigdelset() returns -1 and sets errno to EINVAL.

Example

CELEBS16

```
/* CELEBS16

   This example deletes specific signals.

 */
#define _POSIX_SOURCE
#include <stdio.h>
#include <signal.h>
#include <unistd.h>
```

```
void catcher(int signum) {
  puts("catcher() has gained control");
}

main() {
  struct   sigaction sact;
  sigset_t sigset;

  sigemptyset(&sact.sa_mask);
  sact.sa_flags = 0;
  sact.sa_handler = catcher;
  sigaction(SIGUSR1, &sact, NULL);

  sigfillset(&sigset);
  sigprocmask(SIG_SETMASK, &sigset, NULL);

  puts("before kill()");
  kill(getpid(), SIGUSR1);

  puts("before unblocking SIGUSR1");
  sigdelset(&sigset, SIGUSR1);
  sigprocmask(SIG_SETMASK, &sigset, NULL);
  puts("after unblocking SIGUSR1");
}
```

Output

```
before kill()
before unblocking SIGUSR1
catcher() has gained control
after unblocking SIGUSR1
```

Related information

- “signal.h” on page 63
- “sigaction() — Examine or change a signal action” on page 1611
- “sigaddset() — Add a signal to the signal mask” on page 1628
- “sigemptyset() — Initialize a signal mask to exclude all signals”
- “sigfillset() — Initialize a signal mask to include all signals” on page 1635
- “sighold() — Add a signal to a thread” on page 1636
- “sigismember() — Test if a signal is in a signal mask” on page 1638
- “signal() — Handle interrupts” on page 1643
- “sigpause() — Unblock a signal and wait for a signal” on page 1649
- “sigpending() — Examine pending signals” on page 1650
- “sigprocmask() — Examine or change a thread” on page 1651
- “sigset() — Change a signal action or a thread” on page 1656
- “sigsuspend() — Change mask and suspend the thread” on page 1663

sigemptyset() — Initialize a signal mask to exclude all signals

Standards

Standards / Extensions	C or C++	Dependencies
POSIX.1 XPG4 XPG4.2 Single UNIX Specification, Version 3	both	

Format

```
#define _POSIX_SOURCE
#include <signal.h>

int sigemptyset(sigset_t *set);
```

General description

Initializes a signal set *set* to the empty set. All recognized signals are excluded.

sigemptyset() is part of a family of functions that manipulate signal sets. *Signal sets* are data objects that let a process keep track of groups of signals. For example, a process can create one signal set to record which signals it is blocking, and another signal set to record which signals are pending. Signal sets are used to manipulate groups of signals used by other functions (such as sigprocmask()) or to examine signal sets returned by other functions (such as sigpending()).

Returned value

If successful, sigemptyset() returns 0.

There are no documented errno values.

Example

CELEBS17

```
/* CELEBS17

   This example initializes a set of signals to an empty set.

 */
#define _POSIX_SOURCE
#include <stdio.h>
#include <signal.h>
#include <unistd.h>

main() {
  struct sigaction sact;

  sigemptyset(&sact.sa_mask);
  sact.sa_flags = 0;
  sact.sa_handler = SIG_IGN;

  sigaction(SIGUSR2, &sact, NULL);

  puts("before kill()");
  kill(getpid(), SIGUSR2);
  puts("after kill()");
}
```

Output

```
before kill()
after kill()
```

Related information

- “signal.h” on page 63
- “sigaction() — Examine or change a signal action” on page 1611
- “sigaddset() — Add a signal to the signal mask” on page 1628
- “sigdelset() — Delete a signal from the signal mask” on page 1632
- “sigfillset() — Initialize a signal mask to include all signals” on page 1635

- “sighold() — Add a signal to a thread” on page 1636
- “sigismember() — Test if a signal is in a signal mask” on page 1638
- “signal() — Handle interrupts” on page 1643
- “sigpause() — Unblock a signal and wait for a signal” on page 1649
- “sigpending() — Examine pending signals” on page 1650
- “sigprocmask() — Examine or change a thread” on page 1651
- “sigset() — Change a signal action or a thread” on page 1656
- “sigsuspend() — Change mask and suspend the thread” on page 1663

sigfillset() — Initialize a signal mask to include all signals

Standards

Standards / Extensions	C or C++	Dependencies
POSIX.1 XPG4 XPG4.2 Single UNIX Specification, Version 3	both	

Format

```
#define _POSIX_SOURCE
#include <signal.h>

int sigfillset(sigset_t *set);
```

General description

Initializes a signal set *set* to the complete set of supported signals.

sigfillset() is part of a family of functions that manipulate signal sets. *Signal sets* are data objects that let a process keep track of groups of signals. For example, a process can create one signal set to record which signals it is blocking, and another signal set to record which signals are pending. Signal sets are used to manipulate groups of signals used by other functions (such as sigprocmask()) or to examine signal sets returned by other functions (such as sigpending()).

Returned value

If successful, sigfillset() returns 0.

There are no documented errno values.

Example

CELEBS18

```
/* CELEBS18

   This example initializes a set of signals to a complete set.

 */
#define _POSIX_SOURCE
#include <stdio.h>
#include <signal.h>
#include <unistd.h>

main() {
  sigset_t sigset;
  sigfillset(&sigset);

  sigprocmask(SIG_SETMASK, &sigset, NULL);

  puts("before kill()");
  kill(getpid(), SIGSEGV);
  puts("after kill()");
}
```

Output

```
before kill()
after kill()
```

Related information

- “signal.h” on page 63
- “sigaction() — Examine or change a signal action” on page 1611
- “sigaddset() — Add a signal to the signal mask” on page 1628
- “sigdelset() — Delete a signal from the signal mask” on page 1632
- “sigemptyset() — Initialize a signal mask to exclude all signals” on page 1633
- “sighold() — Add a signal to a thread”
- “sigismember() — Test if a signal is in a signal mask” on page 1638
- “signal() — Handle interrupts” on page 1643
- “sigpause() — Unblock a signal and wait for a signal” on page 1649
- “sigpending() — Examine pending signals” on page 1650
- “sigprocmask() — Examine or change a thread” on page 1651
- “sigset() — Change a signal action or a thread” on page 1656
- “sigsuspend() — Change mask and suspend the thread” on page 1663

sighold() — Add a signal to a thread

Standards

Standards / Extensions	C or C++	Dependencies
XPG4.2 Single UNIX Specification, Version 3	both	

Format

```
#define _XOPEN_SOURCE_EXTENDED 1
#include <signal.h>

int sighold(int sig);
```

General description

The sighold() function provides a simplified method for adding the signal specified by the argument *sig* to the calling thread's signal mask.

Usage note

The use of the SIGTHSTOP and SIGTHCONT signal is not supported with this function.

Returned value

If successful, sighold() returns 0.

If unsuccessful, sighold() returns -1 and sets errno to one of the following values:

Error Code
Description

EINVAL
The value of the argument *sig* is not a valid signal type or it is SIGKILL, SIGSTOP, or SIGTRACE.

Related information

- “signal.h” on page 63
- “sigaddset() — Add a signal to the signal mask” on page 1628
- “sigpause() — Unblock a signal and wait for a signal” on page 1649
- “sigprocmask() — Examine or change a thread” on page 1651
- “sigrelse() — Remove a signal from a thread” on page 1656

sigignore() — Set disposition to ignore a signal

Standards

Standards / Extensions	C or C++	Dependencies
XPG4.2 Single UNIX Specification, Version 3	both	POSIX(ON)

Format

```
#define _XOPEN_SOURCE_EXTENDED 1
#include <signal.h>

int sigignore(int sig);
```

General description

The sigignore() function provides a simplified method for setting the signal action of the signal specified by the argument *sig* to SIG_IGN.

Usage note

The use of the SIGTHSTOP and SIGTHCONT signal is not supported with this function.

Returned value

If successful, sigignore() returns 0.

If unsuccessful, sigignore() returns -1 and sets errno to one of the following values:

Error Code
Description

EINVAL
The value of the argument *sig* is not a valid signal type or it is SIGKILL, SIGSTOP, or SIGTRACE.

Related information

- “signal.h” on page 63
- “bsd_signal() — BSD version of signal()” on page 219
- “sigaction() — Examine or change a signal action” on page 1611
- “signal() — Handle interrupts” on page 1643
- “sigset() — Change a signal action or a thread” on page 1656

siginterrupt() — Allow signals to interrupt functions

Standards

Standards / Extensions	C or C++	Dependencies
XPG4.2 Single UNIX Specification, Version 3	both	POSIX(ON)

Format

```
#define _XOPEN_SOURCE_EXTENDED 1
#include <signal.h>

int siginterrupt(int sig, int flag);
```

General description

The siginterrupt() function provides a simplified method for changing the restart behavior when a function is interrupted by the signal specified in the argument *sig*.

The argument *flag* serves as a binary switch to enable or disable restart behavior. When *flag* is nonzero, restart behavior will be disabled. Otherwise it is enabled.

Returned value

If successful, siginterrupt() returns 0.

If unsuccessful, siginterrupt() returns -1 and sets errno to one of the following values:

Error Code
  Description

EINVAL
  The value of the argument *sig* is not a valid signal type.

Related information

- “signal.h” on page 63
- “sigaction() — Examine or change a signal action” on page 1611

sigismember() — Test if a signal is in a signal mask

Standards

Standards / Extensions	C or C++	Dependencies
POSIX.1 XPG4 XPG4.2 Single UNIX Specification, Version 3	both	

Format

```
#define _POSIX_SOURCE
#include <signal.h>

int sigismember(const sigset_t *set, int signal);
```

General description

Tests whether a specified signal number *signal* is a member of a signal set *set*.

sigismember() is part of a family of functions that manipulate signal sets. *Signal sets* are data objects that let a process keep track of groups of signals. For example, a process can create one signal set to record which signals it is blocking, and another signal set to record which signals are pending. Signal sets are used to manipulate groups of signals used by other functions (such as sigprocmask()) or to examine signal sets returned by other functions (such as sigpending()).

Applications should call either sigemptyset() or sigfillset() at least once for each object of type sigset_t prior to any other use of that object. If such an object is not initialized in this way, but is nonetheless supplied as an argument to any of pthread_sigmask(), sigaction(), sigaddset(), sigdelset(), sigismember(), sigpending(), sigprocmask(), sigsuspend(), sigtimedwait(), sigwait(), or sigwaitinfo(), the results are undefined.

Usage note

The use of the `SIGTHSTOP` and `SIGTHCONT` signal is not supported with this function.

Returned value

sigismember() returns 1 if *signal* is in *set*, and it returns 0 if it is not.

If unsuccessful, sigismember() returns -1 and sets errno to one of the following values:

Error Code
    Description

EINVAL
    The value of *signal* is not one of the supported signals.

Example

CELEBS19

```
/* CELEBS19

   This example tests signals.

 */
#define _POSIX_SOURCE
#include <stdio.h>
#include <signal.h>

void check(sigset_t set, int signum, char *signame) {
  printf("%-8s is ", signame);
  if (!sigismember(&set, signum))
    printf("not ");
  puts("in the set");
}

main() {
  sigset_t sigset;

  sigemptyset(&sigset);
  sigaddset(&sigset, SIGUSR1);
  sigaddset(&sigset, SIGKILL);
  sigaddset(&sigset, SIGCHLD);

  check(sigset, SIGUSR1, "SIGUSR1");
  check(sigset, SIGUSR2, "SIGUSR2");
  check(sigset, SIGFPE,  "SIGFPE");
  check(sigset, SIGKILL, "SIGKILL");
}
```

Output

```
SIGUSR1  is in the set
SIGUSR2  is not in the set
SIGFPE   is not in the set
SIGKILL  is in the set
```

Related information

- “signal.h” on page 63
- “sigaction() — Examine or change a signal action” on page 1611
- “sigaddset() — Add a signal to the signal mask” on page 1628
- “sigdelset() — Delete a signal from the signal mask” on page 1632
- “sigemptyset() — Initialize a signal mask to exclude all signals” on page 1633
- “sigfillset() — Initialize a signal mask to include all signals” on page 1635
- “sighold() — Add a signal to a thread” on page 1636
- “signal() — Handle interrupts” on page 1643
- “sigpause() — Unblock a signal and wait for a signal” on page 1649
- “sigpending() — Examine pending signals” on page 1650
- “sigprocmask() — Examine or change a thread” on page 1651
- “sigset() — Change a signal action or a thread” on page 1656
- “sigsuspend() — Change mask and suspend the thread” on page 1663

siglongjmp() — Restore the stack environment and signal mask

Standards

Standards / Extensions	C or C++	Dependencies
POSIX.1 XPG4 XPG4.2 Single UNIX Specification, Version 3	both	

Format

```
#define _POSIX_SOURCE
#include <setjmp.h>

void siglongjmp(sigjmp_buf env, int val);
```

General description

For a stack environment previously saved in *env* by sigsetjmp(), the siglongjmp() function restores all the stack environment and, optionally, the signal mask, depending on whether it was saved by sigsetjmp(). The sigsetjmp() and siglongjmp() functions provide a way to perform a nonlocal `goto`.

env is an address for a `sigjmp_buf` structure.

val is the return value from siglongjmp().

siglongjmp() is similar to longjmp(), except for the optional capability of restoring the signal mask. The sigsetjmp()—siglongjmp() pair, the setjmp()—longjmp() pair, the _setjmp()—_longjmp() pair, and the getcontext()—setcontext() pair cannot be intermixed. A stack environment and signal mask saved by sigsetjmp() can be restored only by siglongjmp().

A call to sigsetjmp() causes the current stack environment including, optionally, the signal mask to be saved in *env*. A subsequent call to siglongjmp() restores the saved environment and signal mask (if saved by sigsetjmp()) and returns control to a point in the program corresponding to the sigsetjmp() call. Execution resumes as if the sigsetjmp() call had just returned the given *value*. All variables (except register variables) that are accessible to the function that receives control contain the values they had when you called siglongjmp(). The values of register variables are unpredictable. Nonvolatile auto variables that are changed between calls to sigsetjmp() and siglongjmp() are also unpredictable.

Notes:

1. If you call siglongjmp(), the function in which the corresponding call to sigsetjmp() was made must not have returned first. After the function calling sigsetjmp() returns, calling siglongjmp() causes unpredictable program behavior.
2. If siglongjmp() is used to jump back into an XPLINK routine, any alloca() requests issued by the XPLINK routine after the earlier sigsetjmp() (or getcontext(),and so on.) was called and before siglongjmp() is called are backed out. All storage obtained by these alloca() requests is freed before the XPLINK routine is resumed.
3. If siglongjmp() is used to jump back into a non-XPLINK routine, alloca() requests made after sigsetjmp() and before siglongjmp() are not backed out.

The *value* argument passed to siglongjmp() must be nonzero. If you give a zero argument for *value*, siglongjmp() substitutes the value 1 in its place.

siglongjmp() does not use the normal function call and return mechanisms. siglongjmp() restores the saved signal mask only if the *env* parameter was initialized by a call to sigsetjmp() with a nonzero *savemask* argument.

Special behavior for C++: If sigsetjmp() and siglongjmp() are used to transfer control in a z/OS XL C++ program, the behavior is undefined in terms of the destruction of automatic objects. Additionally, if any automatic objects would be destroyed by a thrown exception transferring control to another (destination) point in the program, then a call to siglongjmp() at the throw point that transfers control to the same (destination) point has undefined behavior. This applies to both z/OS XL C++ and z/OS XL C/C++ ILC modules. The use of sigsetjmp() and siglongjmp() in conjunction with try(), catch(), and throw() is also undefined.

Special behavior for XPLINK-compiled C++: Restrictions concerning setjmp.h and ucontext.h:

1. All XPLINK programs compiled with the V2R10 or later C compilers that are to run with Language Environment V2R10 or later libraries and use the **jmp_buf**, **sigjmp_buf** or **ucontext_t** types must not be compiled with C headers from Language Environment V2R9 or earlier.
2. Non-XPLINK functions compiled with any level of Language Environment headers must not define **jmp_buf**, **sigjmp_buf** or **ucontext_t** data items and pass them to XPLINK functions that call getcontext(), longjmp(), _longjmp(), setjmp(), _setjmp(), setcontext(), sigsetjmp(), or swapcontext() with these passed-in data items.

3. When __XPLINK__ is defined, the Language Environment V2R10 and later headers define a larger **jmp_buf**, **sigjmp_buf** or **ucontext_t** area that is required by setjmp(), getcontext(), and related functions when they are called from an XPLINK routine. If __XPLINK__ is not defined, the Language Environment V2R10 and later headers define a shorter **jmp_buf**, **sigjmp_buf** or **ucontext_t** area. The Language Environment headers before V2R10 also define the shorter version of these data areas. If an XPLINK function calls setjmp(), getcontext() or similar functions with a short **jmp_buf**, **sigjmp_buf** or **ucontext_t** area, a storage overlay or program check may occur when the C library tries to store past the end of the passed-in (too short) data area.

Returned value

siglongjmp() returns no values.

There are no documented errno values.

Example

This example saves the stack environment and signal mask at the statement: `if(sigsetjmp(mark,1) != 0) ...`

When the system first performs the `if` statement, it saves the environment and signal mask in *mark* and sets the condition to false, because sigsetjmp() returns 0 when it saves the environment. The program prints the message: `sigsetjmp() has been called`

The subsequent call to function p() tests for a local error condition, which can cause it to perform siglongjmp(). Then control returns to the original sigsetjmp() function using the environment saved in *mark* and restores the signal mask. This time the condition is true because -1 is the return value from siglongjmp(). The example then performs the statements in the block and prints: `siglongjmp() has been called` Then it performs your recover() function and leaves the program.

```
#define _POSIX_SOURCE
#include <stdio.h>
#include <setjmp.h>

sigjmp_buf mark;

void p(void);
void recover(void);

int main(void)
{
   if (sigsetjmp(mark) != 0) {
      printf("siglongjmp() has been called\n");
      recover();
      exit(1);
      }
   printf("sigsetjmp() has been called\n");
⋮
   p();
⋮
}

void p(void) {
   int error = 0;
⋮
   error = 9;
⋮
   if (error != 0)
      siglongjmp(mark, -1);
⋮
}

void recover(void) {
⋮
}
```

Related information

- “setjmp.h” on page 62
- “getcontext() — Get user context” on page 694
- “longjmp() — Restore stack environment” on page 1005
- “_longjmp() — Nonlocal goto” on page 1008
- “setcontext() — Restore user context” on page 1524
- “setjmp() — Preserve stack environment” on page 1546
- “_setjmp() — Set jump point for a nonlocal goto” on page 1549
- “sigaction() — Examine or change a signal action” on page 1611
- “sigprocmask() — Examine or change a thread” on page 1651
- “sigsetjmp() — Save stack environment and signal mask” on page 1659
- “sigsuspend() — Change mask and suspend the thread” on page 1663
- “swapcontext() — Save and restore user context” on page 1792

signal() — Handle interrupts

Standards

Standards / Extensions	C or C++	Dependencies
ISO C POSIX.1 POSIX.4a XPG4 XPG4.2 C99 Single UNIX Specification, Version 3	both	

Format

```
#include <signal.h>

void(*signal(int sig, void(*func)(int)))(int);
```

General description

Allows a process to choose one of several ways to handle an interrupt signal *sig* from the operating system or from the raise() function.

The *sig* argument must be one of the macros defined in the `signal.h` header file. See Table 50 on page 1645.

The *func* argument must be one of the macros, SIG_DFL or SIG_IGN, defined in the `signal.h` header file, or a function address.

If the value of *func* is SIG_DFL, default handling for that signal will occur. If the value of *func* is SIG_IGN, the signal will be ignored. Otherwise, *func* points to a function to be called when that signal occurs. Such a function is called a *signal handler*.

When a signal occurs, if *func* points to a function:

1. First the equivalent of `signal(sig,SIG_DFL);` is executed or an implementation-defined blocking of the system is performed. (If the value of *sig* is SIGILL, the occurrence of the reset to SIG_DFL is implementation-defined.)
2. Next, the equivalent of `(*func)(sig);` is executed. The function *func* may terminate by executing a `return` statement or by calling the abort(), exit(), or longjmp() function. If *func* executes a return statement and the value of *sig* was SIGFPE or any other implementation-defined value corresponding to a computational exception, the behavior is undefined. Otherwise, the program will resume execution at the point it was interrupted.

If a signal occurs for a reason other than having called the abort() or raise() function, the behavior is undefined if the signal handler calls any function in the standard library other than the signal() function itself (with a first argument of the signal number corresponding to the signal that caused the invocation of the handler). Behavior is also undefined if the signal handler refers to any object with static storage duration other than by assigning a value to a static storage duration variable of type `volatile sig_atomic_t`. Furthermore, if such a call to the signal() function returns SIG_ERR, the value of errno is indeterminate.

At program startup, the equivalent of `signal(sig, SIG_IGN);` may be executed for some selected signals. The equivalent of `signal(sig, SIG_DFL);` is executed for all other signals.

The action taken when the interrupt signal is received depends on the value of *func*.

Value Meaning

SIG_DFL
Default handling for the signal will occur.

SIG_IGN
The signal is to be ignored.

As of Language Environment Release 3, the defaults for SIGUSR1, SIGUSR2, SIGINT, and SIGTERM are changed from the signal being ignored to abnormal termination. To compensate for this change, you would explicitly register that the signal is to be ignored, using a call sequence such as:

```
signal(SIGUSR1, SIG_IGN);
signal(SIGUSR2, SIG_IGN);
signal(SIGINT,  SIG_IGN);
signal(SIGTERM, SIG_IGN);
```

These calls may be made either in the source or they can be made from the HLL user exit CEEBINT, which will require a re-link.

Special behavior for POSIX: For a z/OS UNIX C application running POSIX(ON), the interrupt signal can also come from kill() or from another process. A program can use sigaction() to establish a signal handler; sigaction() blocks the signal while the signal handler has control. If you use signal() to establish a signal handler, the signal reverts back to the default action. If you want the signal handler to get control for the next signal of this type, you must reissue signal().

`signal(sig, func)` is equivalent to `sigaction(sig, &act, NULL)`, where *act* points to a sigaction structure containing an sa_action of *func*, an sa_mask by sigemptyset(), and an sa_flags containing _SA_OLD_STYLE.

Note: The sigaction() function supersedes the signal() interface, and should be the preferred usage. In particular, sigaction() and signal() must not be used in the same process to control the same signal.

For a list of considerations for coding signal-catching functions that will support asynchronous signals, refer to “sigaction() — Examine or change a signal action” on page 1611.

The *sig* argument must be one of the macros defined in the signal.h header file.

Special behavior for C++:

- The behavior when mixing signal-handling with C++ exception handling is undefined. Also, the use of signal-handling with constructors and destructors is undefined.
- C++ and C language linkage conventions are incompatible, and therefore signal() cannot receive C++ function pointers. If you attempt to pass a C++ function pointer to signal(), the compiler will flag it as an error. Therefore, to use the signal() function in the C++ language, you must ensure that signal handler routines established have C linkage, by declaring them as `extern "C"`.

The signals supported are listed below.

Table 50. Signals Supported by C or C++ — POSIX(OFF)

Value	Default Action	Meaning
SIGABND	1	Abend
SIGABRT	1	Abnormal termination (sent by abort())
SIGFPE	1	Arithmetic exceptions that are not masked, for example, overflow, division by zero, and incorrect operation
SIGILL	1	Detection of an incorrect function image
SIGINT	1	Interactive attention
SIGSEGV	1	Incorrect access to memory
SIGTERM	1	Termination request sent to the program
SIGUSR1	1	Intended for use by user applications
SIGUSR2	1	Intended for use by user applications
SIGIOERR	2	A serious I/O error was detected.

In Table 50, the **Default Actions** are:

1 Normal termination of the process.

2 Ignore the signal.

When the runtime option POSIX(ON) is specified, if a signal catcher for a SIGABND, SIGFPE, SIGILL or SIGSEGV signal runs as a result of a program check or an ABEND, and the signal catcher executes a RETURN statement, the process will be terminated.

Returned value

If successful, signal() returns the most recent value of *func*.

If unsuccessful, signal() returns a value of SIG_ERR and a positive value in errno.

There are no documented errno values. If an error occurs, issue perror() using the errno value.

Example

CELEBS20

```
/* CELEBS20

   This example shows you how to establish a signal handler.

 */
#include <signal.h>
#include <stdio.h>
#include <stdlib.h>
#define  ONE_K  1024

#define OUT_OF_STORAGE       (SIGUSR1)

/* The SIGNAL macro does a signal() checking the return code */
#define SIGNAL(handler, StrCln)       {                \
  if (signal((handler), (StrCln)) == SIG_ERR) {        \
    perror("Could not signal user signal");            \
    abort();                                           \
  }                                                    \
}

#ifdef __cplusplus                /* the __cplusplus macro       */
extern "C" void StrCln(int);      /* is automatically defined    */
#else                             /* by the C++/MVS compiler     */
void StrCln(int);
#endif

void DoWork(char **, int);

int main(int argc, char *argv[]) {
  int size;
  char *buffer;

  signal(OUT_OF_STORAGE, StrCln);

  if (argc != 2) {
    printf("Syntax: %s size \n", argv[0]);
    return(-1);
  }

  size = atoi(argv[1]);

  DoWork(&buffer, size);
  return(0);
}

void StrCln(int SIG_TYPE) {
  printf("Failed trying to malloc storage\n");
  signal(SIG_TYPE, SIG_DFL);
  exit(0);
}

void DoWork(char **buffer, int size) {
  int rc;

  while (*buffer !=NULL)
    *buffer = (char *)malloc(size*ONE_K);
  if (*buffer == NULL) {
     if (raise(OUT_OF_STORAGE)) {
        perror("Could not raise user signal");
        abort();
     }
  }
  return;
}
```

Related information

- Signal-handling in *z/OS XL C/C++ Programming Guide*.
- “signal.h” on page 63
- “abort() — Stop a program” on page 101
- “atexit() — Register program termination function” on page 199
- “bsd_signal() — BSD version of signal()” on page 219
- “exit() — End program” on page 443
- “kill() — Send a signal to a process” on page 931
- “pthread_kill() — Send a signal to a thread” on page 1283
- “raise() — Raise signal” on page 1371
- “sigaction() — Examine or change a signal action” on page 1611
- “sigignore() — Set disposition to ignore a signal” on page 1637
- “sigprocmask() — Examine or change a thread” on page 1651
- “sigset() — Change a signal action or a thread” on page 1656
- “sigsuspend() — Change mask and suspend the thread” on page 1663
- “waitid() — Wait for child process to change state” on page 1995
- “wait3() — Wait for child process to change state” on page 2000

signbit() — Determines whether the sign of its argument is negative

Standards

Standards / Extensions	C or C++	Dependencies
C99 Single UNIX Specification, Version 3	both	z/OS V1R8

Format

```
#define _ISOC99_SOURCE
#include <math.h>

int signbit(real-floating x);

#define __STDC_WANT_DEC_FP__
#include <math.h>

int signbit(real-floating x);  /* C only */
int signbit(decimal-floating x);  /* C only */
bool signbit(real-floating x);  /* C++ only */
bool signbit(decimal-floating x);  /* C++ only */


#define _TR1_C99
```

```
#include <math.h>

bool signbit(real-floating x);  /* C++ only */
```

General description

The signbit() macro or function template determines whether the sign of its argument value is negative.

Function	Hex	IEEE
signbit	X	X

Notes:

1. To use IEEE decimal floating-point, the hardware must have the Decimal Floating-Point Facility installed.
2. This function works in IEEE decimal floating-point format. See "IEEE Decimal Floating-Point" for more information.

Returned value

The signbit() macro returns 1 if the sign of its argument value is negative, else returns 0. The C++ function template returns true if the sign of its argument value is negative, else returns false.

Related information

- “math.h” on page 44

__signgam() — Return signgam reference

Standards

Standards / Extensions	C or C++	Dependencies
z/OS UNIX	both	

Format

```
#define _XOPEN_SOURCE
#include <math.h>

int *__signgam(void);
#define signgam (*__signgam())
```

General description

The __signgam() function returns the address of the calling thread's storage for the *signgam* external variable used by the gamma() and lgamma() functions. This extended mechanism is necessary for multithreaded processes which use either of these two functions, since each thread has its own instance of *signgam*. The <math.h> header defines *signgam* to an invocation of __signgam(), so generally, all references to *signgam* will be mapped to calls to __signgam(). If the user eliminates this definition, either by not including the header, or by using #undef, then references to *signgam* will refer to the actual *signgam* external variable, which contains the *signgam* value for the IPT only. In the absence of the definition of *signgam* to a call to __signgam(), *signgam* values in threads other than the IPT are inaccessible.

Returned value

__signgam() is always successful.

Related information

- “math.h” on page 44
- “gamma() — Calculate gamma function” on page 682
- “lgamma(), lgammaf(), lgammal() — Log gamma function” on page 964

sigpause() — Unblock a signal and wait for a signal

Standards

Standards / Extensions	C or C++	Dependencies
XPG4.2 Single UNIX Specification, Version 3	both	POSIX(ON)

Format

```
#define _XOPEN_SOURCE_EXTENDED 1
#include <signal.h>

int sigpause(int sig);
```

General description

The sigpause() function provides a simplified method for removing a signal, specified by the argument *sig*, from the calling thread's signal mask and suspending this thread until a signal is received whose action is either to execute a signal catcher function or to terminate the process.

Usage notes

The use of the `SIGTHSTOP` and `SIGTHCONT` signal is not supported with this function.

Returned value

If successful, sigpause() returns -1 and sets errno to EINTR.

If unsuccessful, sigpause() returns -1 and sets errno to one of the following values:

Error Code
Description

EINVAL
The value of the argument *sig* is not a valid signal type or it is SIGKILL, or SIGTRACE.

Related information

- “signal.h” on page 63
- “pause() — Suspend a process pending a signal” on page 1173
- “sigprocmask() — Examine or change a thread” on page 1651
- “sigsuspend() — Change mask and suspend the thread” on page 1663

sigpending() — Examine pending signals

Standards

Standards / Extensions	C or C++	Dependencies
POSIX.1 XPG4 XPG4.2 Single UNIX Specification, Version 3	both	

Format

```
#define _POSIX_SOURCE
#include <signal.h>

int sigpending(sigset_t *set);
```

General description

Returns the union of the set of signals that are blocked from delivery and pending for the calling thread and the set that are pending for the process. If there is only one thread, it does the same for the calling process. This information is represented as a signal set stored in *set*. For more information on examining the signal set pointed to by *set*, see “sigismember() — Test if a signal is in a signal mask” on page 1638.

Usage notes

The use of the SIGTHSTOP and SIGTHCONT signal is not supported with this function.

Returned value

If successful, sigpending() returns 0.

If unsuccessful, sigpending() returns -1.

There are no documented errno values.

Example

CELEBS22

```
/* CELEBS22

   This example returns blocked or pending signals.

 */
#define _POSIX_SOURCE
#include <signal.h>
#include <stdio.h>
#include <unistd.h>

void catcher(int signum) {
  puts("inside catcher!");
}

void check_pending(int signum, char *signame) {
  sigset_t sigset;

  if (sigpending(&sigset) != 0)
```

```
    perror("sigpending() error");
  else if (sigismember(&sigset, signum))
         printf("a %s signal is pending\n", signame);
       else
         printf("no %s signals are pending\n", signame);
}

main() {
  struct sigaction sigact;
  sigset_t sigset;

  sigemptyset(&sigact.sa_mask);
  sigact.sa_flags = 0;
  sigact.sa_handler = catcher;
  if (sigaction(SIGUSR1, &sigact, NULL) != 0)
    perror("sigaction() error");
  else {
    sigemptyset(&sigset);
    sigaddset(&sigset, SIGUSR1);
    if (sigprocmask(SIG_SETMASK, &sigset, NULL) != 0)
      perror("sigprocmask() error");
    else {
      puts("SIGUSR1 signals are now blocked");
      kill(getpid(), SIGUSR1);
      printf("after kill: ");
      check_pending(SIGUSR1, "SIGUSR1");
      sigemptyset(&sigset);
      sigprocmask(SIG_SETMASK, &sigset, NULL);
      puts("SIGUSR1 signals are no longer blocked");
      check_pending(SIGUSR1, "SIGUSR1");
    }
  }
}
```

Output

```
SIGUSR1 signals are now blocked
after kill: a SIGUSR1 signal is pending
inside catcher!
SIGUSR1 signals are no longer blocked
no SIGUSR1 signals are pending
```

Related information

- “signal.h” on page 63
- “sigismember() — Test if a signal is in a signal mask” on page 1638
- “sigprocmask() — Examine or change a thread”

sigprocmask() — Examine or change a thread

Standards

Standards / Extensions	C or C++	Dependencies
POSIX.1 XPG4 XPG4.2 Single UNIX Specification, Version 3	both	

Format

```
#define _POSIX_SOURCE
#include <signal.h>

int sigprocmask(int option, const sigset_t *__restrict__ new_set,
                sigset_t *__restrict__ old_set);
```

General description

Examines, changes, or examines and changes the signal mask of the calling thread. If there is only one thread, it does the same for the calling process.

Typically, `sigprocmask(SIG_BLOCK, ..., ...)` is used to block signals during a critical section of code. At the end of the critical section of code, `sigprocmask(SIG_SETMASK, ..., ...)` is used to restore the mask to the previous value returned by `sigprocmask(SIG_BLOCK, ..., ...)`.

option Indicates the way in which the existing set of blocked signals should be changed. The following are the possible values for *option*, defined in the `signal.h` header file:

SIG_BLOCK
Indicates that the set of signals given by *new_set* should be blocked, in addition to the set currently being blocked.

SIG_UNBLOCK
Indicates that the set of signals given by *new_set* should not be blocked. These signals are removed from the current set of signals being blocked.

SIG_SETMASK
Indicates that the set of signals given by *new_set* should replace the old set of signals being blocked.

new_set
Points to a signal set giving the new signals that should be blocked or unblocked (depending on the value of *option*) or it points to the new signal mask if the option was *sig_setmask*. Signal sets are described in "sigemptyset() — Initialize a signal mask to exclude all signals" on page 1633. If *new_set* is a NULL pointer, the set of blocked signals is not changed. sigprocmask() determines the current set and returns this information in **old_set*. If *new_set* is NULL, the value of *option* is not significant.

The signal set manipulation functions: sigemptyset(), sigfillset(), sigaddset(), and sigdelset() must be used to establish the new signal set pointed to by *new_set*.

old_set Points to a memory location where sigprocmask() can store a signal set. If *new_set* is NULL, *old_set* returns the current set of signals being blocked. When *new_set* is not NULL, the set of signals pointed to by *old_set* is the previous set.

If there are any pending unblocked signals, either at the process level or at the current thread's level after sigprocmask() has changed the signal mask, then at least one of those signals is delivered to the thread before sigprocmask() returns.

The signals SIGKILL, SIGSTOP, or SIGTRACE cannot be blocked. If you attempt to use sigprocmask() to block these signals, the attempt is simply ignored. sigprocmask() does not return an error status.

SIGFPE, SIGILL, and SIGSEGV signals that are not artificially generated by kill(), killpg(), raise(), sigqueue(), or pthread_kill() (that is, were generated by the system as a result of a hardware or software exception) will not be blocked.

If an artificially raised SIGFPE, SIGILL, or SIGSEGV signal is pending and blocked when an exception causes another SIGFPE, SIGILL, or SIGSEGV signal, both the artificial and exception-caused signals may be delivered to the application.

If sigprocmask() fails, the signal mask of the thread is not changed.

Usage notes

The use of the SIGTHSTOP and SIGTHCONT signal is not supported with this function.

Returned value

If successful, sigprocmask() returns 0.

If unsuccessful, sigprocmask() returns -1 and sets errno to one of the following values:

Error Code
    Description

EINVAL
    option does not have one of the recognized values.

Example

CELEBS23

```
/* CELEBS23

   This example changes the signal mask.

 */
#define _POSIX_SOURCE
#include <signal.h>
#include <stdio.h>
#include <time.h>
#include <unistd.h>

void catcher(int signum) {
  puts("inside catcher");
}

main() {
  time_t start, finish;
  struct sigaction sact;
  sigset_t new_set, old_set;
  double diff;

  sigemptyset(&sact.sa_mask);
  sact.sa_flags = 0;
  sact.sa_handler = catcher;
  if (sigaction(SIGALRM, &sact, NULL) != 0)
    perror("sigaction() error");
  else {
    sigemptyset(&new_set);
    sigaddset(&new_set, SIGALRM);
    if (sigprocmask(SIG_BLOCK, &new_set, &old_set) != 0)
      perror("1st sigprocmask() error");
    else {
      time(&start);
      printf("SIGALRM signals blocked at %s", ctime(&start));
      alarm(1);

      do {
```

```
    time(&finish);
    diff = difftime(finish, start);
   } while (diff < 10);
   if (sigprocmask(SIG_SETMASK, &old_set, NULL) != 0)
    perror("2nd sigprocmask() error");
   else
    printf("SIGALRM signals unblocked at %s", ctime(&finish));
  }
 }
}
```

Output

```
SIGALRM signals blocked at Fri Jun 16 12:24:19 2006
inside catcher
SIGALRM signals unblocked at Fri Jun 16 12:24:29 2006
```

Related information

- "signal.h" on page 63
- "kill() — Send a signal to a process" on page 931
- "killpg() — Send a signal to a process group" on page 934
- "pthread_kill() — Send a signal to a thread" on page 1283
- "raise() — Raise signal" on page 1371
- "sigaction() — Examine or change a signal action" on page 1611
- "sigaddset() — Add a signal to the signal mask" on page 1628
- "sigdelset() — Delete a signal from the signal mask" on page 1632
- "sigemptyset() — Initialize a signal mask to exclude all signals" on page 1633
- "sigfillset() — Initialize a signal mask to include all signals" on page 1635
- "sighold() — Add a signal to a thread" on page 1636
- "sigismember() — Test if a signal is in a signal mask" on page 1638
- "signal() — Handle interrupts" on page 1643
- "sigpending() — Examine pending signals" on page 1650
- "sigpause() — Unblock a signal and wait for a signal" on page 1649
- "sigrelse() — Remove a signal from a thread" on page 1656
- "sigset() — Change a signal action or a thread" on page 1656
- "sigsuspend() — Change mask and suspend the thread" on page 1663

sigqueue() — Queue a signal to a process

Standards

Standards / Extensions	C or C++	Dependencies
Single UNIX Specification, Version 2 Single UNIX Specification, Version 3	both	OS/390 V2R9

Format

```
#define _XOPEN_SOURCE 500
#include <sys/types.h>
#include <signal.h>

int sigqueue(pid_t pid, int signo, const union sigval value);
```

General description

Causes the signal specified by *signo* to be sent with the value specified by *value* to the process specified by *pid*. If *signo* is zero (the null signal), error checking is performed but no signal is actually sent. The null signal can be used to check the validity of *pid*. The conditions required for a process to have permission to queue a signal to another process are the same as for the kill() function.

The sigqueue() function returns immediately. If the resources were available to queue the signal, the signal is queued and sent to the receiving process. The fact that SA_SIGINFO is not set for *signo* does not effect this processing and queueing of the signal.

If the value of *pid* causes *signo* to be generated for the sending process, and if *signo* is not blocked for the calling thread and if no other thread has *signo* unblocked or is waiting in a sigwait() function for signo, either signo or at least the pending, unblocked signal will be delivered to the calling thread before sigqueue() returns.

Usage notes

The use of the `SIGTHSTOP` and `SIGTHCONT` signal is not supported with this function.

Since in AMODE 64 programs, sigval is 64 bits long, and in AMODE 31 programs sigval is only 32 bits long, when passing sigval data between an AMODE 31 and AMODE 64 process, there are the following restrictions:

- In AMODE 64, the sival_int field covers only the first 4 bytes of the sigval field -- only the sival_ptr field can access all 8 bytes of the sigval field.
- When an AMODE 64 program passes a sival_ptr value to an AMODE 31 program, the AMODE 31 program receives only the low 32 bits of the original sival_ptr.
- When an AMODE 31 program passes a sival_ptr value to an AMODE 64 program, the original sival_ptr value is received in the low 32 bits of the AMODE 64 sival_ptr.
- When an AMODE 64 program tries to pass a value to an AMODE 31 program using the sival_int field, the AMODE 31 program will receive 0 in sigval.
- When an AMODE 31 program sends a value to an AMODE 64 program using sival_int, the AMODE 64 program will receive a 0 value in sival_int, but it can access the original value as the low 32 bits of the AMODE 64 sival_ptr field.

Returned value

If successful, sigqueue() returns 0.

If unsuccessful, sigqueue() returns -1 and sets errno to one of the following values:

Error Code
Description

EAGAIN
No resources available to queue the signal or the system-wide resource limit, defined by MAXQUEUEDSIGS, has been exceeded.

EINVAL
The value of the *signo* argument is an invalid or unsupported signal number.

EPERM
The process does not have the appropriate privilege to send the signal to the receiving process.

ESRCH
The process *pid* does not exist.

Related information

- "signal.h" on page 63
- "sys/types.h" on page 75
- "kill() — Send a signal to a process" on page 931

sigrelse() — Remove a signal from a thread

Standards

Standards / Extensions	C or C++	Dependencies
XPG4.2 Single UNIX Specification, Version 3	both	

Format

```
#define _XOPEN_SOURCE_EXTENDED 1
#include <signal.h>

int sigrelse(int sig);
```

General description

The sigrelse() function provides a simplified method for removing the signal specified by the argument *sig* from the calling thread's signal mask.

Usage notes

The use of the `SIGTHSTOP` and `SIGTHCONT` signal is not supported with this function.

Returned value

If successful, sigrelse() returns 0.

If unsuccessful, sigrelse() returns -1 and sets errno to one of the following values:

Error Code
    Description

EINVAL
    The value of the argument *sig* is not a valid signal type or it is SIGKILL, SIGSTOP, or SIGTRACE.

Related information

- "signal.h" on page 63
- "sighold() — Add a signal to a thread" on page 1636
- "sigpause() — Unblock a signal and wait for a signal" on page 1649
- "sigprocmask() — Examine or change a thread" on page 1651

sigset() — Change a signal action or a thread

Standards

Standards / Extensions	C or C++	Dependencies
XPG4.2 Single UNIX Specification, Version 3	both	POSIX(ON)

Format

```
#define _XOPEN_SOURCE_EXTENDED 1
#include <signal.h>

void (*sigset(int sig, void (*disp)(int)))(int);
```

General description

The sigset() function provides a simplified method for changing the action associated with a specific signal and unblock the signal, or to block this signal.

sig The number of a recognized signal. sigset() sets the action associated with this signal and unblock this signal, or adds this signal to the calling thread's signal mask (thus blocking this signal). Refer to Table 48 on page 1612 for a list of the supported values of *sig*.

The value of *sig* can be any valid signal type except SIGKILL, SIGSTOP, or SIGTRACE.

disp There are four possible value that *disp* can have. Three are actions that can be associated with the signal, *sig*: SIG_DFL, SIG_IGN, or a pointer to a function. The fourth value is not a signal action, but a flag to sigset() that affects whether the signal action is changed.

The values that *disp* is permitted to have are:

SIG_DFL

Set the signal action to the signal-specific default.

- The default actions for each signal is shown in Table 48 on page 1612.
- If *disp* is set to SIG_DFL, sigset() will change the signal action associated with *sig* and remove this signal from the calling thread's signal mask (thus unblocking this signal).
- If the default action is to stop the process, the execution of that process is temporarily suspended. When a process stops, a SIGCHLD signal will be generated for its parent process, unless the parent process has set the **SA_NOCLDSTOP** flag. While a process is stopped, any additional signals that are sent to the process will not be delivered until the process is continued, except SIGKILL which always terminates the receiving process. A process that is a member of an orphaned process group will not be allowed to stop in response to the SIGTSTP, SIGTTIN, or SIGTTOU signals. In cases where delivery of one of these signals would stop such a process, the signal will be discarded.
- Setting a signal action to SIG_DFL for a signal that is pending, and whose default action is to ignore the signal (for example SIGCHLD), will cause the pending signal to be discarded.

SIG_IGN

Set the signal action to ignore the signal.

- Delivery of the signal will have no effect on the process.
- If *disp* is set to SIG_IGN, sigset() will change the signal action associated with *sig* and remove this signal from the calling thread's signal mask (thus unblocking this signal).
- Setting a signal action to SIG_IGN for a signal that is pending will cause the pending signal to be discarded. This provides the ability to discard signals that are found to be blocked and pending by sigpending().
- If *sig* is SIGCHLD, child processes of the calling process will not be transformed into 'zombie' processes when they terminate. If the calling

process subsequently waits for its children, and the process has no unwaited from children that were transformed into 'zombie' processes, it will block until all of its children terminate. The wait(), waitid(), or waitpid() function will fail and set errno to **ECHILD**.

SIG_HOLD
Set the calling thread's signal mask to block signal, *sig*.

- The signal action associated with *sig* is not changed.

Pointer to function
Set the signal action to catch the signal.

- sigset() will change the signal action associated with *sig* and remove this signal from the calling thread's signal mask (thus unblocking this signal).
- On delivery of the signal, the receiving process is to execute the signal-catching function at the specified address. After returning from the signal-catching function, the receiving process will resume execution at the point at which it was interrupted.
- The signal-catching function specified by *disp* is invoked as:

  void *function*(int *signo*);

  Where *function* is the specified signal-catching function and *signo* is the signal number of the signal being delivered.

After an action has been specified for a particular signal, using sigset(), it remains installed until it is explicitly changed with another call to sigset(), sigaction(), signal(), one of the exec functions, bsd_signal(), or sigignore().

Special behavior for C++:

- The behavior when mixing signal-handling with C++ exception handling is undefined. Also, the use of signal-handling with constructors and destructors is undefined.
- C++ and C language linkage conventions are incompatible, and therefore sigaction() cannot receive C++ function pointers. If you attempt to pass a C++ function pointer to sigaction(), the compiler will flag it as an error. Therefore, to use the sigaction() function in the C++ language, you must ensure that signal handler routines established have C linkage, by declaring them as `extern "C"`.

Usage notes

The use of the `SIGTHSTOP` and `SIGTHCONT` signal is not supported with this function.

Returned value

If successful, sigset() returns SIG_HOLD if the signal had been blocked and the signal's previous action if it had not been blocked.

If unsuccessful, sigset() returns SIG_ERR and sets errno to one of the following values:

Error Code
Description

EINVAL
The value of the argument *sig* was not a valid signal type, or it was SIGKILL, SIGSTOP, or SIGTRACE.

Related information

- "signal.h" on page 63
- "bsd_signal() — BSD version of signal()" on page 219
- "sigaction() — Examine or change a signal action" on page 1611
- "sighold() — Add a signal to a thread" on page 1636
- "signal() — Handle interrupts" on page 1643
- "sigprocmask() — Examine or change a thread" on page 1651

sigsetjmp() — Save stack environment and signal mask

Standards

Standards / Extensions	C or C++	Dependencies
POSIX.1 XPG4 XPG4.2 Single UNIX Specification, Version 3	both	

Format

```
#define _POSIX_SOURCE
#include <setjmp.h>

int sigsetjmp(sigjmp_buf env, int savemask);
```

General description

Saves the current stack environment including, optionally, the current signal mask. The stack environment and signal mask saved by sigsetjmp() can subsequently be restored by siglongjmp().

env is an address for a `sigjmp_buf` structure. *savemask* is a flag used to determine if the signal mask is to be saved. If it has a value of 0, the current signal mask is not to be saved or restored as part of the environment. Any other value means the current signal mask is saved and restored.

sigsetjmp() is similar to setjmp() and _setjmp(), except for the optional capability of saving the signal mask. Like setjmp() and longjmp(), the sigsetjmp() and siglongjmp() functions provide a way to perform a nonlocal `goto`.

The sigsetjmp()—siglongjmp() pair, the setjmp()—longjmp() pair, the _setjmp()—_longjmp() pair and the getcontext()—setcontext() pair cannot be intermixed. A stack environment and signal mask saved by sigsetjmp() can be restored only by siglongjmp().

A call to sigsetjmp() causes it to save the current stack environment in *env*. If the value of the *savemask* parameter is nonzero, it will also save the current signal mask in *env*. A subsequent call to siglongjmp() restores the saved environment and signal mask (if saved by sigsetjmp()), and returns control to a point corresponding to the sigsetjmp() call. The values of all variables (except register variables) accessible to the function receiving control contain the values they had when siglongjmp() was called. The values of register variables are unpredictable. Nonvolatile auto variables that are changed between calls to sigsetjmp() and siglongjmp() are also unpredictable.

Note: Ensure that the function that calls sigsetjmp() does not return before you call the corresponding siglongjmp() function. Calling siglongjmp() after the function calling sigsetjmp() returns causes unpredictable program behavior.

Special behavior for C++: If sigsetjmp() and siglongjmp() are used to transfer control in a z/OS XL C++ program, the behavior in terms of the destruction of automatic objects is undefined. This applies to both z/OS XL C++ and z/OS XL C/C++ ILC modules. The use of sigsetjmp() and siglongjmp() in conjunction with try(), catch(), and throw() is also undefined.

Special behavior for XPLINK-compiled C++: Restrictions concerning setjmp.h and ucontext.h:

1. All XPLINK programs compiled with the V2R10 or later C compilers that are to run with Language Environment V2R10 or later libraries and use the **jmp_buf**, **sigjmp_buf** or **ucontext_t** types must not be compiled with C headers from Language Environment V2R9 or earlier.
2. Non-XPLINK functions compiled with any level of Language Environment headers must not define **jmp_buf**, **sigjmp_buf** or **ucontext_t** data items and pass them to XPLINK functions that call getcontext(), longjmp(), _longjmp(), setjmp(), _setjmp(), setcontext(), sigsetjmp(), or swapcontext() with these passed-in data items.
3. When __XPLINK__ is defined, the Language Environment V2R10 and later headers define a larger **jmp_buf**, **sigjmp_buf** or **ucontext_t** area that is required by setjmp(), getcontext(), and related functions when they are called from an XPLINK routine. If __XPLINK__ is not defined, the Language Environment V2R10 and later headers define a shorter **jmp_buf**, **sigjmp_buf** or **ucontext_t** area. The Language Environment headers before V2R10 also define the shorter version of these data areas. If an XPLINK function calls setjmp(), getcontext() or similar functions with a short **jmp_buf**, **sigjmp_buf** or **ucontext_t** area, a storage overlay or program check may occur when the C library tries to store past the end of the passed-in (too short) data area.

Returned value

sigsetjmp() returns 0 when it is invoked to save the stack environment and signal mask.

sigsetjmp() returns the value *val*, specified on siglongjmp() (or 1 if the value of *val* is zero), when siglongjmp() causes control to be transferred to the place in the user's program where sigsetjmp() was issued.

There are no documented errno values.

Example

The following saves the stack environment and signal mask at the statement:

```
if(sigsetjmp(mark,1) != 0) ...
```

When the system first performs the `if` statement, it saves the environment and signal mask in *mark* and sets the condition to false because sigsetjmp() returns 0 when it saves the environment. The program prints the message:

```
sigsetjmp() has been called
```

The subsequent call to function p() tests for a local error condition, which can cause it to perform siglongjmp(). Then control returns to the original sigsetjmp()

function using the environment saved in *mark* and the restored signal mask. This time the condition is true because -1 is the return value from siglongjmp(). The program then performs the statements in the block and prints:

```
siglongjmp() has been called
```

Then the program performs the sample recover() function and exits.

```
#define _POSIX_SOURCE
#include <stdio.h>
#include <setjmp.h>

sigjmp_buf mark;

void p(void);
void recover(void);

int main(void)
{
   if (sigsetjmp(mark,1) != 0) {
      printf("siglongjmp() has been called\n");
      recover();
      exit(1);
      }
   printf("sigsetjmp() has been called\n");
⋮
   p();
⋮
}

void p(void)
{
   int error = 0;
⋮
   error = 9;
⋮
   if (error != 0)
      siglongjmp(mark, -1);
⋮
}

void recover(void)
{
⋮
}
```

Related information

- "setjmp.h" on page 62
- "getcontext() — Get user context" on page 694
- "longjmp() — Restore stack environment" on page 1005
- "_longjmp() — Nonlocal goto" on page 1008
- "setcontext() — Restore user context" on page 1524
- "setjmp() — Preserve stack environment" on page 1546
- "_setjmp() — Set jump point for a nonlocal goto" on page 1549
- "sigaction() — Examine or change a signal action" on page 1611
- "siglongjmp() — Restore the stack environment and signal mask" on page 1640
- "sigprocmask() — Examine or change a thread" on page 1651
- "sigsuspend() — Change mask and suspend the thread" on page 1663
- "swapcontext() — Save and restore user context" on page 1792

sigstack() — Set or get signal stack context

Standards

Standards / Extensions	C or C++	Dependencies
XPG4.2	both	POSIX(ON)

Format

```
#define _XOPEN_SOURCE_EXTENDED 1
#include <signal.h>

int sigstack(struct sigstack *ss, struct sigstack *oss);
```

General description

The sigstack() function allows the calling thread to indicate, to the system, an area of its address space to be used for processing signals received by this thread.

Note: To explicitly declare that a signal catcher is to run on the alternate signal stack, the SA_ONSTACK flag must be set in the sa_flags when the signal action is set using sigaction().

If the *ss* argument is not a NULL pointer, it must point to a `sigstack` structure. The length of the application-supplied stack must be at least SIGSTKSZ bytes. If the alternate signal stack overflows, the resulting behavior is undefined.

- The value of the `ss_onstack` member indicates whether the thread wants the system to use an alternate signal stack when delivering signals.
- The value of the `ss_sp` member indicates the desired location of the alternate signal stack area in the process's address space.

  AMODE 64: Storage for this stack must be above the 2GB bar. It may not be storage acquired with the __malloc24() or __malloc31() functions.
- If the *ss* argument is a NULL pointer, the current alternate signal stack context is not changed.

If the *oss* argument is not a NULL pointer, it must point to a `sigstack` structure into which the current alternate signal stack context is placed. The value stored in the `ss_onstack` member of this `sigstack` structure will be nonzero if the thread is currently executing on the alternate signal stack. If the *oss* argument is a NULL pointer, the current alternate signal stack context is not returned.

When a signal's action indicates its handler should execute on the alternate signal stack (specified by calling sigaction()), the implementation checks to see if the thread is currently executing on the alternate signal stack. If it is not, the system will switch to the alternate signal stack for the duration of the signal handler's execution.

After a successful call to one of the exec functions, there are no alternate signal stacks in the new process image.

Notes:

1. If a signal handler is enabled to run on an alternate stack, then all functions called by that signal handler must be compiled with the same linkage. For example, if the signal handler is compiled with XPLINK, then all functions it

calls must also be compiled XPLINK. Since only one alternate stack can be supplied, no mixing of linkages (which would require both upward and downward-growing alternate stacks) is allowed. The type of stack created will be based on the attributes of the signal handler to be given control. If the signal handler has been compiled with XPLINK, then a downward-growing stack will be created in the alternate stack, including, in AMODE 31, using enough storage in the user stack to create a 4k read-only guard page (aligned on a 4k boundary).

2. If a new signal is received while a signal handler is running on an alternate stack, and that new signal specified a signal handler that also runs on the alternate stack, then both signal handlers must have been compiled with the same linkage (XPLINK versus non-XPLINK).
3.

   This function is kept for historical reasons. It was part of the Legacy Feature in Single UNIX Specification, Version 2, but has been withdrawn and is not supported as part of Single UNIX Specification, Version 3. New applications should use sigaltstack() instead of sigstack().

   If it is necessary to continue using this function in an application written for Single UNIX Specification, Version 3, define the feature test macro _UNIX03_WITHDRAWN before including any standard system headers. The macro exposes all interfaces and symbols removed in Single UNIX Specification, Version 3.

Returned value

If successful, sigstack() returns 0.

If unsuccessful, sigstack() returns -1 and sets errno to one of the following values:

Error Code
Description

EPERM
An attempt was made to modify an active stack.

Related information

- “signal.h” on page 63
- “sigaction() — Examine or change a signal action” on page 1611
- “sigaltstack() — Set or get signal alternate stack context” on page 1630
- “sigsetjmp() — Save stack environment and signal mask” on page 1659

sigsuspend() — Change mask and suspend the thread

Standards

Standards / Extensions	C or C++	Dependencies
POSIX.1 XPG4 XPG4.2 Single UNIX Specification, Version 3	both	POSIX(ON)

Format

```
#define _POSIX_SOURCE
#include <signal.h>

int sigsuspend(const sigset_t *mask);
```

General description

Replaces the current signal mask of a thread with the signal set given by **mask* and then suspends execution of the calling thread. The thread does not resume running until a signal is delivered whose action is either to execute a signal-handling function or to end the process. (Signal sets are described in more detail in “sigemptyset() — Initialize a signal mask to exclude all signals” on page 1633.)

The signal mask indicates a set of signals that should be blocked. Such signals do not “wake up” the suspended function. The signals SIGKILL, SIGSTOP, or SIGTRACE cannot be blocked or ignored; they are delivered to the thread no matter what the *mask* argument specifies.

If an incoming unblocked signal ends the thread, sigsuspend() never returns to the caller. If an incoming signal is handled by a signal-handling function, sigsuspend() returns after the signal-handling function returns. The signal mask of the thread is restored to whatever it was before sigsuspend() was called, unless the signal-handling functions explicitly changed the mask.

This function is supported only in a POSIX program.

Usage notes

The use of the `SIGTHSTOP` and `SIGTHCONT` signal is not supported with this function.

Returned value

If sigsuspend() returns, it always returns -1.

If unsuccessful, sigsuspend() sets errno to one of the following values:

Error Code
Description

EINTR
A signal was received and handled successfully.

Example

CELEBS25

```
/* CELEBS25

   This example replaces the signal mask and then suspends execution.

 */
#define _POSIX_SOURCE
#include <stdio.h>
#include <signal.h>
#include <time.h>
#include <unistd.h>

void catcher(int signum) {
  switch (signum) {
    case SIGUSR1: puts("catcher caught SIGUSR1");
                  break;
    case SIGUSR2: puts("catcher caught SIGUSR2");
                  break;
    default:      printf("catcher caught unexpected signal %d\n",
                         signum);
  }
}

main() {
  sigset_t sigset;
  struct sigaction sact;
  time_t   t;

  if (fork() == 0) {
    sleep(10);
    puts("child is sending SIGUSR2 signal - which should be blocked");
    kill(getppid(), SIGUSR2);
    sleep(5);
    puts("child is sending SIGUSR1 signal - which should be caught");
    kill(getppid(), SIGUSR1);
    exit(0);
  }

  sigemptyset(&sact.sa_mask);
  sact.sa_flags = 0;
  sact.sa_handler = catcher;
  if (sigaction(SIGUSR1, &sact, NULL) != 0)
    perror("1st sigaction() error");

  else if (sigaction(SIGUSR2, &sact, NULL) != 0)
    perror("2nd sigaction() error");

  else {
    sigfillset(&sigset);
    sigdelset(&sigset, SIGUSR1);
    time(&t);
    printf("parent is waiting for child to send SIGUSR1 at %s",
         ctime(&t));
    if (sigsuspend(&sigset) == -1)
      perror("sigsuspend() returned -1 as expected");
    time(&t);
    printf("sigsuspend is over at %s", ctime(&t));
  }
}
```

Output

```
parent is waiting for child to send SIGUSR1 at Fri Jun 16 12:30:57 2006
child is sending SIGUSR2 signal - which should be blocked
child is sending SIGUSR1 signal - which should be caught
catcher caught SIGUSR2
catcher caught SIGUSR1
sigsuspend() returned -1 as expected: Interrupted function call
sigsuspend is over at Fri Jun 16 12:31:12 2006
```

Related information

- “signal.h” on page 63
- “bsd_signal() — BSD version of signal()” on page 219
- “kill() — Send a signal to a process” on page 931
- “killpg() — Send a signal to a process group” on page 934
- “pause() — Suspend a process pending a signal” on page 1173
- “pthread_kill() — Send a signal to a thread” on page 1283
- “raise() — Raise signal” on page 1371
- “sigaction() — Examine or change a signal action” on page 1611

- "sigaddset() — Add a signal to the signal mask" on page 1628
- "sigdelset() — Delete a signal from the signal mask" on page 1632
- "sigemptyset() — Initialize a signal mask to exclude all signals" on page 1633
- "sigfillset() — Initialize a signal mask to include all signals" on page 1635
- "sigignore() — Set disposition to ignore a signal" on page 1637
- "signal() — Handle interrupts" on page 1643
- "sigpause() — Unblock a signal and wait for a signal" on page 1649
- "sigpending() — Examine pending signals" on page 1650
- "sigprocmask() — Examine or change a thread" on page 1651
- "sigset() — Change a signal action or a thread" on page 1656

sigtimedwait() — Wait for queued signals

Standards

Standards / Extensions	C or C++	Dependencies
Single UNIX Specification, Version 2 Single UNIX Specification, Version 3	both	OS/390 V2R7

Format

```
#define _XOPEN_SOURCE 500
#include <signal.h>

int sigtimedwait(const sigset_t *set, siginfo_t *info
                 const struct timespec *timeout);
```

General description

The sigtimedwait() function selects a pending signal from the `sigset_t` object (signal set) pointed to by *set*, automatically clearing it from the system's set of pending signals, and returning that signal number. If there are multiple pending signals, the lowest numbered signal will be selected.

If no signal in the signal set is pending at the time of the call to sigtimedwait(), the thread is suspended until one or more of the signals specified in the signal set become pending or until it is interrupted by an unblocked, caught signal. The signals defined in the `sigset_t` object (signal set) pointed to by *set* may be unblocked during the call to this routine and will be blocked when the thread returns from the call unless some other thread is currently waiting for one of those signals.

If more than one thread is using sigtimedwait() to wait for the same signal, only one of these threads will return from this routine with the signal number, until a second signal of the same type is received.

The function sigtimedwait() behaves the same as the sigwait() function if the *info* argument is NULL. If the *info* argument is not NULL, then in addition to behaving the same as sigwait(), sigtimedwait() places the selected signal number in the `si_signo` member, places the cause of the signal in the `si_code` member, and, if any value is queued to the selected signal, sigtimedwait() will place it in the `si_value` member of *info*. However, if there is no value queued for the selected signal then the content of `si_value` is undefined.

If the sigtimedwait() function finds that none of the signals specified by *set* are pending, it waits for the time interval specified in the `timespec` structure referenced by *timeout* If the `timespec` structure pointed to by *timeout* is zero-valued and if none of the signals specified by *set* are pending, then sigtimedwait() returns immediately with an error. A `timespec` with the tv_sec field set with *INT_MAX*, as defined in <limits.h>, will cause the sigtimedwait() service to wait until a signal is received. If *timeout* is the NULL pointer, the behavior is not necessarily the same on all platforms but for this platform it will be treated the same as when `timespec` structure was supplied with with the tv_sec field set with *INT_MAX*.

Usage notes

The use of the `SIGTHSTOP` and `SIGTHCONT` signal is not supported with this function.

Returned value

If successful, sigtimedwait() returns the signal number.

If unsuccessful, sigtimedwait() returns -1 and sets errno to one of the following values:

Error Code
Description

EAGAIN
No signal specified by set was generated within the specified time out period.

EINTR
The wait was interrupted by an unblocked, caught signal. No further waiting will occur for this call. sigtimedwait() can be reissued to begin waiting again.

EINVAL
set points to a **sigset_t** that contains a signal number that is either not valid or not supported.

Related information

- "signal.h" on page 63
- "time.h" on page 79
- "pause() — Suspend a process pending a signal" on page 1173
- "sigaction() — Examine or change a signal action" on page 1611
- "sigpending() — Examine pending signals" on page 1650
- "sigprocmask() — Examine or change a thread" on page 1651
- "sigsuspend() — Change mask and suspend the thread" on page 1663
- "sigwait() — Wait for an asynchronous signal"

sigwait() — Wait for an asynchronous signal

Standards

Standards / Extensions	C or C++	Dependencies
POSIX.1 Single UNIX Specification, Version 3	both	

Format

```
#define _OPEN_THREADS
#include <signal.h>

int sigwait(sigset_t *set);
```

SUSV3:

```
#define _POSIX_C_SOURCE 200112L
#include <signal.h>
int sigwait(const sigset_t *__restrict__ set, int *__restrict__ sig);
```

General description

Causes a thread to wait for an asynchronous signal by choosing a pending signal from *set*, automatically clearing it from the system's set of pending signals, and returning that signal number in the return code.

If no signal in *set* is pending at the time of the call, the thread is suspended until one or more of the signals in *set* become pending. The signals defined by *set* may be unblocked during the call to this routine, and will be blocked when the thread returns from the call unless some other thread is currently waiting for one of those signals.

If more than one thread is using this routine to wait for the same signal, only one of these threads will return from this routine with the signal number.

Special behavior for SUSV3: The sigwait() function selects a pending signal from set, atomically clear it from the system's set of pending signals, and return that signal number in the location referenced by sig.

Argument	Description
sig	location reference where the signal number is stored

Usage notes

The use of the SIGTHSTOP and SIGTHCONT signal is not supported with this function.

Returned value

If successful, sigwait() returns the signal number.

If unsuccessful, sigwait() returns -1 and sets errno to one of the following values:

Error Code
Description

EINVAL
The *set* argument contains an invalid or unsupported signal number.

Special behavior for SUSV3: Upon successful completion, sigwait() stores the signal number of the received signal at the location referenced by sig and return zero. Otherwise, an error number is returned to indicate the error.

Example

CELEBS26

```
/* CELEBS26 */
#define _OPEN_THREADS

#include <stdio.h>
#include <errno.h>
#include <signal.h>
#include <pthread.h>
#include <unistd.h>

void           *threadfunc(void *parm)
{
 int        threadnum;
 int        *tnum;
 sigset_t   set;

 tnum = parm;
 threadnum = *tnum;

 printf("Thread %d executing\n", threadnum);
 sigemptyset(&set);
 if(sigaddset(&set, SIGUSR1) == -1) {
    perror("Sigaddset error");
    pthread_exit((void *)1);
 }

 if(sigwait(&set) != SIGUSR1) {
    perror("Sigwait error");
    pthread_exit((void *)2);
 }

 pthread_exit((void *)0);
}

main() {
 int          status;
 int          threadparm = 1;
 pthread_t    threadid;
 int          thread_stat;


 status = pthread_create( &threadid, NULL,
                          threadfunc,
                          (void *)&threadparm);
 if ( status <  0) {
    perror("pthread_create failed");
    exit(1);
 }

 sleep(5);

 status = pthread_kill( threadid, SIGUSR1);
 if ( status <  0)
    perror("pthread_kill failed");

 status = pthread_join( threadid, (void *)&thread_stat);
 if ( status <  0)
    perror("pthread_join failed");

 exit(0);
}
```

CELEBP73

```
/* CELEBS73

   This example demonstrates the use of the sigwait() function.
   The program will wait until a SIGINT signal is received from the
   command line.

   Expected output:
   SIGINT was received

*/

#define _POSIX_C_SOURCE 200112L
#include <signal.h>
#include <stdio.h>
#include <errno.h>

void main() {
  sigset_t set;
  int sig;
  int *sigptr = &sig;
  int ret_val;
  sigemptyset(&set);
  sigaddset(&set, SIGINT);
  sigprocmask( SIG_BLOCK, &set, NULL );

  printf("Waiting for a SIGINT signal\n");

  ret_val = sigwait(&set,sigptr);
  if(ret_val == -1)
     perror("sigwait failed\n");
  else {
     if(*sigptr == 2)
        printf("SIGINT was received\n");
     else
        printf("sigwait returned with sig: %d\n", *sigptr);
  }
}
```

Related information

- “signal.h” on page 63
- “bsd_signal() — BSD version of signal()” on page 219
- “pause() — Suspend a process pending a signal” on page 1173
- “sigaction() — Examine or change a signal action” on page 1611
- “sigignore() — Set disposition to ignore a signal” on page 1637
- “sigpause() — Unblock a signal and wait for a signal” on page 1649
- “sigpending() — Examine pending signals” on page 1650
- “sigprocmask() — Examine or change a thread” on page 1651
- “sigset() — Change a signal action or a thread” on page 1656
- “sigsuspend() — Change mask and suspend the thread” on page 1663

sigwaitinfo() — Wait for queued signals

Standards

Standards / Extensions	C or C++	Dependencies
Single UNIX Specification, Version 2 Single UNIX Specification, Version 3	both	OS/390 V2R7

Format

```
#define _XOPEN_SOURCE 500
#include <signal.h>

int sigwaitinfo(const sigset_t *set, siginfo_t *info);
```

General description

The sigwaitinfo() function selects a pending signal from the `sigset_t` object (signal set) pointed to by *set*, automatically clearing it from the system's set of pending signals, and returning that signal number. If there are multiple pending signals, the lowest numbered signal will be selected.

If no signal in the signal set is pending at the time of the call to sigwaitinfo(), the thread is suspended until one or more of the signals specified in the signal set become pending or until it is interrupted by an unblocked, caught signal. The signals defined in the `sigset_t` object (signal set) pointed to by *set* may be unblocked during the call to this routine and will be blocked when the thread returns from the call unless some other thread is currently waiting for one of those signals.

If more than one thread is using sigwaitinfo() to wait for the same signal, only one of these threads will return from this routine with the signal number, until a second signal of the same type is received.

The function sigwaitinfo() behaves the same as the sigwait() function if the *info* argument is NULL. If the *info* argument is not NULL, then in addition to behaving the same as sigwait(), sigwaitinfo() places the selected signal number in the `si_signo` member, places the cause of the signal in the `si_code` member, and, if any value is queued to the selected signal, sigwaitinfo() will place it in the `si_value` member of *info*. However, if there is no value queued for the selected signal then the content of *si_value* is undefined.

Usage notes

The use of the `SIGTHSTOP` and `SIGTHCONT` signal is not supported with this function.

Returned value

If successful, sigwaitinfo() returns the signal number.

If unsuccessful, sigwaitinfo() returns -1 and sets errno to one of the following values:

Error Code
Description

EINTR
The wait was interrupted by an unblocked, caught signal. No further waiting will occur for this call. sigwaitinfo() can be reissued to begin waiting again.

EINVAL
set points to a `sigset_t` that contains a signal number that is either not valid or not supported.

Related information

- "signal.h" on page 63
- "time.h" on page 79
- "pause() — Suspend a process pending a signal" on page 1173
- "sigaction() — Examine or change a signal action" on page 1611
- "sigpending() — Examine pending signals" on page 1650
- "sigsuspend() — Change mask and suspend the thread" on page 1663

- "sigwait() — Wait for an asynchronous signal" on page 1667

sin(), sinf(), sinl() — Calculate sine

Standards

Standards / Extensions	C or C++	Dependencies
ISO C POSIX.1 XPG4 XPG4.2 ISO/ANSI C++ C99 Single UNIX Specification, Version 3 C++ TR1 C99	both	

Format

```
#include <math.h>

double sin(double x);
float sin(float x);                  /* C++ only */
long double sin(long double x);      /* C++ only */
float sinf(float x);
long double sinl(long double x);
```

General description

Calculates the sine of *x*, with *x* expressed in radians.

Note: These functions work in both IEEE Binary Floating-Point and hexadecimal floating-point formats. See "IEEE binary floating-point" on page 94 for more information about IEEE Binary Floating-Point.

Returned value

If successful, the function returns the calculated value, expressed as a double, float, or long double. Otherwise, if the result is an underflow, the function returns 0 and sets the errno to ERANGE.

Special Behavior for XPG4.2: The following error is added:

Error Code
Description

EDOM
The argument exceeded an internal limit for the function (approximately 2^{50}).

Example

CELEBS27

```
/* CELEBS27

   This example computes y as the sine of &pi.&slr.2.

 */
#include <math.h>
#include <stdio.h>
```

```
int main(void)
{
   double pi, x, y;

   pi = 3.1415926535;
   x = pi/2;
   y = sin(x);

   printf("sin( %lf ) = %lf\n", x, y);
}
```

Output

```
sin( 1.570796 ) = 1.000000
```

Related information

- “math.h” on page 44
- “acos(), acosf(), acosl() — Calculate arccosine” on page 135
- “acosh(), acoshf(), acoshl() — Calculate hyperbolic arccosine” on page 139
- “asin(), asinf(), asinl() — Calculate arcsine” on page 185
- “asinh(), asinhf(), asinhl() — Calculate hyperbolic arcsine” on page 188
- “atan(), atanf(), atanl(), atan2(), atan2f(), atan2l() — Calculate arctangent” on page 192
- “atanh(), atanhf(), atanhl() — Calculate hyperbolic arctangent” on page 195
- “cos(), cosf(), cosl() — Calculate cosine” on page 328
- “cosh(), coshf(), coshl() — Calculate hyperbolic cosine” on page 331
- “sinh(), sinhf(), sinhl() — Calculate hyperbolic sine” on page 1674
- “tan(), tanf(), tanl() — Calculate tangent” on page 1815
- “tanh(), tanhf(), tanhl() — Calculate hyperbolic tangent” on page 1818

sind32(), sind64(), sind128() — Calculate sine

Standards

Standards / Extensions	C or C++	Dependencies
C/C++ DFP	both	z/OS V1.8

Format

```
#define __STDC_WANT_DEC_FP__
#include <math.h>

_Decimal32  sind32(_Decimal32 x);
_Decimal64  sind64(_Decimal64 x);
_Decimal128 sind128(_Decimal128 x);
_Decimal32  sin(_Decimal32 x);       /* C++ only */
_Decimal64  sin(_Decimal64 x);       /* C++ only */
_Decimal128 sin(_Decimal128 x);      /* C++ only */
```

General description

Calculates the sine of *x*, with *x* expressed in radians.

Notes:

1. To use IEEE decimal floating-point, the hardware must have the Decimal Floating-Point Facility installed.
2. These functions work in IEEE decimal floating-point format. See "IEEE Decimal Floating-Point" for more information.

Returned value

If successful, the function returns the calculated value, expressed as a _Decimal32, _Decimal64, or _Decimal128.

If x is outside prescribed limits, the value is not calculated. Instead, the function returns 0 and sets errno to EDOM.

Example

```
/* CELEBS69

   This example illustrates the sind32() function.

*/

#define  __STDC_WANT_DEC_FP__
#include <math.h>
#include <stdio.h>

int main(void)
{
   _Decimal32 pi, x, y;

   pi = 3.141593DF;
   x = pi/2.0DF;
   y = sind32(x);

   printf("sind32(%Hf) = %Hf\n", x, y);
}
```

Related information

- "math.h" on page 44
- "cosd32(), cosd64(), cosd128() — Calculate cosine" on page 329
- "sin(), sinf(), sinl() — Calculate sine" on page 1672

sinh(), sinhf(), sinhl() — Calculate hyperbolic sine

Standards

Standards / Extensions	C or C++	Dependencies
ISO C POSIX.1 XPG4 XPG4.2 ISO/ANSI C++ C99 Single UNIX Specification, Version 3 C++ TR1 C99	both	

Format

```
#include <math.h>

double sinh(double x);
float sinh(float x);                 /* C++ only */
long double sinh(long double x);     /* C++ only */
float sinhf(float x);
long double sinhl(long double x);
```

General description

Calculates the hyperbolic sine of *x*, with *x* expressed in radians.

Note: These functions work in both IEEE Binary Floating-Point and hexadecimal floating-point formats. See “IEEE binary floating-point” on page 94 for more information about IEEE Binary Floating-Point.

Returned value

If successful, the function returns the calculated value.

Otherwise, if the result is too large, the function sets errno to ERANGE and returns ±HUGE_VAL, depending on the value of *x*. If the value underflows, the function returns 0 and sets errno to ERANGE.

Special behavior for IEEE: If successful, the function returns the hyperbolic sine of *x* with *x* expressed in radians.

If the result would overflow, the function returns ±HUGE_VAL, according to the value of *x*, and sets errno to ERANGE. No other errors can occur.

Example

CELEBS28

```
/* CELEBS28

   This example computes y as the hyperbolic sine of &pi.&slr.2.

 */
#include <math.h>
#include <stdio.h>

int main(void)
{
   double pi, x, y;

   pi = 3.1415926535;
   x = pi/2;
   y = sinh(x);

   printf("sinh( %lf ) = %lf\n", x, y);
}
```

Output

```
sinh( 1.570796 ) = 2.301299
```

Related information

- “math.h” on page 44
- “acos(), acosf(), acosl() — Calculate arccosine” on page 135
- “acosh(), acoshf(), acoshl() — Calculate hyperbolic arccosine” on page 139
- “asin(), asinf(), asinl() — Calculate arcsine” on page 185
- “asinh(), asinhf(), asinhl() — Calculate hyperbolic arcsine” on page 188
- “atan(), atanf(), atanl(), atan2(), atan2f(), atan2l() — Calculate arctangent” on page 192
- “atanh(), atanhf(), atanhl() — Calculate hyperbolic arctangent” on page 195
- “cos(), cosf(), cosl() — Calculate cosine” on page 328
- “cosh(), coshf(), coshl() — Calculate hyperbolic cosine” on page 331

- “sin(), sinf(), sinl() — Calculate sine” on page 1672
- “tan(), tanf(), tanl() — Calculate tangent” on page 1815
- “tanh(), tanhf(), tanhl() — Calculate hyperbolic tangent” on page 1818

sinhd32(), sinhd64(), sinhd128() - Calculate hyperbolic sine

Standards

Standards / Extensions	C or C++	Dependencies
C/C++ DFP	both	z/OS V1.10

Format

```
#define __STDC_WANT_DEC_FP__
#include <math.h>

_Decimal32  sinhd32(_Decimal32 x);
_Decimal64  sinhd64(_Decimal64 x);
_Decimal128 sinhd128(_Decimal128 x);
_Decimal32  sinh(_Decimal32 x);     /* C++ only */
_Decimal64  sinh(_Decimal64 x);     /* C++ only */
_Decimal128 sinh(_Decimal128 x);    /* C++ only */
```

General description

Calculates the hyperbolic sine of *x*, with *x* expressed in radians.

These functions work in IEEE decimal floating-point format. See “IEEE decimal floating-point” on page 95 for more information.

Note: To use IEEE decimal floating-point, the hardware must have the Decimal Floating-Point Facility installed.

Returned value

If successful, the function returns the hyperbolic sine of *x* with x expressed in radians.

If the result would overflow, the function returns ±HUGE_VAL_D32, ±HUGE_VAL_D64, or ±HUGE_VAL_D128 according to the value of *x*, and sets errno to ERANGE. No other errors can occur.

Example

CELEBS75

```
/* CELEBS75

   This example illustrates the sinhd64() function.

*/

#define __STDC_WANT_DEC_FP__
#include <math.h>
#include <stdio.h>

int main(void)
{
   _Decimal64 pi, x, y;
   pi = 3.1415926535DD;
   x = pi/2.0DD;
   y = sinhd64(x);

   printf("sinhd64( %Df ) = %Df\n", x, y);
}
```

Related information

- “math.h” on page 44
- “atanh(), atanhf(), atanhl() — Calculate hyperbolic arctangent” on page 195
- “acosd32(), acosd64(), acosd128() - Calculate arccosine” on page 137
- “acoshd32(), acoshd64(), acoshd128() - Calculate hyperbolic arccosine” on page 140
- “asind32(), asind64(), asind128() - Calculate arcsine” on page 187
- “asinhd32(), asinhd64(), asinhd128() - Calculate hyperbolic arcsine” on page 189
- “atand32(), atand64(), atand128(), atan2d32(), atan2d64(), atan2d128() - Calculate arctangent” on page 194
- “atanhd32(), atanhd64(), atanhd128() - Calculate hyperbolic arctangent” on page 197
- “cosd32(), cosd64(), cosd128() — Calculate cosine” on page 329
- “coshd32(), coshd64(), coshd128() - Calculate hyperbolic cosine” on page 332
- “sind32(), sind64(), sind128() — Calculate sine” on page 1673
- “tand32(), tand64(), tand128() - Calculate tangent” on page 1817
- “tanhd32(), tanhd64(), tanhd128() - Calculate hyperbolic tangent” on page 1820

__sinpid32(), __sinpid64(), __sinpid128() — Calculate sine of pi * x

Standards

Standards / Extensions	C or C++	Dependencies
Language Environment	both	z/OS V1.8

Format

```
#define __STDC_WANT_DEC_FP__
#include <math.h>

_Decimal32  __sinpid32(_Decimal32 x);
_Decimal64  __sinpid64(_Decimal64 x);
_Decimal128 __sinpid128(_Decimal128 x);
```

General description

Calculates the sine of pi * *x*, with x expressed in radians.

Notes:

1. To use IEEE decimal floating-point, the hardware must have the Decimal Floating-Point Facility installed.
2. These functions work in IEEE decimal floating-point format. See "IEEE Decimal Floating-Point" for more information.

Returned value

If successful, the function returns the calculated value, expressed as a _Decimal32, _Decimal64, or _Decimal128 number.

If x is outside prescribed limits, the value is not calculated. Instead, the function returns 0 and sets errno to EDOM.

Example

```
/* CELEBS70

   This example illustrates the __sinpid64() function.

*/

#define  __STDC_WANT_DEC_FP__
#include <math.h>
#include <stdio.h>

int main(void)
{
   _Decimal64 x, y;

   x = 0.5DD;
   y = __sinpid64(x);

   printf("__sinpid64(%Df) = %Df\n", x, y);
}
```

Related information

- "math.h" on page 44
- "__cospid32(), __cospid64(), __cospid128() — Calculate cosine of pi *x" on page 333

sleep() — Suspend execution of a thread

Standards

Standards / Extensions	C or C++	Dependencies
POSIX.1 XPG4 XPG4.2 Single UNIX Specification, Version 3	both	POSIX(ON)

Format

```
#define _POSIX_SOURCE
#include <unistd.h>

unsigned int sleep(unsigned int seconds);
```

General description

Suspends thread execution for a specified number of *seconds.* Because of processor delays, the thread can sleep slightly longer than this specified time. An unblocked signal received during this time (for which the action is to invoke a signal handler function or to end the thread) "wakes up" the thread prematurely. When that function returns, sleep() returns immediately even if there is sleep time remaining.

This function is supported only in a POSIX program.

Returned value

If the thread slept for the full specified time, sleep() returns 0.

If the thread awoke prematurely because of a signal whose action is to invoke a signal-handling function or to end the thread, sleep() returns the number of seconds remaining in its sleep time (that is, the value of *seconds* minus the actual number of seconds that the thread was suspended).

sleep() always succeeds, so there is no failure return. An abend is generated when any failures are encountered that prevent this function from completing successfully.

There are no documented errno values.

Example

CELEBS29

```
/* CELEBS29

   This example suspends execution for a specified time.

 */
#define _POSIX_SOURCE
#include <stdio.h>
#include <time.h>
#include <unistd.h>

main() {
  unsigned int ret;
  time_t t;
  time(&t);
  printf("starting sleep at %s", ctime(&t));
  ret = sleep(10);
  time(&t);
  printf("naptime over at %s", ctime(&t));
  printf("sleep() returned %d\n", ret);
}
```

Output

```
starting sleep at Fri Jun 16 07:44:47 2006
naptime over at Fri Jun 16 07:44:58 2006
sleep() returned 0
```

Related information

- “signal.h” on page 63
- “unistd.h” on page 82
- “alarm() — Set an alarm” on page 156
- “bsd_signal() — BSD version of signal()” on page 219
- “kill() — Send a signal to a process” on page 931
- “killpg() — Send a signal to a process group” on page 934
- “longjmp() — Restore stack environment” on page 1005
- “_longjmp() — Nonlocal goto” on page 1008
- “pause() — Suspend a process pending a signal” on page 1173
- “pthread_kill() — Send a signal to a thread” on page 1283
- “sigaction() — Examine or change a signal action” on page 1611
- “sigignore() — Set disposition to ignore a signal” on page 1637
- “siglongjmp() — Restore the stack environment and signal mask” on page 1640
- “signal() — Handle interrupts” on page 1643
- “sigprocmask() — Examine or change a thread” on page 1651
- “sigset() — Change a signal action or a thread” on page 1656
- “sigsuspend() — Change mask and suspend the thread” on page 1663

__smf_record() — Record an SMF record

Standards

Standards / Extensions	C or C++	Dependencies
z/OS UNIX	both	

Format

```
#include <unistd.h>

int __smf_record(int smf_record_type,
                 int smf_record_subtype,
                 int smf_record_length,
                 char *smf_record);
```

General description

The __smf_record() function writes an SMF record pointed to by *smf_record* of length *smf_record_length* for SMF record type *smf_record_type* and subtype *smf_record_subtyp*e to the SMF data set.

The service can also be used to determine if a particular type or subtype of SMF record is being recorded to avoid the overhead of data collection if the SMF record is not going to be recorded. See *z/OS MVS System Management Facilities (SMF), SA38-0667* for more information on SMF record types and layout.

The caller of this service must be APF-authorized or permitted to either the BPX.SMF facility class profile or the BPX.SMF.type.subtype resource profile in the FACILITY class where type is the SMF record type to be written and subtype is the SMF record subtype to be written. For more information about creating and using these profiles and the restrictions on their use, refer to *z/OS UNIX System Services Planning, GA32-0884*.

Returned value

If successful, __smf_record() returns 0.

If unsuccessful, __smf_record() returns -1 and sets errno to one of the following values:

Error Code
    Description

EINVAL
    The value specified on the length operand was incorrect. Use __errno2() to retrieve the reason code that is associated with the error. The following reason codes can accompany the return code: JRSMFBadRecordLength, JrSMFTypeSubtypeMismatch, or JrSMFRecordLenMismatch.

EMVSERR
    The SMF service returned a nonzero return code. Use __errno2() to determine why the error occurred. The following reason codes can accompany the return code: JRSMFNotAccepting, JRSMFError, JRBadAddress, or JRInternalError.

ENOMEM
    Not enough storage.

EPERM

The calling process is not permitted to the BPX.SMF resource in the FACILITY class and the calling processes are not APF-authorized, or the calling process is permitted to the BPX.SMF.xxx.yyy FACILITY class resource but the environment is dirty. The following reason codes can accompany the return code:

`JRSMFNotAuthorized, JREnvDirty`

Related information

None.

__smf_record2() — Record an SMF record with exit control

Standards

Standards / Extensions	C or C++	Dependencies
z/OS UNIX	both	z/OS V1.12

Format

```
#include <unistd.h>

int __smf_record2 (int smf_record_type,
                   int smf_record_subtype,
                   int smf_record_length,
                   char *smf_record
                   unsigned int smf_exit );
```

General description

The __smf_record2() function writes an SMF record pointed to by *smf_record* of length *smf_record_length* for SMF record type *smf_record_type* and subtype *smf_record_subtype* to the SMF data set.

The *smf_exit* argument defines which user exit is to take control when the SMF record is written. Acceptable values for this argument are as follows:

_SMF_IEFU83 Cause SMF exit IEFU83 to gain control.
_SMF_IEFU84 Cause SMF exit IEFU84 to gain control.

and are defined in <unistd.h>

The service can also be used to determine if a particular type or subtype of SMF record is being recorded to avoid the overhead of data collection if the SMF record is not going to be recorded. See *z/OS MVS System Management Facilities (SMF)* for more information on SMF record types and layout.

The caller of this service must be APF-authorized or permitted to either the BPX.SMF facility class profile or the BPX.SMF.type.subtype resource profile in the FACILITY class where type is the SMF record type to be written and subtype is the SMF record subtype to be written. For more information about creating and using these profiles and the restrictions on their use, see *z/OS UNIX System Services Planning*.

This function is supported under CICS Transaction Server for z/OS only when run in an open transaction environment (OTE).

Note: To determine if a particular SMF record type or subtype is being recorded, specify NULL for the *smf_record* argument. If the return value is 0, the type or subtype is being recorded. If the return value is -1 and errno is EMVSERR with a reason code of JRSMFNotAccepting, SMF is not recording this type or subtype.

Returned value

If successful, __smf_record2() returns 0.

If unsuccessful, __smf_record2() returns -1 and sets errno to one of the following values:

Error Code
    Description

EINVAL
    The value that was specified for an argument was incorrect. Use __errno2() to retrieve the reason code that is associated with the error. The following reason codes can accompany the return code: JRSMFBadRecordLength, JrSMFTypeSubtypeMismatch, or JrSMFRecordLenMismatch.

EMVSERR
    The SMF service returned a nonzero return code. Use __errno2() to retrieve the reason code associated with the error. The following reason codes can accompany the return code: JRSMFNotAccepting, JRSMFError, JRBadAddress, or JRInternalError.

ENOMEM
    Not enough storage is available.

EPERM
    The calling process is not permitted to the BPX.SMF resource in the FACILITY class and the calling processes are not APF-authorized, or the calling process is permitted to the BPX.SMF.xxx.yyy FACILITY class resource but the environment is dirty. The following reason codes can accompany the return code:

```
JRSMFNotAuthorized, JREnvDirty
```

Related information

- “unistd.h” on page 82
- “__smf_record() — Record an SMF record” on page 1680

snprintf() — Format and write data

Standards

Standards / Extensions	C or C++	Dependencies
C99 Single UNIX Specification, Version 3 C++ TR1 C99	both	z/OS V1R6

Format

```
#define _ISOC99_SOURCE
#include <stdio.h>

int  snprintf(char *__restrict__ s, size_t n, const char *__restrict__ format, ...);
```

Note: The snprintf() function is also available under the alternate name __snprtf(). The __snprtf() function is accessible in the open name space.

```
#include <stdio.h>

int  __snprtf(char *__restrict__ s, size_t n, const char *__restrict__ format, ...);
```

General description

Equivalent to fprintf(), except that the output is written into an array (specified by argument *s*) rather than to a stream. If *n* is zero, nothing is written, and *s* may be a null pointer. Otherwise, output characters beyond the n-1st are discarded rather than being written to the array, and a null character is written at the end of the characters actually written into the array. If copying takes place between objects that overlap, the behavior is undefined.

Returned value

Returns the number of characters that would have been written had *n* been sufficiently large, not counting the terminating null character, or a negative value if an encoding error occurred. Thus, the null-terminated output has been completely written if and only if the returned value is nonnegative and less than *n*.

Errors

Function fails if:

- The value of *n* is greater than {INT_MAX} or the number of bytes needed to hold the output excluding the terminating null is greater than {INT_MAX}. In this case, the function returns a negative value and sets errno to EOVERFLOW

sockatmark() — Determine whether a socket is at the out-of-band mark

Standards

Standards / Extensions	C or C++	Dependencies
Single UNIX Specification, Version 3	both	z/OS V1R9

Format

SUSV3:

```
#define _POSIX_C_SOURCE 200112L
#include <sys/socket.h>

int sockatmark(int s);
```

General description

The sockatmark() function determines whether the socket specified by the descriptor *s* is at the out-of-band data mark. If the protocol for the socket supports out-of-band data by marking the stream with an out-of-band data mark, the sockatmark() function returns 1 when all data preceding the mark has been read and the out-of-band data mark is the first element in the receive queue. The sockatmark() function does not remove the mark from the stream.

Argument	Description
s	the descriptor used to determine if the socket is at the out-of-band data mark

Returned value

Upon successful completion, the sockatmark() function returns a value indicating whether the socket is at an out-of-band data mark. If the protocol has marked the data stream and all data preceding the mark has been read, the return value is 1; if there is no mark, or if data precedes the mark in the receive queue, the sockatmark() function returns 0. Otherwise, it returns a value of -1 and set errno to indicate the error.

Error Code
Description

EBADF
The *s* argument is not a valid file descriptor.

ENOTTY
The *s* argument does not specify a descriptor for a socket.

Example

CELEBS74

```
/* CELEBS74

   This example demonstrates the use of the sockatmark() function.

   Expected output:
   C: Sending regular data
   C: Sending OOB data
   S: Received "123a"
   S: At the mark
   S: Received "b"

*/

#define _POSIX_C_SOURCE 200112L
#include <sys/socket.h>
#include <netinet/in.h>
#include <unistd.h>
#include <stdlib.h>
#include <string.h>
#include <stdio.h>

int main(int argc, char **argv) {
   struct sockaddr_in saddr;
   socklen_t addr_len = sizeof(saddr);
   int port = 12121, n, ld, connfd, servfd;
   char buffer[25];
   pid_t pid;

   if((ld = socket(AF_INET,SOCK_STREAM,0)) == -1){
      printf("socket error\n");
      return 0;
   }

   saddr.sin_family = AF_INET;
   saddr.sin_port    = 12121;

   if(bind(ld,(struct sockaddr *)&saddr,addr_len) == -1){
      printf("bind error\n");
      return 0;
   }

   if(listen(ld,5) == -1){
      printf("listen error\n");
      return 0;
   }

   pid = fork();
   if(pid==0){
      if((connfd = socket(AF_INET,SOCK_STREAM,0)) == -1){
         printf("socket error\n");
         exit(0);
      }

      if(connect(connfd,(struct sockaddr *)&saddr,addr_len) == -1){
         printf("connect error\n");
         exit(0);
      }

      printf("C: Sending regular data\n");
      send(connfd,"123",3,0);
      printf("C: Sending OOB data\n");
      send(connfd,"ab",2,MSG_OOB);

      close(connfd);
      exit(0);
   }
   else {
      servfd = accept(ld,(struct sockaddr *)&saddr,&addr_len);
      if(servfd == -1) {
         printf("accept error\n");
         exit(0);
      }

      sleep(5);
      memset(buffer,0,sizeof(buffer));
      recv(servfd,&buffer,sizeof(buffer),0);
      printf("S: Received \"%s\"\n",buffer);

      memset(buffer,0,sizeof(buffer));

      n = sockatmark(servfd);
      if(n == 1) printf("S: At the mark\n");
      recv(servfd,&buffer,sizeof(buffer),MSG_OOB);
      printf("S: Received \"%s\"\n",buffer);

      close(servfd);
      close(ld);
   }
}
```

Related information

- “sys/socket.h” on page 74

sock_debug() — Provide syscall tracing facility

Standards

Standards / Extensions	C or C++	Dependencies
z/OS UNIX	both	

Format

```
#define _OPEN_SYS_SOCK_EXT
#include <sys/socket.h>

void sock_debug(int onoff);
```

General description

Bulk mode sockets are not supported. This function has no effect.

Returned value

sock_debug() returns no values.

Related information

- "sys/socket.h" on page 74

sock_debug_bulk_perf0() — Produce a report when a socket is configured

Standards

Standards / Extensions	C or C++	Dependencies
z/OS UNIX	both	

Format

```
#define _OPEN_SYS_SOCK_EXT
#include <sys/socket.h>

void sock_debug_bulk_perf0(int onoff);
```

General description

Restriction: This function is not supported in AMODE 64.

Bulk mode sockets are not supported. This function has no effect.

Returned value

sock_debug_bulk_perf0() returns no values.

Related information

- "sys/socket.h" on page 74

sock_do_bulkmode() — Use bulk mode for messages read by a socket

Standards

Standards / Extensions	C or C++	Dependencies
z/OS UNIX	both	

Format

```
#define _OPEN_SYS_SOCK_EXT
#include <sys/socket.h>

void sock_do_bulkmode(int onoff);
```

General description

Restriction: This function is not supported in AMODE 64.

Bulk mode sockets are not supported. This function has no effect.

Returned value

sock_do_bulkmode() returns no values.

Related information

- "sys/socket.h" on page 74

sock_do_teststor() — Check for attempt to access storage outside

Standards

Standards / Extensions	C or C++	Dependencies
z/OS UNIX	both	

Format

```
#define _OPEN_SYS_SOCK_EXT
#include <sys/socket.h>

void sock_do_teststor(int onoff);
```

General description

The sock_do_teststor call is used to check for calls that attempt to access storage outside the caller's address space.

Parameter
Description

onoff A parameter that can be set to zero or nonzero.

If *onoff* is set to a nonzero value, for either inbound or outbound sockets, both the address of the message buffer and the message buffer are checked for addressability for each bulk mode socket call. The EFAULT error condition is set if there is an addressing problem. If *onoff* is set to 0, address checking is not done by the socket library program. If an error occurs when *onoff* is 0, normal runtime error handling reports the exception condition.

To improve response time, you can disable this checking when your program has been tested.

As an alternative to calling sock_do_teststor, with *onoff* set to a nonzero value, you can include the statement SOCKTESTSTOR in the file /etc/resolv.conf or data set tcpip.TCPIP.DATA. When the process is started, all the programs using bulk mode sockets for this process will validate the storage for the caller's parameters.

Returned value

sock_do_teststor() returns no values.

Related information

- "sys/socket.h" on page 74

socket() — Create a socket

Standards

Standards / Extensions	C or C++	Dependencies
XPG4.2 Single UNIX Specification, Version 3	both	

Format

X/Open:

```
#define _XOPEN_SOURCE_EXTENDED 1
#include <sys/socket.h>

int socket(int domain, int type, int protocol);
```

Berkeley sockets:

```
#define _OE_SOCKETS
#include <sys/socket.h>

int socket(int *domain, int type, int protocol);
```

General description

The socket() function creates an endpoint for communication and returns a socket descriptor representing the endpoint. Different types of sockets provide different communication services.

Parameter
Description

domain The address domain requested, either AF_INET, AF_INET6, AF_UNIX, or AF_RAW.

type The type of socket created, either SOCK_STREAM, SOCK_DGRAM, or SOCK_RAW.

protocol
The protocol requested. Some possible values are 0, IPPROTO_UDP, or IPPROTO_TCP.

The *domain* parameter specifies a communication domain within which communication is to take place. This parameter selects the address family (format of addresses within a domain) that is used. The families supported are AF_INET and AF_INET6, which is the Internet domain, and AF_UNIX, which is the local socket domain. These constants are defined in the **sys/socket.h** include file.

The *type* parameter specifies the type of socket created. The type is analogous with the semantics of the communication requested. These socket type constants are defined in the **sys/socket.h** include file. The types supported are:

Socket Type
Description

SOCK_DGRAM
Provides datagrams, which are connectionless messages of a fixed maximum length whose reliability is not guaranteed. Datagrams can be corrupted, received out of order, lost, or delivered multiple times. This type is supported in the AF_INET, AF_INET6, and AF_UNIX domains.

SOCK_RAW
Provides the interface to internal protocols (such as IP and ICMP). This type is supported in the AF_INET and AF_INET6 domains. You must be a superuser to use this type.

SOCK_STREAM
Provides sequenced, two-way byte streams that are reliable and connection-oriented. They support a mechanism for out-of-band data. This type is supported in the AF_INET, AF_INET6, and AF_UNIX domains.

Understanding the socket() Parameters: The *protocol* parameter specifies a particular protocol to be used with the socket. In most cases, a single protocol exists to support a particular type of socket in a particular address family. If the *protocol* parameter is set to 0, the system selects the default protocol number for the domain and socket type requested. Protocol numbers are found in the *tcpip*.ETC.PROTO data set. Alternatively, the getprotobyname() call can be used to get the protocol number for a protocol with a known name.

Note: The *protocol* field *must* be set to 0, if the *domain* parameter is set to AF_UNIX.

SOCK_STREAM sockets model duplex-byte streams. They provide reliable, flow-controlled connections between peer application programs. Stream sockets are either active or passive. Active sockets are used by clients who start connection requests with connect(). By default, socket() creates active sockets. Passive sockets are used by servers to accept connection requests with the connect() call. You can transform an active socket into a passive socket by binding a name to the socket with the bind() call and by indicating a willingness to accept connections with the listen() call. After a socket is passive, it cannot be used to start connection requests.

In the AF_INET and AF_INET6 domains, the bind() call applied to a stream socket lets the application program specify the networks from which it is willing to accept connection requests. The application program can fully specify the network interface by setting the *Internet address* field in the **address** structure to the Internet address of a network interface. Alternatively, the application program can use a *wildcard* to specify that it wants to receive connection requests from any network. For AF_INET sockets, this is done by setting the *Internet address* field in the **address** structure to the constant INADDR_ANY, as defined in **<netinet/in.h>**. For AF_INET6 sockets, this is done by setting the *Internet address* field in the address structure to in6addr_any as defined in **<netinet/in.h>**.

After a connection has been established between stream sockets, any of the data transfer calls can be used: (read(), readv(), recv(), recvfrom(), recvmsg(), send(), sendmsg(), sendto(), write(), and writev()). Usually, the read()-write() or send()-recv() pairs are used for sending data on stream sockets. If out-of-band data is to be exchanged, the send()-recv() pair is normally used.

SOCK_DGRAM sockets model datagrams. They provide connectionless message exchange without guarantees of reliability. Messages sent have a maximum size. Datagram sockets are supported in the AF_UNIX domain.

There is no active or passive analogy to stream sockets with datagram sockets. Servers must still call bind() to name a socket and to specify from which network interfaces it wishes to receive packets. Wildcard addressing, as described for stream sockets, applies for datagram sockets also. Because datagram sockets are connectionless, the listen() call has no meaning for them and must not be used with them.

After an application program has received a datagram socket, it can exchange datagrams using the sendto() and recvfrom(), or sendmsg() and recvmsg() calls. If the application program goes one step further by calling connect() and fully specifying the name of the peer with which all messages will be exchanged, then the other data transfer calls read(), write(), readv(), writev(), send(), and recv() can also be used. For more information on placing a socket into the connected state, see "connect() — Connect a socket" on page 308.

Datagram sockets allow messages to be broadcast to multiple recipients. Setting the destination address to be a broadcast address is network-interface-dependent (it depends on the class of address and whether *subnets*—logical networks divided into smaller physical networks to simplify routing—are used). The constant INADDR_BROADCAST, defined in **netinet/in.h**, can be used to broadcast to the primary network if the primary network configured supports broadcast.

Outgoing packets have an IP header prefixed to them. IP options can be set and inspected using the setsockopt() and getsockopt() calls, respectively. Incoming packets are received with the IP header and options intact.

Sockets are deallocated with the close() call.

Note: For AF_UNIX, when closing sockets that were bound, you should also use unlink() to delete the file created at bind() time.

Special behavior for C++: To use this function with C++, you must use the _XOPEN_SOURCE_EXTENDED 1 feature test macro.

Returned value

If successful, socket() returns a nonnegative socket descriptor.

If unsuccessful, socket() returns -1 and sets errno to one of the following values:

Error Code
Description

EACCES
Permission to create a socket of the specified type or protocol is denied.

EAFNOSUPPORT
The address family is not supported (it is not AF_UNIX, AF_INET, or AF_INET6).

EAGAIN
Resource temporarily unavailable.

EINVAL
The request is invalid or not supported.

EIO There has been a network or transport failure.

ENOBUFS
Insufficient system resources are available to complete the call.

ENOENT
There was no NETWORK statement in the parmlib member to match the specified domain.

EPROTONOSUPPORT
The protocol is not supported in this domain or this protocol is not supported for this socket type.

EPROTOTYPE
The socket type is not supported by the protocol.

Example

The following are examples of the socket() call.

```
int s;
char *name;
int socket(int domain, int type, int protocol);
⋮
/* Get stream socket in Internet
domain with default protocol */
s = socket(AF_INET, SOCK_STREAM, 0);
⋮
/* Get stream socket in local socket
domain with default protocol */
s = socket(AF_UNIX, SOCK_STREAM, 0);
```

Related information

- “sys/socket.h” on page 74
- “accept() — Accept a new connection on a socket” on page 105
- “bind() — Bind a name to a socket” on page 210
- “close() — Close a file” on page 288
- “connect() — Connect a socket” on page 308
- “fcntl() — Control open file descriptors” on page 474
- “getprotobyname() — Get a protocol entry by name” on page 756
- “getsockname() — Get the name of a socket” on page 775
- “getsockopt() — Get the options associated with a socket” on page 777
- “ioctl() — Control device” on page 869
- “read() — Read from a file or socket” on page 1376
- “readv() — Read data on a file or socket and store in a set of buffers” on page 1389
- “recv() — Receive data on a socket” on page 1398
- “recvfrom() — Receive messages on a socket” on page 1401
- “recvmsg() — Receive messages on a socket and store in an array of message headers” on page 1405
- “select(), pselect() — Monitor activity on files or sockets and message queues” on page 1470
- “selectex() — Monitor activity on files or sockets and message queues” on page 1479
- “send() — Send data on a socket” on page 1494
- “sendmsg() — Send messages on a socket” on page 1500
- “shutdown() — Shut down all or part of a duplex connection” on page 1606
- “write() — Write data on a file or socket” on page 2085

- "writev() — Write data on a file or socket from an array" on page 2092

socketpair() — Create a pair of sockets

Standards

Standards / Extensions	C or C++	Dependencies
XPG4.2 Single UNIX Specification, Version 3	both	

Format

X/Open:

```
#define _XOPEN_SOURCE_EXTENDED 1
#include <sys/socket.h>

int socketpair(int *domain, int type, int protocol, int socket_vector[2]);
```

Berkeley sockets:

```
#define _OE_SOCKETS
#include <sys/socket.h>

int socketpair(int *domain, int type, int protocol, int sv[2]);
```

General description

The socketpair() function acquires a pair of sockets of the type specified that are unnamed and connected in the specified domain and using the specified protocol. For socket pairs in the AF_UNIX domain, the protocol *must* be 0.

Parameter
Description

domain The domain in which to open the socket. Although socket pairs can be obtained for AF_INET domain sockets, it is recommended that AF_UNIX domain sockets be used for socket pairs.

type The type of socket created, either SOCK_STREAM, or SOCK_DGRAM.

protocol
The protocol requested *must* be 0.

sv The descriptors used to refer to the obtained sockets.

Special behavior for C++: To use this function with C++, you must use the _XOPEN_SOURCE_EXTENDED 1 feature test macro.

Returned value

If successful, socketpair() returns a nonnegative socket descriptor.

If unsuccessful, socketpair() returns -1 and sets errno to one of the following values:

Error Code
Description

EACCES
Permission to create a socket of the specified type or protocol is denied.

EFAULT
sv is not in the writable part of the user's address space.

EINVAL
The request is invalid or not supported.

EMFILE
Too many files are open for this process.

ENFILE
Too many files are open in the system.

ENOBUFS
Insufficient system resources are available to complete the call.

EOPNOSUPPORT
The protocol does not allow for the creation of socket pairs.

EPROTONOSUPPORT
The protocol is not supported in this domain or this protocol is not supported for this socket type.

EPROTOTYPE
The socket type is not supported by the protocol.

Example

The following are examples of the socketpair() call.

```
#include <types.h>
#include <sys/socket.h>

int sv[2];
⋮
/* Get stream socket in UNIX
domain with default protocol */
if (socketpair(AF_UNIX, SOCK_STREAM, 0, sv) < 0)
printf ("Error occurred while trying to get a socket pair.\n");
else
⋮
```

Related information

- “sys/socket.h” on page 74
- “socket() — Create a socket” on page 1688

spawn(), spawnp() — Spawn a new process

Standards

Standards / Extensions	C or C++	Dependencies
POSIX.4b z/OS UNIX	both	

Format

```
#define _POSIX_SOURCE
#include <spawn.h>

pid_t spawn(const char *path,
            const int fd_count,
            const int fd_map[],
            const struct inheritance *inherit,
            const char *argv[],
              const char *envp[]);

pid_t spawnp(const char *file,
             const int fd_count,
             const int fd_map[],
             const struct inheritance *inherit,
             const char *argv[],
             const char *envp[]);
```

General description

The spawn() and spawnp() functions create a new process from the specified process image. spawn() and spawnp() create the new process image from a regular executable file called the new process image file.

To execute a C program as a result of this call, enter the function call as follows:

```
int main (int argc, char *argv[]);
```

Where *argc* is the argument count and *argv* is an array of character pointers to the arguments themselves. In addition, the following variable:

```
extern char **environ;
```

is initialized as a pointer to an array of character pointers to the environment strings. The *argv* and `environ` arrays are each terminated by a NULL pointer. The NULL pointer terminating the *argv* array is not counted in *argc*.

Supported parameters are:

Parameter
Description

path
Path name used by spawn() that identifies the new process image file to execute.

file
Used by spawnp() to construct path name that identifies the new process image file. If the file parameter contains a slash character, spawnp() uses the file parameter as a path name for the new process image file. Otherwise, spawnp() obtains the path prefix for this file by a search of the directories passed as the environment variable `PATH`.

fd_count
Specifies the number of file descriptors the child process inherits. It may take values from zero to **OPEN_MAX**. Except for those file descriptors designated by `SPAWN_FDCLOSED`, each of the child's file descriptors, `x`, in the range zero to *fd_count*-1 inherits descriptor *fd_map*[*x*] from the parent process.

The files from *fd_count* through **OPEN_MAX** are closed in the child process, as are any elements of *fd_map* designated as `SPAWN_FDCLOSED`.

fd_map
If the *fd_map* parameter is NULL, *fd_count* and *fd_map* are ignored. All file descriptors except those with the `FD_CLOEXEC` or `FD_CLOFORK` attribute are inherited without reordering. File descriptors with the `FD_CLOEXEC` or `FD_CLOFORK` attribute are closed under simple inheritance.

For those file descriptors that remain open, all other attributes of the associated file descriptor object and open file description remain unchanged by this operation.

Directory streams open in the calling process are closed in the new process image.

If an element of *fd_map* refers to an invalid file descriptor, then the (EBADF) spawn() or spawnp() posts the error status.

The FD_CLOEXEC and FD_CLOFORK file descriptor attributes are never inherited.

The FD_CLOEXEC and FD_CLOFORK file descriptor attributes have no effect on inheritance when the *fd_map* parameter is not NULL.

Note: For XTI endpoints, *fd_map* must not map to a number greater than 65535 in the child process.

inherit The name of a data area that contains the inheritance structure.

The 'struct inheritance' is defined as follows:

```
struct inheritance {
short flags; --Flags
pid_t pgroup; --Process group
sigset_t sigmask; --Signal mask
sigset_t sigdefault; --Signals set to SIG_DFL
int ctlttyfd; --Cntl tty FD for tcsetpgrp()
}
```

The inherit.flags effect spawn() and spawnp() as follows:

SPAWN_SETGROUP

If the SPAWN_SETGROUP flag is set in inherit.flags, then the child's process group is as specified in inherit.pgroup.

If the SPAWN_SETGROUP flag is set in inherit.flags and inherit.prgroup is set to SPAWN_NEWPGROUP, then the child is in a new process group with a process group ID equal to its process ID.

If the SPAWN_SETGROUP flag is not set in inherit.flags, the new child process inherits the parent's process group ID.

SPAWN_SETSIGMASK

If the SPAWN_SETSIGMASK flag is set in inherit.flags, the child process initially has the signal mask specified in inherit.sigmask.

SPAWN_SETSIGDEF

If the SPAWN_SETSIGDEF flag is set in inherit.flags, the signals specified in inherit.sigdefault are set to their default actions in the child process. Signals set to the default action in the parent process, are set to the default action in the new process.

Signals set to be caught by the calling process are set to the default action in the child process.

Signals set to be ignored by the calling process are set to be ignored by the new process, unless otherwise specified by the SPAWN_SETSIGDEF flag being set in inherit.flags and the signal being indicated in inherit.sigdefault.

SPAWN_SETTCPGRP

If the SPAWN_SETTCPGRP flag is set in inherit.flags, the file descriptor specified in inherit.ctlttyfd is used to set the controlling terminal file descriptor (tcsetpgrp()) for the child's foreground process group. The child's foreground process group is inherited from the parent, unless the SPAWN_SETGROUP flag in inherit.flags is set, indicating that the value specified in inherit.pgroup is to be used to determine the child's process group.

SPAWN_PROCESS_INITTAB
If this flag is set, spawn attempts to read the /etc/inittab file and process the entries found there. This processing involves the spawning of child shell processes to run each of the commands identified in the file. Only the SPAWN_SETSIGMASK flag can be set in combination with this flag. All other flags will be ignored. Use of this flag implies that only file descriptors 0, 1, and 2 will be initially opened in the child process. File descriptor 0 will be initially opened as /dev/null, while file descriptors 1 and 2 will initially opened as /etc/log. The fd_count and fd_map parameters will be ignored. This flag is currently restricted to the /usr/sbin/init process. See *z/OS UNIX System Services Planning*for more information on the /etc/inittab support.

argv The value in the first element of *argv* should point to a file name that is associated with the process being started by the spawn() or spawnp() operation.

The number of bytes available for the new process's combined argument and environment lists is **ARG_MAX**.

envp The value *envp* contains the list of environmental variables that is to be passed to the specified program.

If the set-user-ID mode bit of the new process image file is set, the effective user ID of the new process image is set to the owner id of the new process image file. Similarly, if the set-group-ID mode bit of the new process image file is set, the effective group ID of the new process image is set to the group id of the new process image file. The real user ID, real group ID, and supplementary group IDs of the new process image remain the same as those of the calling process image. The effective user ID and effective group ID of the new process image are saved (as the saved set-user-ID and the saved set-group-ID) for use by the setuid() function.

The new process image inherits the following attributes from the calling process image:
- Process group ID (unless the **SPAWN_SETGROUP** flag is set in inherit.flags)
- Session membership
- Real user ID
- Real group ID
- Supplementary group IDs
- Priority
- Current working directory
- Root directory
- File creation mask
- Signal mask (unless the **SPAWN_SETSIGMASK** flag is set in inherit.flags)
- Signal actions specified as default (SIG_DFL)
- Signal actions specified as ignore (SIG_IGN) (except as modified by inherit.sigdefault and the **SPAWN_SETSIGDEF** flag set in inherit.flags)

The following are differences between the parent process and the child process:
- Signals set to be caught by the calling process are set to the default action (SIG_DFL).
- The process and system utilization times for the child are set to zero.
- Any file locks previously set by the parent are not inherited by the child.
- The child process has no alarms set.
- The child process has no interval timers set.

- The child has no pending signals.
- Memory mappings established by the parent are not inherited by the child.

If the process image was read from a writable file system, then upon successful completion, the spawn() or spawnp() function mark for update the `st_atime` field of the new process image file.

If the spawn() or spawnp() function is successful, the new child process image file is opened, with all the effects of the open() function.

Special behavior for z/OS UNIX Services:

Note: If an application spawns a shell command or utility that performs terminal I/O, the command may fail due to the fact that the shell file descriptors are not initialized. The Shell file descriptors must be defined. An example of how these can be defined in a C application are as follows:

```
stdin = fopen("/tmp/sys.stdin","r");
stdout = fopen("/tmp/sys.stdout","w");
stderr = fopen("/tmp/sys.stderr","w");
```

Aspects of spawn processing are controlled by environment variables. The environment variables that affect spawn processing are the ones that are passed into the spawn syscall and not the environment variables of the calling process. The environment variables of the calling process do not affect spawn processing unless they are the same as those that are passed in *envp*.

The _BPXK_JOBLOG environment variable can be used to specify that WTO messages are to be written to an open HFS job log file. The following are the allowable values:

Value	Description
nn	Job log messages are to be written to open file descriptor nn.
STDERR	Job log messages are to be written to the standard error file descriptor, 2.
None	Job log messages are not to be written. This is the default.

The file that is used to capture messages can be changed at any time by calling the oe_env_np service (BPX1ENV) and specifying _BPXK_JOBLOG with a different file descriptor.

Message capturing is turned off if the specified file descriptor is marked for close on a fork or exec.

Message capturing is process-related. All threads under a given process share the same job log file. Message capturing may be initiated by any thread under that process.

Multiple processes in a single address space can each have different files active as the JOBLOG file; some or all of them can share the same file; and some processes can have message capturing active while others do not.

Only files that can be represented by file descriptors may be used as job log files; MVS data sets are not supported.

Message capturing will be propagated on a fork() or spawn(). In the case where a file descriptor was specified, the physical file must be the same for message capturing to continue in the forked or spawned process. If STDERR was specified, the file descriptor may be re-mapped to a different physical file.

Message capturing may be overridden on exec() or spawn() by specifying the _BPXK_JOBLOG environment variable as a parameter to the exec() or spawn().

Message capturing will only work in forked (BPXAS) address spaces.

Note: This is not true joblog support, messages that would normally go to the JESYSMSG data set are captured, but messages that go to JESMSGLG are not captured.

For more information on the use of environment variables, see *z/OS UNIX System Services Programming: Assembler Callable Services Reference, SA23-2281*.

Security information from the parent's address space is propagated to the child's address space, unless the _BPX_USERID environment variable specifies otherwise. As a result, the child has a security environment equivalent to that of the parent.

The TASKLIB, STEPLIB, or JOBLIB DD data set allocations that are active for the current task are propagated to the child's address space, unless the STEPLIB environment variable specifies otherwise. This causes the child address space to have the same exact MVS program search order as the calling parent task.

The accounting information of the parent's address space is propagated to the child's address space. See *z/OS UNIX System Services Planning, GA32-0884*.

The jobname of the parent is propagated to the child and appended with a numeric value in the range of 1-9 if the jobname is 7 characters or less. If the jobname is 8 characters, then it is propagated as-is. When a jobname is appended with a numeric value, the count wraps back to 1 when it exceeds 9.

If the calling parent task is in a WLM enclave, the child is joined to the same WLM enclave. This allows WLM to manage the parent and child as one "business unit of work" entity for system accounting and management purposes.

To allow the caller to control whether the spawned child process runs in a separate address space from the parent address space or in the same address space, the spawn service allows for the specification of the _BPX_SHAREAS environment variable. The following are the accepted values for the _BPX_SHAREAS environment variable, and the actions taken for each value:

1. _BPX_SHAREAS=YES - Indicates that the child process that is to be created is to run in the same address space as the parent. In the following circumstances, the _BPX_SHAREAS=YES value cannot be honored, and the child process is created in its own address space:
   - If the program to be run is a set-user-ID or set-group-ID program that would cause the effective user-ID or group-ID of the child process to be different from that of the parent process.
   - If the program to be run is an APF-authorized HFS or MVS program and the caller is not running APF authorized.
   - If the program to be run is an unauthorized HFS or MVS program and the caller is running APF authorized.

- If the specified file name represents an external link or a sticky bit file. However, if the program that is to be run is a shell script and _BPX_SPAWN_SCRIPT=YES is set, the process runs in the same address space. _BPX_SPAWN_SCRIPT only has an effect while running in the z/OS shell (/bin/sh... NOT /bin/tcsh).
- If the address space of the parent lacks the necessary resources to create another process within the address space.

Note that only one local spawned process per TSO address space is supported at a given time. This is done to reduce conflict among multiple shells running in the same address space.

2. _BPX_SHAREAS=MUST - Indicates that the child process that is to be created must run in the same address space as the parent, or the spawn request will fail. In the following circumstances, the _BPX_SHAREAS=MUST value cannot be honored, and the spawn invocation fails:
   - If the program to be run is a set-user-ID or set-group-ID program that would cause the effective user ID or group ID of the child process to be different from that of the parent process.
   - If the program to be run is an APF-authorized HFS or MVS program and the caller is not running APF authorized.
   - If the program to be run is an unauthorized HFS or MVS program and the caller is running APF authorized.
   - If the address space of the parent lacks the necessary resources to create another process within the address space.
3. _BPX_SHAREAS=REUSE - Indicates that the child process to be created is to run in the same address space as the parent; also, that it will be created as a medium-weight process. Specifying REUSE allows the caller to indicate that it wants to reuse the existing process structure for locally spawned processes.

   The same rules that apply to the creation of a local spawn process apply to the specification of a local spawn medium-weight process. In addition, in the following circumstances, the _BPX_SHAREAS=REUSE value cannot be honored, and the child process will be created as a non-medium weight local spawn process:
   - If PTRACE is active for the process.
   - If the program to execute is a REXX exec.

   For performance reasons, the STEPLIB that is specified for each medium-weight process that is created for the address space should be the same.
4. _BPX_SHAREAS=NO - Indicates that the child process that is to be created is to run in a separate address space from the address space of the parent. This is the default behavior for the spawn service if the _BPX_SHAREAS environment variable is not specified, or if it contains an unsupported value.

If you specify the _BPX_USERID environment variable, then spawn() creates the new address space and image with the specified userid's identity. The invoker of spawn() must be authorized to change MVS identity. The resulting spawn() image will emerge as if a program had done a fork(), setgid(), initgroups(), setuid(), and exec.

The value of _BPX_USERID can be any 1-to-8-character XPG4 compliant username. If you specify both _BPX_USERID and _BPX_SHAREAS, then spawn() ignores _BPX_SHAREAS, and creates a new address space with the new identity.

If the caller of the spawn() function is the z/OS UNIX shell (i.e /bin/sh), then the setting of the _BPX_SPAWN_SCRIPT= environment variable to YES is recommended. The setting of this variable to YES provides a more efficient mechanism to invoke z/OS UNIX shell scripts.

To support the creation and propagation of a STEPLIB environment to the new process image, spawn() and spawnp() allow for the specification of a STEPLIB environment variable. The following are the accepted values for the STEPLIB environment variable and the actions taken for each:

- STEPLIB=NONE. No Steplib DD is to be created for the new process image.
- STEPLIB=CURRENT. The TASKLIB, STEPLIB or JOBLIB DD data set allocations that are active for the calling task at the time of the call to spawn() and spawnp() are propagated to the new process image, if found to be cataloged. Uncataloged data sets are not propagated to the new process image.
- STEPLIB=Dsn1:Dsn2:,...DsnN. The specified data sets, Dsn1:Dsn2:...DsnN, are built into a STEPLIB DD in the new process image.

  Note: The actual name of the DD is not STEPLIB, but is a system-generated name that has the same effect as a STEPLIB DD.

The data sets are concatenated in the order specified. The specified data sets must follow standard MVS data set naming conventions. Data sets found to be in violation of this standard are ignored. If the data sets do follow the standard, but:

- The caller does not have the proper security access to a data set.
- A data set is uncataloged or is not in load library format.

then the data set is ignored. Because the data sets in error are ignored, the executable file may run without the proper STEPLIB environment. If a data set is in error due to improper security access a X'913' abend is generated. The dump for this abend can be suppressed by your installation.

If the STEPLIB environment variable is not specified, spawn() and spawnp() default behavior is the same as if STEPLIB=CURRENT were specified.

If the program to be invoked is a set-user-ID or set-group-ID file and the user-ID or group-ID of the file is different from that of the current process image, then the data sets to be built into the STEPLIB environment for the new process image must be found in the system sanction list for set-user-id and set-group-id programs. Only those data sets that are found in the sanction list are built into the STEPLIB environment for the new process image. For detailed information regarding the sanction list, and for information on STEPLIB performance considerations, see *z/OS UNIX System Services Planning, GA32-0884*.

Notes:

1. A prior loaded copy of an HFS program in the same address space is reused under the same circumstances that apply to the reuse of a prior loaded MVS unauthorized program from an unauthorized library by the MVS XCTL service with the following exceptions:
   - If the calling process is in Ptrace debug mode, a prior loaded copy is not reused.
   - If the calling process is not in Ptrace debug mode, but the only prior loaded usable copy found of the HFS program is in storage modifiable by the caller, the prior copy is not reused.
2. If the specified file name represents an external link or a sticky bit file, the program is loaded from the caller's MVS load library search order. For an external link, the external name is only used if the name is eight characters or

less, otherwise the caller receives an error from the `loadhfs` service. For a sticky bit program, the file name is used if it is eight characters or less. Otherwise, the program is loaded from the HFS, and the following restriction exists for sticky bit programs that have the `set-user-ID` or `set-group-ID` attribute:

If the program is found in the MVS program search order, the MVS program name must have a `BPX.STICKYSUG.program_name` resource profile defined in the RACF FACILITY class. See the*z/OS UNIX System Services Planning* book for details on defining the resource profile. Failure to follow this restriction will cause the abend EC6 with code xxxxE055.

3. If the calling task is in a WLM enclave, the resulting task in the new process image is joined to the same WLM enclave. This allows WLM to manage the old and new process images as one "business unit of work" entity for system accounting and management purposes.

Note: If you are expecting this function to take advantage of the z/OS UNIX magic number support, the Language Environment runtime option to POSIX(ON) must have been set when the process was initialized. Attempting to use magic number support with a process initialized with POSIX(OFF) may produce undesirable effects. See *z/OS UNIX System Services Planning, GA32-0884* and *z/OS V2R2.0 UNIX System Services User's Guide, SA23-2279* for details and uses of the z/OS UNIX magic number.

Returned value

If successful, spawn() and spawnp() return the value of the process ID of the child process to the parent process.

If unsuccessful, spawn() and spawnp() return -1, no child process is created, and they set errno to one of the following values:

Error Code
Description

E2BIG The number of bytes used by the argument and environment list of the new process image is greater than the system-imposed limit of **ARG_MAX** bytes.

EACCES
Search permission is denied for a directory in the path of the new process image file or the new process image file denies execution permission, or the new process image file is not a regular file and the implementation does not support execution of files of its type.

EAGAIN
The system lacked the necessary resources to create another process or the system-imposed limit on the total number of processes or UIDs under execution by a single user would be exceeded.

EBADF
An entry in the `fd_map` array refers to an invalid file descriptor or the controlling terminal file descriptor specified in the `inherit.ctlttyfd` is not valid.

EFAULT
The system detected an invalid address in attempting to use a parameter of the call.

EINVAL
One or more of the following conditions were detected:

- The username that was specified on the _BPX_USERID environment variable has an incorrect length.
- An attribute that was specified in the inheritance structure (BPXYINHE) is not valid or contains an unsupported value.
- The version number that was specified for the inheritance structure (BPXYINHE) is not valid.
- The inheritance structure length that was specified by the Inherit_area_len parameter or within the inheritance structure does not contain a length that is appropriate for the BPXYINHE version.
- The process group ID that was specified in the inheritance structure is less than zero or has some other unsupported value.

The following reason codes can accompany the return code: JROK, JRUserNameLenError, JRJsRacXtr, JRInheUserid, JRInheRegion, JRInheCPUTime, JRInheDynamber, JRInheAccountData, JRInheCWD, JRInheSetPgrp, JRInheVersion, and JRInheLength.

ELOOP
A loop exists in symbolic links encountered during resolution *file* argument. This error is issued if more than 8 symbolic links are detected in the resolution of file name.

EMVSERR
An MVS internal error has occurred. This may indicate a problem with security permissions for the user calling spawn() or spawnp().

EMVSSAF2ERR
The executable file is a `set-user-ID` or `set-group-ID` file and the file owner's UID or GID is not defined to the Security Authorization Facility (SAF), or _BPX_USERID was specified and the specified username was not defined to SAF with a z/OS UNIX segment.

ENAMETOOLONG
The length of the *path* or *file* arguments, or an element of the environment variable PATH prefixed to *file* exceeds **PATH_MAX**, or a path name component is longer than **NAME_MAX** and {**_POSIX_NO_TRUNC**} is in effect for that file.

ENOENT
One or more components of the path name of the new process image file do not exist or the *path* or *file* argument is empty.

ENOEXEC
The new process image file has the appropriate access permission but is not in the proper format.

Note: Reason codes further qualify the errno. For most of the reason codes, see *z/OS UNIX System Services Messages and Codes*.

For ENOEXEC, the reason codes are:

Reason Code	Explanation
X'xxxx0C27'	The target HFS file is not in the correct format to be an executable file.
X'xxxx0C31'	The target HFS file is built at a level that is higher than that supported by the running system.

ENOMEM
The new process requires more memory than is permitted by the hardware or the operating system.

ENOTDIR
A component of the path prefix of the new process image file is not a directory.

ENOTTY
The tcsetpgrp() failed for the specified controlling terminal file descriptor in `inherit.ctlttyfd`. The failure occurred because the calling process does not have a controlling terminal, or the specified file descriptor is not associated with the controlling terminal, or the controlling terminal is no longer associated with the session of the calling process.

EPERM
The spawn failed for one of the following reasons:

- The spawned process is not a process group leader.
- The _BPX_USERID environment variable was specified, but the invoker does not have appropriate privileges to change the MVS identity.
- The invoker does not have the appropriate privileges to change one or more of the attributes specified in the inheritance structure (BPXYINHE).

The following reason codes can accompany the return code: JROK, JRNoChangeIdentity, JRInheUserid, JRInheRegion, JRInheCPUTime, JRInheUmask, and JRInheCWD.

ESRCH
The process group ID specified in `inherit.pgroup` is not that of a process group in the session of the calling process.

Example

The following is an example of a parent program that uses spawn to create a child process.

```
#define _XOPEN_SOURCE_EXTENDED 1
#include <unistd.h>
#include <spawn.h>
#include <stdio.h>
#include <errno.h>
#include <sys/types.h>

/* This program uses spawn instead of fork/exec to create a child
 * process and uses unnamed pipes to allow the parent and child to
 * exchange communication.
 */


void main(int argc, char *argv[])  {
 pid_t child;
 int fd_count, fd_map[10];
 struct inheritance inherit;
 const char *c_argv[10], *c_envp[10];
 char buf[256];
 int nbytes;

 int c_stdin[2], c_stdout[2], c_stderr[2];  /* Pipes for child
       * communication */

 /* Create pipes to communicate with child via stdin/stdout/stderr */
 if(pipe(c_stdin)  ||
    pipe(c_stdout) ||
    pipe(c_stderr) ) {
  perror("Bad pipe");
                exit(-1);
                }

 /* Set up file descriptor map for child process */
 fd_map[0]=dup(c_stdin[0]);  /* child stdin is read end of pipe */
 fd_map[1]=dup(c_stdout[1]); /* child stdout is write end of pipe */
 fd_map[2]=dup(c_stderr[1]); /* child stderr is write end of pipe */
 fd_count=3;

 /* Close unused end of pipes for the parent */
        close(c_stdin[0]); close(c_stdout[1]); close(c_stderr[1]);

 /* Build the argument structure for child arguments.
        *  [0] is the program name */
 c_argv[0]="spawnc";
 c_argv[1]="arg1"; c_argv[2]="arg2"; c_argv[3]=NULL;

 /* Build the environment structure which defines the child's
        * environment variables */
 c_envp[0]="TEST_ENV=YES"; c_envp[1]="BPX_SHAREAS=NO"; c_envp[2]=NULL;

 /* Spawn the child process */
 child=spawnp("spawnc", fd_count, fd_map, &inherit, c_argv, c_envp);
 if(child==-1) {
  perror("Error on spawn");
  exit(-1);
  }
 else printf("Spawned %i\n", child);

 /* Test interaction with the child process */
 printf("parent: Asking child, \"what are you doing?\\n\"\n");
 strcpy(buf, "child from parent: what are you doing?\n");
 if(write(c_stdin[1], buf, sizeof(buf))==-1)  {
  perror("write stdout");
  exit(-1);
  }

 memset(buf, 0, 255);  /* Just zeroing out the buffer */
 printf("parent: reading from child now\n");
 if((nbytes=read(c_stdout[0], buf, 255))==-1) {
  perror("read error:");
  exit(-1);
  }
 printf("parent: child says, \"%s\"\n", buf);

 /* Cleanup pipes before exiting */
 close(c_stdin[1]); close(c_stdout[0]); close(c_stderr[0]);

  exit(0);
}
```

Example

The following is an example of a child program used by spawn.

```
#include <stdlib.h>
#include <stdio.h>

/* This is a sample child program used by spawn.  This program will
 * work stand-alone as well as from spawn or fork/exec. */

extern char ** environ; /* External used to access the environment
      directly instead of using getenv */

void main(int argc, char *argv[]) {
 char *e, **env=environ;  /* Used to step through the environment
      * to write out to file. */
 char buf[256]={0};
 FILE *fp=fopen("spawntest.out","w");
  int i;

 /* Print out the environment variables */
 i=0;
 fprintf(fp, "Environment:\n");
 while(e=env[i++])  fprintf(fp, "%s\n", e);
 fprintf(fp, "\n\n");

 /* Just to prove getenv works */
 fprintf(fp, "TEST_ENV envvar = %s", getenv("TEST_ENV"));

 /* Print out the command line arguments */
 i=0;
 fprintf(fp,"Args:\n");
 while(e=argv[i++]) fprintf(fp,"%s\n", e);
 fprintf(fp, "\n\n");

 /* Print out what was sent on stdin */
 fprintf(fp, "Child/parent\n");
 if(!gets(buf)) {
  ferror(stdin);
  exit(-1);
  }
 fprintf(fp, "child from parent: %i bytes,[%s]\n", strlen(buf), buf);

 /* Send something to stdout */
 printf("nothing");

 fclose(fp);
 exit(0);
}
```

Related information

- "spawn.h" on page 64
- "sys/wait.h" on page 77
- "alarm() — Set an alarm" on page 156
- "chmod() — Change the mode of a file or directory" on page 272
- "exit() — End program" on page 443
- "_exit() — End a process and bypass the cleanup" on page 445
- "fcntl() — Control open file descriptors" on page 474
- "fork() — Create a new process" on page 571
- "kill() — Send a signal to a process" on page 931
- "rexec() — Execute commands one at a time on a remote host" on page 1447
- "setuid() — Set the effective user ID" on page 1592
- "__spawn2(), __spawnp2() — Spawn a new process using enhanced inheritance structure" on page 1706
- "stat() — Get file information" on page 1720
- "times() — Get process and child process times" on page 1879
- "wait() — Wait for a child process to end" on page 1992
- "waitpid() — Wait for a specific child process to end" on page 1997

__spawn2(), __spawnp2() — Spawn a new process using enhanced inheritance structure

Standards

Standards / Extensions	C or C++	Dependencies
XPG4 XPG4.2	both	POSIX(ON)

Format

```
#include <spawn.h>

pid_t __spawn2(const char *path,
               const int fd_count,
               const int fd_map[],
               const struct __inheritance *inherit,
               const char *argv[],
               const char *envp[]);

pid_t __spawnp2(const char *file,
                const int fd_count,
                const int fd_map[],
                const struct __inheritance *inherit,
                const char *argv[],
                const char *envp[]);
```

General description

The __spawn2() and __spawnp2() functions creates a new process from the specified process image. The new process image is constructed from a regular executable file called the new process image file.

To execute a C program as a result of this call, enter the function call as follows:

```
int main (int argc, char *argv[]);
```

Where *argc* is the argument count and *argv* is an array of character pointers to the arguments themselves. In addition, the following variable:

```
extern char **environ;
```

is initialized as a pointer to an array of character pointers to the environment strings. The *argv* and `environ` arrays are each terminated by a NULL pointer. The NULL pointer terminating the *argv* array is not counted in *argc*.

Supported parameters are:

Parameter
Description

path Path name used by __spawn2() that identifies the new process image file to execute.

file Used by __spawnp2() to construct a path name that identifies the new process image file. If the file parameter contains a slash character, the file parameter shall be used as a path name for the new process image file. Otherwise, the path prefix for this file shall be obtained by a search of the directories passed as the environment variable PATH.

fd-count
Specifies the number of file descriptors the child process shall inherit. It may take values from zero to **OPEN_MAX**. Except those file descriptors designated by SPAWN_FDCLOSED, each of the child's file descriptors, x, in the range zero to *fd_count*-1 shall inherit descriptor *fd_map(x)* from the parent process.

The files from *fd_count* through **OPEN_MAX** are closed in the child process, as are any elements of *fd_map* designated as SPAWN_FDCLOSED.

fd-map If the *fd_map* parameter is NULL, *fd_count* and *fd_map* are ignored. All file descriptors except those with the `FD_CLOEXEC` or `FD_CLOFORK` attribute are inherited without reordering. File descriptors with the `FD_CLOEXEC` or `FD_CLOFORK` attribute are closed under simple inheritance.

For those file descriptors that remain open, all other attributes of the associated file descriptor object and open file description shall remain unchanged by this operation.

Directory streams open in the calling process image shall be closed in the new process image, with the effect of the closedir() operation.

If an element of *fd_map* refers to an invalid file descriptor, then the (EBADF) error status shall be posted by __spawn2() or __spawnp2().

The `FD_CLOEXEC` and `FD_CLOFORK` file descriptor attributes are never inherited.

The `FD_CLOEXEC` and `FD_CLOFORK` file descriptor attributes have no effect on inheritance when the *fd_map* parameter is not NULL.

Note: For XTI endpoints, *fd_map* must not map to a number greater than 65535 in the child process.

inherit The name of a data area that contains the inheritance structure.

The 'struct __inheritance' is defined as follows:

```
struct __inheritance {
short       flags;        -- Flags
pid_t       pgroup;       -- Process group
sigset_t    sigmask;      -- Signal mask
sigset_t    sigdefault;   -- Signals set to SIG_DFL
int         ctlttyfd;     -- Cntl tty FD for tcsetpgrp()
char        *cwdptr;      -- Pointer to the users CWD
int         cwdlen;       -- Length of the users CWD
int         acctdatalen;  -- Length of account data area
char        *acctdataptr; -- Ptr to account data area
int         umask;        -- Users UMASK
char        userid[9];    -- New A.S. user identity
char        jobname[9];   -- New A.S. jobname
int         regionsize;   -- New A.S. region size
int         timelimit;    -- New A.S. time limit
union {                   -- New A.S. memlimit # of bytes
#ifdef _LP64
unsigned long        memlimit;
#define __memlimit   memlimit_u.memlimit
#endif
#ifdef __LL
unsigned long long   memlimit_ll;
#define __memlimit_ll memlimit_u.memlimit_ll
#endif
unsigned int         memlimit_i[2];
#define __memlimit_h  memlimit_u.memlimit_i[0]
#define __memlimit_l  memlimit_u.memlimit_i[1]
double                memlimit_d;
} memlimit_u;
}
```

The `inherit.flags` effect spawn() and spawnp() as follows:

SPAWN_SETGROUP
If the SPAWN_SETGROUP flag is set in *inherit.flags*, then the child's process group shall be as specified in *inherit.pgroup*.

If the SPAWN_SETGROUP flag is set in *inherit.flags* and *inherit.pgroup* is set to SPAWN_NEWPGROUP, then the child shall be in a new process group with a process group ID equal to its process ID.

If the SPAWN_SETGROUP flag is not set in *inherit.flags*, the new child shall inherit the parent's process group ID.

SPAWN_SETSIGMASK
If the SPAWN_SETSIGMASK flag is set in *inherit.flags*, the child process shall initially have the signal mask specified in *inherit.sigmask*.

SPAWN_SETSIGDEF
If the SPAWN_SETSIGDEF flag is set in *inherit.flags*, the signals specified in *inherit.sigdefault* shall be set to their default actions in the child process. Signals set the default action in the parent process shall be set to the default action in the new process.

Signals set to be caught by the calling process shall be set to the default action in the child process.

Signals set to be ignored by the calling process shall be set to be ignored by the new process, unless otherwise specified by the SPAWN_SETSIGDEF flag being set in *inherit_flags* and the signal being indicated *inherit.sigdefault*.

SPAWN_SETTCPGRP
If the SPAWN_SETTCPGRP flag is set in *inherit.flag*, the file descriptor specified in *inherit.ctlttyfd* is used to set the controlling terminal file descriptor (tcsetpgrp()) for the child's foreground process group. The child's foreground process group is inherited from the parent, unless the SPAWN_SETGROUP flag in *inherit.flags* is set, indicating that the value specified in *inherit.pgroup* is to be used to determine the child's process group.

argv The value in the first element of *argv* should point to a file name that is associated with the process being started by the spawn2() or spawnp2() operation.

The number of bytes available for the new process's combined argument and environment lists is **ARG_MAX**.

envp The value *envp* contains the list of environmental variables that is to be passed to the specified program.

If the set-user-ID mode bit of the new process image file is set, the effective user ID of the new process image shall be set to the owner ID of the new process image file. Similarly, if the set-group-ID mode bit of the new process image file is set, the effective group ID of the new process image shall be set to the group ID of the new process image file. The real user ID, real group ID, and supplementary group IDs of the new process image shall remain the same as those of the calling process image. The effective user ID and effective group ID of the new process image shall be saved (as the saved set-user-ID and set-group-ID) for use by the setuid() function.

The new process image shall inherit the following attributes of the calling process image:
- Process group ID (unless the SPAWN_SETGROUP flag is set in *inherit.flags*).
- Session membership.
- Real user ID.
- Real group ID.
- Supplementary group IDs.
- Priority.
- Current working directory.
- File creation mask.
- Signal mask (unless the SPAWN_SETSIGMASK flag is set in *inherit.flags*).
- Signal actions specified as default (SIG_DFL).
- Signal actions specified as ignore (SIG_IGN) (except as modified by *inherit.sigdefault* and the SPAWN_SETSIGDEF flag set in *inherit.flags*).

The following are differences between the parent process and child:
- Signals set to be caught by the calling process shall be set to the default action (SIG_DFL).
- The process and system utilization times for the child are set to zero.
- Any file locks previously set by the parent are not inherited by the child.
- The child process has no alarms set and has no interval timers set.
- The child has no pending signals.
- Memory mappings established by the parent are not inherited by the child.

If the process image was read from a writable file system, then upon successful completion, the __spawn2() and __spawnp2() functions will mark for update the *st_time* field of the new process image file.

If the __spawn2() or __spawnp2() function is successful, the new child process image file shall be opened with all the effects of the open() function.

All the following inherit flags are used by __spawn2() and __spawnp2():

SPAWN_SETCWD
Specifies the Current Working Directory that the child process will run when first created. This will override the CWD that would normally be set up or propagated by the child process.

SPAWN_SETUMASK
Specifies the UMASK that the child process will run when first created. This will override the UMASK that would normally be set up in the child process. The invoker must have superuser privileges to specify UMASK.

SPAWN_SETUSERID
When this flag is set, this attribute will be the equivalent of the _BPX_USERID environment variable. If specified, the invoking userid will be checked for daemon authority. If the invoker is authorized and the userid is valid, the child process will be created with RACF identity and POSIX permissions associated with the input USERID. If not authorized or the userid is invalid, the __spawn2() or __spawnp2() function will fail. If the USERID value is specified, any value in _BPX_USERID will be ignored.

SPAWN_SETREGIONSZ
Specifies the number of megabytes the child process will have available for

private storage. The authority/ranges required will be as per RLIMIT_AS rules. Unless the invoker has superuser privileges, the region size range will be checked and if it exceeds the hard limit, the __spawn2() or __spawnp2() function will fail. This value will override RLIMIT_AS or the normal spawn propagation rules.

SPAWN_SETTIMELIMIT
Specifies the number of seconds of CPU time that is allowed by the child process before receiving a SIGXCPU signal. Unless the invoker has superuser privileges, the time limit range will be checked and if it exceeds the hard limit, the __spawn2() or __spawnp2() function will fail. This value will override the RLIMIT_CPU or the normal spawn propagation rules.

SPAWN_SETACCTDATA
Specifies account data of the child process. The format and length will be as per the _BPX_ACCT_DATA environmental variable. No special authority is needed to change account data. This will override the target userid's default account data and any value specified on the _BPX_ACCT_DATA will be ignored.

SPAWN_SETJOBNAME
When this flag is set, it is the equivalent of the _BPX_JOBNAME environment variable. If specified, the invoking userid will be checked for superuser authority. If the invoker is authorized and the jobname is valid, the child process will be created with the specified jobname. If not authorized or the jobname is invalid, __spawn2() or __spawnp2() will ignore the JOBNAME attribute and continue. If the JOBNAME value is specified, any value in _BPX_JOBNAME will be ignored.

SPAWN_MUSTBELOCAL
When this flag is set, it is the equivalent of the _BPX_SHAREAS=MUST environment variable. If specified, the system will attempt a local spawn, otherwise the request will fail.

SPAWN_SETMEMLIMIT
When this flag is set and not doing a local spawn, inherit.memlimit_u determines the maximum amount of bytes the child address space is allowed to obtain above the 2-gigabyte bar. If the specified value exceeds the current hard limit, the invoking userid must have appropriate privileges, otherwise the request will fail.

SPAWN_PROCESS_INITTAB
If this flag is set, spawn attempts to read the /etc/inittab file and process the entries found there. This processing involves the spawning of child shell processes to run each of the commands identified in the file. Only the `SPAWN_SETSIGMASK` flag can be set in combination with this flag. All other flags will be ignored. Use of this flag implies that only file descriptors 0, 1, and 2 will be initially opened in the child process. File descriptor 0 will be initially opened as /dev/null, while file descriptors 1 and 2 will initially opened as /etc/log. The `fd_count` and `fd_map` parameters will be ignored. This flag is currently restricted to the /usr/sbin/init process. See *z/OS UNIX System Services Planning*for more information on the /etc/inittab support.

For more information on the use of inheritance structure flags, see *z/OS UNIX System Services Programming: Assembler Callable Services Reference, SA23-2281*.

Returned value

If successful, __spawn2() or __spawnp2() returns the process ID of the child process to the parent.

If unsuccessful, __spawn2() or __spawnp2() returns -1 to the parent process, no child is created, and they set errno to one of the following values:

Error Code
: **Description**

E2BIG
: The number of bytes used by the argument and environment list of the new process image is greater than the system-imposed limit of **ARG_MAX** bytes.

EACCES
: Search permission is denied for a directory in the path of the new process image file or the new process image file denies execution permission, or the the new process image file is not a regular file and the implementation does not support execution of files of its type.

EAGAIN
: The system lacked the necessary resources to create another process, or the system-imposed limit on the total number of processes under execution by a single user would be exceeded. The resources required to let another process be created are not available, or you have already reached the maximum number of processes or UIDs you are allowed to create. This error will also be generated if _BPX_USERID or INHEUSERID was specified and the username was not defined to SAF with a segment.

EBADF
: An entry in the *fd_map* array refers to an invalid file descriptor or the controlling terminal file descriptor specified in the __inheritance structure.

EFAULT
: The system detected an invalid address while attempting to use a parameter of the call.

EINVAL
: One or more of the following conditions were detected:

- The username that was specified on the _BPX_USERID environment variable has an incorrect length.
- An attribute that was specified in the inheritance structure (BPXYINHE) is not valid or contains an unsupported value.
- The version number that was specified for the inheritance structure (BPXYINHE) is not valid.
- The inheritance structure length that was specified by the Inherit_area_len parameter or within the inheritance structure does not contain a length that is appropriate for the BPXYINHE version.
- The process group ID that was specified in the inheritance structure is less than zero or has some other unsupported value.

The following reason codes can accompany the return code: JROK, JRUserNameLenError, JRJsRacXtr, JRInheUserid, JRInheRegion, JRInheCPUTime, JRInheDynamber, JRInheAccountData, JRInheCWD, JRInheSetPgrp, JRInheVersion, and JRInheLength.

ELOOP
A loop exists in symbolic links encountered during resolution of the file name argument. This error is issued if more than 8 symbolic links are detected.

EMVSERR

The spawn failed for the following reason: Local spawn not allowed. Either the environment variable _BPX_SHAREAS was set to MUST (_BPX_SHAREAS=MUST), or the __inheritance structure specified SPAWN_MUSTBELOCAL.

The following reason code can accompany the return code: JRLocalSpawnNotAllowed.

EMVSSAF2ERR
The executable file is a set-user-ID or set-group-ID file and the owner's UID or GID is not defined to the Security Authorization Facility (SAF).

ENAMETOOLONG
The length of the path or file parameter, or an element of the environmental variable PATH prefixed to a file, exceeds **PATH_MAX**, or a path name component is longer than **NAME_MAX** and {_POSIX_N_TRUNC_} is in effect for that file.

ENOENT
One or more components of the path name of the new process image file do not exist or the path or file parameter is empty.

ENOEXEC
The new process image file has the appropriate access permission, but is not in the proper format.

Note:

Reason codes further qualify the errno. For most of the reason codes, see *z/OS UNIX System Services Messages and Codes*.

For ENOEXEC, the reason codes are:

Reason Code	Explanation
X'xxxx0C27'	The target HFS file is not in the correct format to be an executable file.
X'xxxx0C31'	The target HFS file is built at a level that is higher than that supported by the running system.

ENOMEM
The new process requires more memory than is permitted by the hardware or operating system.

ENOTDIR
A component of the path prefix of the new process image file is not a directory.

ENOTTY
tcsetpgrp() failed for the specified controlling terminal file descriptor in __inheritance structure. The failure occurred because the calling process does not have a controlling terminal, or the specified file descriptor is not associated with the controlling terminal, or the controlling terminal is no longer associated with the session of the calling process.

EPERM

The spawn failed for one of the following reasons:

- The spawned process is not a process group leader.
- The _BPX_USERID environment variable was specified, and the invoker does not have appropriate privileges to change the MVS identity.
- The invoker does not have the appropriate privileges to change one or more of the attributes specified in the inheritance structure (BPXYINHE).

The following reason codes can accompany the return code: JROK, JRNoChangeIdentity, JRInheUserid, JRInheRegion, JRInheCPUTime, JRInheUmask, JRInheCWD and JRInheMemLimit.

ESRCH

The process group ID specified in the __inheritance structure is not that of s process group in the calling process's session.

Related information

- "spawn.h" on page 64
- "sys/wait.h" on page 77
- "alarm() — Set an alarm" on page 156
- "chmod() — Change the mode of a file or directory" on page 272
- "exit() — End program" on page 443
- "_exit() — End a process and bypass the cleanup" on page 445
- "fcntl() — Control open file descriptors" on page 474
- "fork() — Create a new process" on page 571
- "kill() — Send a signal to a process" on page 931
- "rexec() — Execute commands one at a time on a remote host" on page 1447
- "setuid() — Set the effective user ID" on page 1592
- "spawn(), spawnp() — Spawn a new process" on page 1693
- "stat() — Get file information" on page 1720
- "times() — Get process and child process times" on page 1879
- "wait() — Wait for a child process to end" on page 1992
- "waitpid() — Wait for a specific child process to end" on page 1997

sprintf() — Format and write data to buffer

The information for this function is included in "fprintf(), printf(), sprintf() — Format and write data" on page 588.

sqrt(), sqrtf(), sqrtl() — Calculate square root

Standards

Standards / Extensions	C or C++	Dependencies
ISO C POSIX.1 XPG4 XPG4.2 ISO/ANSI C++ C99 Single UNIX Specification, Version 3 C++ TR1 C99	both	

Format

```
#include <math.h>

double sqrt(double x);
float sqrt(float x);                    /* C++ only */
long double sqrt(long double x);        /* C++ only */
float sqrtf(float x);
long double sqrtl(long double x);
```

General description

Calculates the square root of *x*.

Note: These functions work in both IEEE Binary Floating-Point and hexadecimal floating-point formats. See "IEEE binary floating-point" on page 94 for more information about IEEE Binary Floating-Point.

Returned value

If successful, returns the square root of *x*.

If *x* is negative, the function sets errno to EDOM, and returns 0. If the correct value would cause underflow, zero is returned and the value ERANGE is stored in errno.

Special behavior for IEEE

If *x*< -0, the function returns NaNQ and sets errno to EDOM.

If *x* is a NaN, a NaN will be returned.

If *x* is ±0 or +INF, *x* will be returned.

If *x* is -INF, a EDOM will be set, and NaNQ will be returned.

Example

CELEBS30

```
/* CELEBS30

   This example computes the square root of the quantity passed
   as the first argument to main.
   It prints an error message if you pass a negative value.

 */
#include <stdio.h>
#include <stdlib.h>
#include <math.h>

int main(int argc, char ** argv)
{
  char * rest;
  double value;

  if ( argc != 2 )
    printf( "Usage: %s value\n", argv[0] );
  else
  {
    value = strtod( argv[1], &rest );
    if ( value < 0.0 )
       printf( "sqrt of a negative number\n" );
    else
       printf("sqrt( %f ) = %f\n", value, sqrt( value ));
  }
}
```

Output

If the input is 45, then the output should be:

```
sqrt( 45.000000 ) = 6.708204
```

Related information

- “math.h” on page 44
- “exp(), expf(), expl() — Calculate exponential function” on page 448
- “hypot(), hypotf(), hypotl() — Calculate the square root of the squares of two arguments” on page 821
- “log(), logf(), logl() — Calculate natural logarithm” on page 990
- “log10(), log10f(), log10l() — Calculate base 10 logarithm” on page 1000
- “pow(), powf(), powl() — Raise to power” on page 1193

sqrtd32(), sqrtd64(), sqrtd128() — Calculate square root

Standards

Standards / Extensions	C or C++	Dependencies
C/C++ DFP	both	z/OS V1.8

Format

```
#define __STDC_WANT_DEC_FP__
#include <math.h>

_Decimal32  sqrtd32(_Decimal32 x);
_Decimal64  sqrtd64(_Decimal64 x);
_Decimal128 sqrtd128(_Decimal128 x);
_Decimal32  sqrt(_Decimal32 x);      /* C++ only */
_Decimal64  sqrt(_Decimal64 x);      /* C++ only */
_Decimal128 sqrt(_Decimal128 x);     /* C++ only */
```

General description

Calculates the square root of *x*.

Notes:

1. To use IEEE decimal floating-point, the hardware must have the Decimal Floating-Point Facility installed.
2. These functions work in IEEE decimal floating-point format. See "IEEE Decimal Floating-Point" for more information.

Returned value

If successful, returns the square root of *x*.

If the correct value would cause underflow, zero is returned and the value ERANGE is stored in errno.

If *x* < -0, the function returns NaNQ and sets errno to EDOM.

If *x* is a NaN, a NaN will be returned.

If *x* is ±0 or +INF, x will be returned.

If *x* is -INF, the function returns NaNQ and sets errno to EDOM.

Example

```
/* CELEBS71

   This example illustrates the sqrtd32() function, along with
   the strtod32() function.

   This example computes the square root of the quantity passed
   as the first argument to main.
   It prints an error message if you pass a negative value.

*/

#define  __STDC_WANT_DEC_FP__
#include <math.h>
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char ** argv)
{
  char        *rest;
  _Decimal32   value;

  if (argc != 2)
  {
    printf("Usage: %s value\n", argv[0]);
  }
  else
  {
    value = strtod32(argv[1], &rest);

    if (value < 0.0DF)
      printf("sqrt of a negative number\n");
    else
      printf("sqrt(%Hf) = %Hf\n", value, sqrtd32(value));
  }
}
```

Related information

- "math.h" on page 44
- "expd32(), expd64(), expd128() — Calculate exponential function" on page 449
- "logd32(), logd64(), logd128() — Calculate natural logarithm" on page 994
- "log10d32(), log10d64(), log10d128() — Calculate base 10 logarithm" on page 1002
- "powd32(), powd64(), powd128() — Raise to power" on page 1194
- "sqrt(), sqrtf(), sqrtl() — Calculate square root" on page 1713

srand() — Set seed for rand() function

Standards

Standards / Extensions	C or C++	Dependencies
POSIX.1 XPG4 XPG4.2 Single UNIX Specification, Version 3	both	

Format

```
#include <stdlib.h>

void srand(unsigned int seed);
```

General description

srand() uses its argument *seed* as a seed for a new sequence of pseudo-random numbers to be returned by subsequent calls to rand(). If srand() is not called, the rand() seed is set as if `srand(1)` was called at program start. Any other value for *seed* sets the generator to a different starting point. The rand() function generates pseudo-random numbers.

Some people find it convenient to use the return value of the time() function as the argument to srand(), as a way to ensure random sequences of random numbers.

Returned value

srand() returns no values.

Example

CELEBS31

```
/* CELEBS31

   This example first calls &srand. with a value other than 1 to
   initiate the random value sequence.
   Then the program computes 5 random values for the array of
   integers called ranvals.
   If you repeat this code exactly, then the same sequence of
   random values will be generated.

 */
#include <stdlib.h>
#include <stdio.h>

int main(void)
{
   int i, ranvals[5];

   srand(17);
   for (i = 0; i < 5; i++)
   {
      ranvals[i] = rand();
      printf("Iteration %d ranvals [%d] = %d\n", i+1, i, ranvals[i]);
   }
}
```

Output

```
Iteration 1 ranvals [0] = 24107
Iteration 2 ranvals [1] = 16552
Iteration 3 ranvals [2] = 12125
Iteration 4 ranvals [3] = 9427
Iteration 5 ranvals [4] = 13152
```

Related information

- “stdlib.h” on page 70
- “rand() — Generate random number” on page 1374
- “rand_r() — Pseudo-random number generator” on page 1375

srandom() — Use seed to initialize generator for random()

Standards

Standards / Extensions	C or C++	Dependencies
XPG4.2 Single UNIX Specification, Version 3	both	

Format

```
#define _XOPEN_SOURCE_EXTENDED 1
#include <stdlib.h>

void srandom(unsigned seed);
```

General description

The srandom() function initializes the calling thread's current state array for the random() function using the value of *seed*.

Returned value

srandom() returns no values.

Related information

- "stdlib.h" on page 70
- "drand48() — Pseudo-random number generator" on page 402
- "initstate() — Initialize generator for random()" on page 867
- "rand() — Generate random number" on page 1374
- "rand_r() — Pseudo-random number generator" on page 1375
- "random() — A better random-number generator" on page 1375
- "setstate() — Change generator for random()" on page 1590

srand48() — Pseudo-random number initializer

Standards

Standards / Extensions	C or C++	Dependencies
XPG4 XPG4.2 Single UNIX Specification, Version 3	both	

Format

```
#define _XOPEN_SOURCE
#include <stdlib.h>

void srand48(long int seedval);
```

General description

The drand48(), erand48(), jrand48(), lrand48(), mrand48() and nrand48() functions generate uniformly distributed pseudo-random numbers using a linear congruential algorithm and 48-bit integer arithmetic.

The lcong48(), seed48(), and srand48() functions are initialization functions, one of which should be invoked before either the drand48(), lrand48() or mrand48() function is called.

The drand48(), lrand48() and mrand48() functions generate a sequence of 48-bit integer values, X(i), according to the linear congruential formula:

```
X(n+1) = (aX(n) + c)mod(2**48)     n>=0
```

The initial values of X, a, and c are:

```
X(0)= 1
a   = 5deece66d (base 16)
c   = b         (base 16)
```

C/370 provides storage to save the most recent 48-bit integer value of the sequence, X(i). This storage is shared by the drand48(), lrand48() and mrand48() functions. The srand48() function is used to reinitialize the most recent 48-bit value in this storage. The srand48() function replaces the high-order (leftmost) 32 bits of this storage with *seedval* argument value. The srand48() function replaces the low-order 16 bits of this storage with the value **330E** (base 16).

The values a and c, may be changed by calling the lcong48() function. The srand48()function restores the initial values of a and c.

Special behavior for z/OS UNIX Services: You can make the srand48() function and other functions in the drand48 family thread-specific by setting the environment variable _RAND48 to the value THREAD before calling any function in the drand48 family.

If you do not request thread-specific behavior for the drand48 family, C/370 serializes access to the storage for X(n), a and c by functions in the drand48 family when they are called by a multithreaded application.

If thread-specific behavior is requested, calls to the drand48(), lrand48() and mrand48() functions from thread t generate a sequence of 48-bit integer values, X(t,i), according to the linear congruential formula:

```
X(t,n+1) = (a(t)X(t,n) + c(t))mod(2**48)     n>=0
```

C/370 provides thread-specific storage to save the most recent 48-bit integer value of the sequence, X(t,i). When the srand48()function is called from thread t, it reinitializes the most recent 48-bit value in this storage. The srand48() function replaces the high-order (leftmost) 32 bits of this storage with *seedval* argument value. The srand48() function replaces the low-order 16 bits of this storage with the value **330E** (base 16).

The values of a(t) and c(t) may be changed by calling the lcong48() function from thread t. When the srand48()function is called from this thread it restores the initial values of a(t) and c(t) for the thread which are:

```
a(t)   = 5deece66d (base 16)
c(t)   = b         (base 16)
```

Returned value

srand48() returns no values.

srand48() returns after it has used the value of the argument *seedval* to reinitialize storage for the most recent 48-bit integer value in the sequence, X(i), and has restored the initial values of a and c.

Special behavior for z/OS UNIX Services: If thread-specific behavior is requested for the drand48 family and the srand48() function is called on thread `t`, it uses the value of the argument *seedval* to reinitialize storage for the most recent 48-bit integer value in the sequence, X(t,i), for the thread. It also restores the initial values of a(t) and c(t) for the thread. Then it returns.

Related information

- "stdlib.h" on page 70
- "drand48() — Pseudo-random number generator" on page 402
- "erand48() — Pseudo-random number generator" on page 426
- "jrand48() — Pseudo-random number generator" on page 928
- "lcong48() — Pseudo-random number initializer" on page 939
- "lrand48() — Pseudo-random number generator" on page 1010
- "mrand48() — Pseudo-random number generator" on page 1101
- "nrand48() — Pseudo-random number generator" on page 1148
- "seed48() — Pseudo-random number initializer" on page 1467

sscanf() — Read and format data from buffer

The information for this function is included in "fscanf(), scanf(), sscanf() — Read and format data" on page 625.

stat() — Get file information

Standards

Standards / Extensions	C or C++	Dependencies
POSIX.1 XPG4 XPG4.2 Single UNIX Specification, Version 3	both	

Format

```
#define _POSIX_SOURCE
#include <sys/stat.h>

int stat(const char *__restrict__ pathname, struct stat *__restrict__ info);
```

General description

Gets status information about a specified file and places it in the area of memory pointed to by the *info* argument. The process does not need permissions on the file itself, but must have search permission on all directory components of the *pathname*. If the named file is a symbolic link, stat() resolves the symbolic link. It also returns information about the resulting file.

The information is returned as shown in the following `stat` structure table, as defined in the sys/stat.h header file.

Table 51. Values Returned in stat Structure

Value	Description
mode_t st_mode	A bit string indicating the permissions and privileges of the Symbols are defined in the sys/stat.h header file to refer to bits in a `mode_t` value; these symbols are listed in "chmod() — Change the mode of a file or directory" on page 272.
ino_t st_ino	The serial number of the file.
dev_t st_dev	The numeric ID of the device containing the file.
nlink_t st_nlink	The number of links to the file.
uid_t st_uid	The numeric user ID (UID) of the file's owner.
gid_t st_gid	The numeric group ID (GID) of the file's group.
off_t st_size	For regular files, the file's size in bytes. For other kinds of files, the value of this field is unspecified.
time_t st_atime	The most recent time the file was accessed.
time_t st_ctime	The most recent time the status of the file was changed.
time_t st_mtime	The most recent time the contents of the file were changed.

Values for `time_t_` are given in terms of seconds since epoch.

stat() updates the time-related fields before putting information in the `stat` structure.

You can examine properties of a `mode_t` value from the `st_mode` field using a collection of macros defined in the sys/modes.h header file. If *mode* is a `mode_t` value, and *genvalue* is an `unsigned int` value from the stat structure, then:

S_ISBLK(*mode***)**
Is nonzero for block special files.

S_ISCHR(*mode***)**
Is nonzero for character special files.

S_ISDIR(*mode***)**
Is nonzero for directories.

S_ISEXTL(*mode,genvalue***)**
Is nonzero for external links.

S_ISFIFO(*mode***)**
Is nonzero for pipes and FIFO special files.

S_ISLNK(*mode***)**
Is nonzero for symbolic links.

S_ISREG(*mode***)**
Is nonzero for regular files.

S_ISSOCK(*mode***)**
Is nonzero for sockets.

If stat() successfully determines this information, it stores it in the area indicated by the *info* argument. The size of the buffer determines how much information is stored; data that exceeds the size of the buffer is truncated.

Large file support for z/OS UNIX files: Large z/OS UNIX files are supported automatically for AMODE 64 C/C++ applications. AMODE 31 C/C++ applications must be compiled with the option LANGLVL(LONGLONG) and define the _LARGE_FILES feature test macro before any headers are included to enable this function to operate on z/OS UNIX files that are larger than 2 GB in size. File size and offset fields are enlarged to 63 bits in width. Therefore, any other function operating on the file is required to define the _LARGE_FILES feature test macro as well.

Returned value

If successful, stat() returns 0.

If unsuccessful, stat() returns -1 and sets errno to one of the following values:

Error Code
Description

EACCES
The process does not have search permission on some component of the *pathname* prefix.

EINVAL
info is a NULL pointer.

EIO **Added for XPG4.2:** An error occurred while reading from the file system.

ELOOP
A loop exists in symbolic links encountered during resolution of the *pathname* argument. This error is returned if more than POSIX_SYMLOOP (defined in the limits.h header file) symbolic links are encountered during resolution of the *pathname* argument.

ENAMETOOLONG
pathname is longer than PATH_MAX characters, or some component of *pathname* is longer than NAME_MAX characters while _POSIX_NO_TRUNC is in effect. For symbolic links, the length of the path name string substituted for a symbolic link exceeds PATH_MAX. The PATH_MAX and NAME_MAX values can be determined with pathconf().

ENOENT
There is no file named *pathname*, or *pathname* is an empty string.

ENOTDIR
A component of the *pathname* prefix is not a directory.

EOVERFLOW
The file size in bytes or the number of blocks allocated to the file or the file serial number cannot be represented correctly in the structure pointed to by info.

Note: Environment variable _EDC_EOVERFLOW can be used to control behavior of the stat() function with respect to detecting an EOVERFLOW condition for z/OS UNIX files. By default, the stat() function will not set EOVERFLOW when the file size can not be represented correctly in structure pointed to by buf. When _EDC_EOVERFLOW is set to YES, the stat() function will check for an overflow condition.

Example

CELEBS33

```
/* CELEBS33

   This example gets status information about a file.

 */
#define _POSIX_SOURCE
#include <sys/types.h>
#include <sys/stat.h>
#undef _POSIX_SOURCE
#include <stdio.h>
#include <time.h>

main() {
  struct stat info;

  if (stat("/", &info) != 0)
    perror("stat() error");
  else {
    puts("stat() returned the following information about root f/s:");
    printf("  inode:   %d\n",   (int) info.st_ino);
    printf(" dev id:   %d\n",   (int) info.st_dev);
    printf("   mode:   %08x\n",       info.st_mode);
    printf("  links:   %d\n",         info.st_nlink);
    printf("    uid:   %d\n",   (int) info.st_uid);
    printf("    gid:   %d\n",   (int) info.st_gid);
    printf("created:   %s",           ctime(&info.st_createtime));
  }
}
```

Output

```
stat() returned the following information about root f/s:
  inode:   0
 dev id:   1
   mode:   010001ed
  links:   11
    uid:   0
    gid:   500
created:   Fri Jun 16 10:07:55 2006
```

Related information

- “sys/stat.h” on page 75
- “sys/types.h” on page 75
- “chmod() — Change the mode of a file or directory” on page 272
- “chown() — Change the owner or group of a file or directory” on page 275
- “creat() — Create a new file or rewrite an existing one” on page 340
- “dup() — Duplicate an open file descriptor” on page 404
- “fcntl() — Control open file descriptors” on page 474
- “fstat() — Get status information about a file” on page 651
- “link() — Create a link to a file” on page 970
- “lstat() — Get status of file or symbolic link” on page 1022
- “mkdir() — Make a directory” on page 1068
- “mkfifo() — Make a FIFO special file” on page 1071
- “open() — Open a file” on page 1152
- “pipe() — Create an unnamed pipe” on page 1179
- “read() — Read from a file or socket” on page 1376
- “readlink() — Read the value of a symbolic link” on page 1387
- “remove() — Delete file” on page 1428
- “rexec() — Execute commands one at a time on a remote host” on page 1447
- “symlink() — Create a symbolic link to a path name” on page 1795
- “unlink() — Remove a directory entry” on page 1962
- “utime() — Set file access and modification times” on page 1967
- “write() — Write data on a file or socket” on page 2085

statvfs() — Get file system information

Standards

Standards / Extensions	C or C++	Dependencies
XPG4.2 Single UNIX Specification, Version 3	both	

Format

```
#define _XOPEN_SOURCE_EXTENDED 1
#include <sys/statvfs.h>

int statvfs(const char *_restrict_ pathname, struct statvfs *_restrict_ fsinfo);
```

General description

The statvfs() function obtains information about the file system containing the file named by *pathname* and stores it in the area of memory pointed to by the *fsinfo* argument. The process does not need permissions on the file itself, but must have search permission on all directory components of the *pathname*.

The information is returned in the following `statvfs` structure, as defined in the sys/statvfs.h header file.

Table 52. Values Returned in statvfs Structure

Value	Description
char f_OEcbid[4]	The structure acronym (eye catcher).
int f_OEcblen	The length of the structure.
unsigned long f_bsize	The file system block size.
unsigned long f_blocks	The total number of blocks on the file system in units of f_frsize.
unsigned long f_OEusedspace	The allocated space in block size units.
unsigned long f_bavail	The number of free blocks available to non-privileged process.
unsigned long f_fsid	The file system ID.
unsigned long f_flag	A bit string indicating file system status.
int f_OEmaxfilesizehw	The high word of maximum file size.
unsigned long f_OEmaxfilesizelw	The low word of maximum file size.
unsigned long f_frsize	The fundamental file system block size.
unsigned long f_bfree	The total number of free blocks.
unsigned long f_files	The total number of file serial numbers.
unsigned long f_ffree	The total number of free file serial numbers.
unsigned long f_favail	The number of file serial numbers available to non-privileged process.
unsigned long f_namemax	The maximum file name length.
unsigned long f_OEinvarsec	The number of seconds the file system will remain unchanged.

The following flags can be returned in the f_flag member:

ST_RDONLY
read-only file system

ST_NOSUID
setuid/setgid bits ignored by exec

ST_OEEXPORTED
file system is exported

If statvfs() successfully determines this information, it stores in the area indicated by the *fsinfo* argument. The size of the buffer determines how much information is stored; data that exceeds the size of the buffer is truncated.

Returned value

If successful, statvfs() returns 0.

If unsuccessful, statvfs() returns -1 and sets errno to one of the following values:

Error Code
Description

EACCES
The process does not have search permission on some component of the *pathname* prefix.

EINTR
A signal was caught during the execution of the function.

EIO An I/O error has occurred while reading the file system.

ELOOP
A loop exists in symbolic links encountered during resolution of the *pathname* argument. This error is issued if more than the system-defined limit of symbolic links, 8, are detected in the resolution of *pathname*.

ENAMETOOLONG
The length of the *pathname* exceeds **PATH_MAX** or a component of *pathname* is longer than **NAME_MAX**.

ENOENT
There is no file named *pathname*, or *pathname* is an empty string.

ENOTDIR
A component of the *pathname* prefix is not a directory.

Example

```
#include <sys/statvfs.h>
#include <stdio.h>

main() {
  int fd;
  struct statvfs buf;

  if (statvfs(".", &buf) == -1)
    perror("statvfs() error");
  else {
    printf("each block is %d bytes big\n", fs,
           buf.f_bsize);
    printf("there are %d blocks available out of a total of %d\n",
           buf.f_bavail, buf.f_blocks);
    printf("in bytes, that's %.0f bytes free out of a total of %.0f\n
          ((double)buf.f_bavail * buf.f_bsize),
          ((double)buf.f_blocks * buf.f_bsize));
  }
}
```

Output

```
each block is 4096 bytes big
there are 2089 blocks available out of a total of 2400
in bytes, that's 8556544 bytes free out of a total of 9830400
```

Related information

- “sys/statvfs.h” on page 75
- “chmod() — Change the mode of a file or directory” on page 272
- “chown() — Change the owner or group of a file or directory” on page 275
- “creat() — Create a new file or rewrite an existing one” on page 340
- “dup() — Duplicate an open file descriptor” on page 404
- “exec functions” on page 436
- “fcntl() — Control open file descriptors” on page 474
- “link() — Create a link to a file” on page 970
- “mknod() — Make a directory or file” on page 1074
- “open() — Open a file” on page 1152
- “pipe() — Create an unnamed pipe” on page 1179
- “read() — Read from a file or socket” on page 1376
- “time(),time64() — Determine current UTC time” on page 1878
- “unlink() — Remove a directory entry” on page 1962
- “utime() — Set file access and modification times” on page 1967
- “write() — Write data on a file or socket” on page 2085

step() — Pattern match with regular expression

Standards

Standards / Extensions	C or C++	Dependencies
XPG4 XPG4.2	both	

Format

```
#define _XOPEN_SOURCE
#include <regexp.h>

int step(const char *string, const char *expbuf);

extern char *loc1, *loc2;
```

General description

Restriction: This function is not supported in AMODE 64.

The step() function attempts to match an input string of characters with the compiled regular expression which was obtained by an earlier call to compile().

The first parameter *string* is a pointer to a string of characters to be checked for a match.

expbuf is the pointer to the regular expression which was previously obtained by a call to compile().

Usage notes

1. The external variables *cirf*, *sed*, and *nbra* are reserved.
2. The application must provide the proper serialization for the compile(), step(), and advance() functions if they are run under a multithreaded environment.
3. The compile(), step() and advance() functions are provided for historical reasons. These functions were part of the Legacy Feature in Single UNIX Specification, Version 2. They have been withdrawn and are not supported as part of Single UNIX Specification, Version 3. New applications should use the newer functions fnmatch(), glob(), regcomp(), and regexec(), which provide full internationalized regular expression functionality compatible with ISO POSIX.2 standard.

Returned value

If some substring of *string* matches the regular expression in *expbuf*, step() returns nonzero.

If there is no match, step() returns 0.

If there is a match, step() sets two external pointers, as follows:

- The variable *loc1* points to the first character that matched the regular expression.
- The variable *loc2* points to the character after the last character that matched the regular expression.

For example, if the regular expression matches the entire input *loc1* will point to the first character of *string* and *loc2* will point to the NULL at the end of *string*.

Related information

- “regexp.h” on page 61
- “advance() — Pattern match given a compiled regular expression” on page 141
- “compile() — Compile regular expression” on page 301
- “fnmatch() — Match file name or path name” on page 564
- “glob() — Generate path names matching a pattern” on page 807
- “regcomp() — Compile regular expression” on page 1415
- “regexec() — Execute compiled regular expression” on page 1421

strcasecmp() — Case-insensitive string comparison

Standards

Standards / Extensions	C or C++	Dependencies
XPG4.2 Single UNIX Specification, Version 3	both	

Format

```
#define  _XOPEN_SOURCE_EXTENDED 1
#include <strings.h>

int strcasecmp(const char *string1, const char *string2);
```

General description

The strcasecmp() function compares, while ignoring differences in case, the string pointed to by *string1* to the string pointed to by *string2*.

The string arguments to the function must contain a NULL character (\0) marking the end of the string.

The strcasecmp() function is locale-sensitive.

Returned value

strcasecmp() returns a value indicating the relationship between the strings, while ignoring case, as follows:

Value	Meaning
< 0	String pointed to by *string1* is less than string pointed to by *string2*.
= 0	String pointed to by *string1* is equal to string pointed to by *string2*.
> 0	String pointed to by *string1* is greater than string pointed to by *string2*.

There are no errno values defined.

Related information

- "strings.h" on page 72
- "setlocale() — Set locale" on page 1552
- "strcspn() — Compare strings" on page 1735
- "strncasecmp() — Case-insensitive string comparison" on page 1749

strcat() — Concatenate strings

Standards

Standards / Extensions	C or C++	Dependencies
ISO C POSIX.1 XPG4 XPG4.2 C99 Single UNIX Specification, Version 3	both	

Format

```
#include <string.h>

char *strcat(char * __restrict__string1, const char * __restrict__string2);
```

General description

The strcat() built-in function concatenates *string2* with *string1* and ends the resulting string with the NULL character. In other words, strcat() appends a copy of the string pointed to by *string2*—including the terminating NULL byte— to the end of a string pointed to by *string1*, with its last byte (that is, the terminating NULL byte of *string1*) overwritten by the first byte of the appended string.

Do not use a literal string for a *string1* value, although *string2* may be a literal string.

If the storage of *string1* overlaps the storage of *string2*, the behavior is undefined.

Returned value

Returns the value of *string1*, the concatenated string.

Example

CELEBS34

```
/* CELEBS34

   This example creates the string "computer program" using strcat().

 */
#include <stdio.h>
#include <string.h>

#define SIZE 40


int main(void)
{
  char buffer1[SIZE] = "computer";
  char * ptr;

  ptr = strcat( buffer1, " program" );
  printf( "buffer1 = %s\n", buffer1 );

}
```

Output

```
buffer1 = computer program
```

Related information

- "string.h" on page 72
- "strchr() — Search for character"
- "strcmp() — Compare strings" on page 1731
- "strcpy() — Copy string" on page 1734
- "strcspn() — Compare strings" on page 1735
- "strncat() — Concatenate strings" on page 1750

strchr() — Search for character

Standards

Standards / Extensions	C or C++	Dependencies
ISO C XPG4 XPG4.2 C99 Single UNIX Specification, Version 3	both	

Format

C:

```
#include <string.h>

char *strchr(const char *string, int c);
```

C++:

```
#include <string.h>

const char *strchr(const char *string, int c);
      char *strchr(char *string, int c);
```

General description

The strchr() built-in function finds the first occurrence of *c* converted to char, in the string **string*. The character *c* can be the NULL character (`\0`); the ending NULL character of *string* is included in the search.

The strchr() function operates on NULL-terminated strings. The string argument to the function *must* contain a NULL character (`\0`) marking the end of the string.

Returned value

If successful, strchr() returns a pointer to the first occurrence of *c* (converted to a character) in *string*.

If the character is not found, strchr() returns a NULL pointer.

Example

CELEBS35

```
/* CELEBS35

   This example finds the first occurrence of the character p in
   "computer program".

 */
#include <stdio.h>
#include <string.h>

#define SIZE 40

int main(void)
{
  char buffer1[SIZE] = "computer program";
  char * ptr;
  int    ch = 'p';

  ptr = strchr( buffer1, ch );
  printf( "The first occurrence of %c in '%s' is '%s'\n",
            ch, buffer1, ptr );

}
```

Output

```
The first occurrence of p in 'computer program' is 'puter program'
```

Related information

- “string.h” on page 72
- “memchr() — Search buffer” on page 1061
- “strcat() — Concatenate strings” on page 1728
- “strcmp() — Compare strings” on page 1731
- “strcpy() — Copy string” on page 1734
- “strcspn() — Compare strings” on page 1735
- “strncmp() — Compare strings” on page 1751
- “strpbrk() — Find characters in string” on page 1754
- “strrchr() — Find last occurrence of character in string” on page 1759

- "strspn() — Search string" on page 1761

strcmp() — Compare strings

Standards

Standards / Extensions	C or C++	Dependencies
ISO C POSIX.1 XPG4 XPG4.2 C99 Single UNIX Specification, Version 3	both	

Format

```
#include <string.h>

int strcmp(const char *string1, const char *string2);
```

General description

The strcmp() built-in function compares the string pointed to by *string1* to the string pointed to by *string2* The string arguments to the function must contain a NULL character (`\0`) marking the end of the string.

The relation between the strings is determined by subtracting: *string1*[i] - *string2*[i], as *i* increases from 0 to *strlen* of the smaller string. The sign of a nonzero return value is determined by the sign of the difference between the values of the first pair of bytes (both interpreted as type unsigned char) that differ in the strings being compared. This function is *not* locale-sensitive.

Returned value

strcmp() returns a value indicating the relationship between the strings, as listed below.

Value	Meaning
< 0	String pointed to by *string1* less than string pointed to by *string2*
= 0	String pointed to by *string1* equivalent to string pointed to by *string2*
> 0	String pointed to by *string1* greater than string pointed to by *string2*

Example

CELEBS36

```
/* CELEBS36

   This example compares the two strings passed to main using
   &strcmp..

 */
#include <stdio.h>
#include <string.h>

int main(int argc, char ** argv)
{
  int  result;
  if ( argc != 3 )
  {
    printf( "Usage: %s string1 string2\n", argv[0] );
  }
  else
  {

    result = strcmp( argv[1], argv[2] );

    if ( result == 0 )
      printf( "\"%s\" is identical to \"%s\"\n", argv[1], argv[2] );
    else if ( result < 0 )
      printf( "\"%s\" is less than \"%s\"\n", argv[1], argv[2] );
    else
      printf( "\"%s\" is greater than \"%s\"\n", argv[1], argv[2] );
  }
}
```

Output

If the input is the strings `"is this first?"` and `"is this before that one?"` then the expected output is:

```
"is this first?" is greater than "is this before that one?"
```

Related information

- “string.h” on page 72
- “memcmp() — Compare bytes” on page 1062
- “strcspn() — Compare strings” on page 1735
- “strncmp() — Compare strings” on page 1751
- “strpbrk() — Find characters in string” on page 1754
- “strrchr() — Find last occurrence of character in string” on page 1759
- “strspn() — Search string” on page 1761

strcoll() — Compare strings

Standards

Standards / Extensions	C or C++	Dependencies
ISO C POSIX.1 XPG4 XPG4.2 C99 Single UNIX Specification, Version 3	both	

Format

```
#include <string.h>

int strcoll(const char *string1, const char *string2);
```

General description

Compares the string pointed to by *string1* against the string pointed to by *string2*, both interpreted according to the information in the LC_COLLATE category of the current locale.

Returned value

strcoll() returns a value indicating the relationship between the strings, as listed below.

Value	Meaning
< 0	string pointed to by *string1* less than string pointed to by *string2*
= 0	string pointed to by *string1* equivalent to string pointed to by *string2*
> 0	string pointed to by *string1* greater than string pointed to by *string2*

Notes:

1. The strcoll() function may need to allocate additional memory to perform the comparison algorithm specified in the LC_COLLATE. If the memory request cannot be satisfied (by malloc()), strcoll() fails.
2. If the locale supports double-byte characters (`MB_CUR_MAX` specified as 4), the strcoll() function validates the multibyte characters, whereas previously the strcoll() function did not validate the string. The strcoll() function will fail if the string contains invalid multibyte characters.
3. If `MB_CUR_MAX` is specified as 4, but the charmap file does not specify the DBCS characters, the DBCS characters will collate after the single-byte characters.

Example

CELEBS37

```
/* CELEBS37

   This example compares the two strings passed to main.

 */
#include <stdio.h>
#include <string.h>

int main(int argc, char ** argv)
{
  int  result;

  if ( argc != 3 ) {
    printf( "Usage: %s string1 string2\n", argv[0] );
  }
  else {

    result = strcoll( argv[1], argv[2] );

    if ( result == 0 )
      printf( "\"%s\" is identical to \"%s\"\n", argv[1], argv[2] );
    else if ( result < 0 )
      printf( "\"%s\" is less than \"%s\"\n", argv[1], argv[2] );
    else
      printf( "\"%s\" is greater than \"%s\"\n", argv[1], argv[2] );
  }
}
```

Output

If the input is the strings “firststring” and “secondstring”, then the expected output is:

```
"firststring" is less than "secondstring"
```

Related information

- “string.h” on page 72
- “setlocale() — Set locale” on page 1552
- “strcmp() — Compare strings” on page 1731
- “strncmp() — Compare strings” on page 1751

strcpy() — Copy string

Standards

Standards / Extensions	C or C++	Dependencies
ISO C XPG4 XPG4.2 C99 Single UNIX Specification, Version 3	both	

Format

```
#include <string.h>

char *strcpy(char * __restrict__string1, const char * __restrict__string2);
```

General description

The strcpy() built-in function copies *string2*, including the ending NULL character, to the location specified by *string1*. The *string2* argument to strcpy() must contain a NULL character (\0) marking the end of the string. You cannot use a literal string for a *string1* value, although *string2* may be a literal string. If the two objects overlap, the behavior is undefined.

Returned value

strcpy() returns the value of *string1*.

Example

CELEBS38

```
/* CELEBS38

   This example copies the contents of source to destination.

 */
#include <stdio.h>
#include <string.h>

#define SIZE    40

int main(void)
{
  char source[ SIZE ] = "This is the source string";
  char destination[ SIZE ] = "And this is the destination string";
  char * return_string;

  printf( "destination is originally = \"%s\"\n", destination );
  return_string = strcpy( destination, source );
  printf( "After strcpy, destination becomes \"%s\"\n", destination );
}
```

Output

```
destination is originally = "And this is the destination string"
After strcpy, destination becomes "This is the source string"
```

Related information

- "string.h" on page 72
- "memcpy() — Copy buffer" on page 1063
- "strcat() — Concatenate strings" on page 1728
- "strchr() — Search for character" on page 1729
- "strcmp() — Compare strings" on page 1731
- "strcspn() — Compare strings"
- "strncpy() — Copy string" on page 1753
- "strpbrk() — Find characters in string" on page 1754
- "strrchr() — Find last occurrence of character in string" on page 1759
- "strspn() — Search string" on page 1761

strcspn() — Compare strings

Standards

Standards / Extensions	C or C++	Dependencies
ISO C XPG4 XPG4.2 C99 Single UNIX Specification, Version 3	both	

Format

```
#include <string.h>

size_t strcspn(const char *string1, const char *string2);
```

General description

Computes the length of the initial portion of the string pointed to by *string1* that contains no characters from the string pointed to by *string2*.

Returned value

strcspn() returns the calculated length of the initial portion found.

Example

CELEBS39

```
/* CELEBS39

   This example uses &strcspn. to find the first occurrence of
   any of the characters a, x, l or e in string.

 */
#include <stdio.h>
#include <string.h>

#define SIZE    40

int main(void)
{
  char string[ SIZE ] = "This is the source string";
  char * substring = "axle";

  printf( "The first %i characters in the string \"%s\"\
  are not in the " "string \"%s\" \n",
            strcspn(string, substring), string, substring);
}
```

Output

```
The first 10 characters in the string "This is the source string"
are not in the string "axle"
```

Related information

- "string.h" on page 72
- "strcat() — Concatenate strings" on page 1728
- "strchr() — Search for character" on page 1729
- "strcmp() — Compare strings" on page 1731
- "strcpy() — Copy string" on page 1734
- "strncmp() — Compare strings" on page 1751
- "strpbrk() — Find characters in string" on page 1754
- "strrchr() — Find last occurrence of character in string" on page 1759
- "strspn() — Search string" on page 1761

strdup() — Duplicate a string

Standards

Standards / Extensions	C or C++	Dependencies
XPG4.2 Single UNIX Specification, Version 3	both	

Format

```
#define  _XOPEN_SOURCE_EXTENDED 1
#include <string.h>

char *strdup(const char *string);
```

General description

The strdup() function creates a duplicate of the string pointed to by *string*.

Returned value

If successful, strdup() returns a pointer to a new string which is a duplicate of *string*.

Otherwise, strdup() returns a NULL pointer.

Note: The caller of strdup() should free the storage obtained for the string.

Error Code
Description

ENOMEM
Insufficient storage space is available.

Related information

- “string.h” on page 72
- “free() — Free a block of storage” on page 618
- “malloc() — Reserve storage block” on page 1031

strerror() — Get pointer to runtime error message

Standards

Standards / Extensions	C or C++	Dependencies
ISO C POSIX.1 XPG4 XPG4.2 C99 Single UNIX Specification, Version 3	both	

Format

```
#include <string.h>

char *strerror(int errnum);
```

General description

Maps the error number in *errnum* to an error message string. The errnum must be a valid errno value.

This function does not produce a locale-dependent error message string.

Returned value

strerror() returns a pointer to the string, which may be overwritten by a subsequent call to strerror().

Note: Do not allow the content of this string to be modified by the program.

Example

```
/* This example opens a file and prints a runtime error message if an
   error occurs.
 */
#include <stdio.h>
#include <string.h>
#include <errno.h>

int main(void)
{
   FILE *stream;
⋮
   if ((stream = fopen("myfile.dat", "r")) == NULL)
      printf(" %s \n", strerror(errno));
}
```

Related information

- “string.h” on page 72
- “clearerr() — Reset error and end of file (EOF)” on page 284
- “ferror() — Test for read and write errors” on page 503
- “perror() — Print error message” on page 1175

strerror_r() — Get copy of runtime error message

Standards

Standards / Extensions	C or C++	Dependencies
Single UNIX Specification, Version 3	both	z/OS V1R7

Format

```
#define _UNIX03_SOURCE
#include <string.h>

int strerror_r(int errnum, char *strerrbuf, size_t buflen);
```

General description

strerror_r() maps the error number in *errnum* to an error message string and copies the message string into the buffer pointed to by *strerrbuf* with length *buflen*. If the length of the message string is greater than or equal to *buflen*, strerror_r() copies the first *buflen*-1 characters of the message string into *strerrbuf*, terminates *strerrbuf* with a null character (\0) and returns ERANGE. The error number must be a valid errno value.

This function does not produce a locale-dependent error message string.

Returned value

If successful, strerror_r() returns 0.

If unsuccessful, strerror_r() returns an error number to indicate the error.
- EINVAL – The value of errnum is not a valid error number.
- ERANGE – Insufficient storage was supplied via strerrbuf and buflen to contain the generated message string.

Related information
- errno.h
- string.h
- clearerr()
- ferror()
- perror()
- strerror()
- unsetenv()

strfmon() — Convert monetary value to string

Standards

Standards / Extensions	C or C++	Dependencies
XPG4 XPG4.2 Single UNIX Specification, Version 3	both	

Format

```
#define _XOPEN_SOURCE
#include <monetary.h>

ssize_t strfmon(char * __restrict__ s, size_t maxsize,
                const char * __restrict__ format, ...);
```

General description

strfmon() produces a formatted monetary output string from a double argument. It has been extended to determine floating-point argument format (hexadecimal floating-point or IEEE Binary Floating-Point) using the __isBFP() function.

Note: In IEEE Binary Floating-Point mode, denormal, infinity and NaN argument values are out of range.

Places characters into the array pointed to by **s* as controlled by the string pointed to by *format*. No more than *maxsize* characters are placed into the array.

The character string *format* contains two types of objects: plain characters, which are copied to the output array, and directives, each of which results in the fetching of zero or more arguments that are converted and formatted. The results are undefined if there are insufficient arguments for the *format*. If the *format* is exhausted while arguments remain, the excess arguments are simply ignored. If objects pointed to by *s* and *format* overlap, the behavior is undefined.

The directive (conversion specification) consists of the following sequence:
1. A % character
2. Optional flags: =f, ^, !, then +, C, or (
3. Optional field width (may be preceded by w
4. Optional left precision: #n
5. Optional right precision: .p
6. Required conversion character to indicate what conversion should be performed: i or n

Each directive is replaced by the appropriate characters, as described in the following list:

%i The double argument is formatted according to the locale's international currency format (for example, in USA: USD 1,234.56). An @euro codeset modifier can be used to request "EUR" instead of a national 4-character monetary string.

%n The double argument is formatted according to the locale's national currency format (for example, in USA: $1,234.56). An @euro codeset modifier can be used to get <euro-sign> instead of <currency>.

%% is replaced by %. No argument is converted.

The following optional conversion specifications may immediately follow the initial % of a directive:

=f A flag, used in conjunction with the maximum digits specification #*n* (see below), specifies that the character *f* should be used as the numeric fill character. The default numeric fill character is the space character. This option does not affect the other fill operations that always use space as the fill character.

^ A flag. Do not format the currency amount with thousands grouping characters. The default is to insert the grouping characters if defined for the current locale.

Note: The code point for the ^ character will be determined according to the current LC_SYNTAX category.

+ | C | (
A flag, specifies the style of representing positive and negative currency amounts. Only one of +, C, or (may be specified. If + is specified, the locale's equivalent of + and - are used (for example, in USA: the empty (NULL) string if positive and - if negative). If C is specified, the locale's equivalent of DB for negative and CR for positive are used. If (is specified, the locale's equivalent of enclosing negative amounts within parentheses is used. If this option is not included, a default specified by the current locale is used.

[-]w The field width. The decimal digit string *w* specifies a minimum field width in which the result of the conversion is right-justified (or left-justified if the optional flag "-" is specified).

#n The left precision. The decimal digit string *n* specifies the maximum number of digits expected to be formatted to the left of the radix character. This option can be used to keep the formatted output from multiple calls to the strfmon() aligned in the same columns. It can also be used to fill unused positions with a special character as in $***123.45. This option causes an amount to be formatted as if it has the number of digits specified by *n*. If more digit positions are required than the number specified, conversion specification is ignored. Digit positions in excess of those actually required are filled with the numeric fill character. (See the =f specification above.)

If the thousands grouping is enabled, the behavior is:

1. Format the number as if it is an *n* digit number.
2. Insert fill characters to the left of the leftmost digit (for example, $0001234.56 or $***1234.56)
3. Insert the separator character (for example, $0,001,234.56 or $*,**1,234.56)
4. If the fill character is not the digit zero, the separators are replaced by the fill character (for example, $****1,234.56).

To ensure alignment, any characters appearing before or after the number in the formatted output such as currency or sign symbols are padded as necessary with space characters to make their positive and negative formats an equal length.

Note: The code point for the # character (in #n) will be determined according to the current LC_SYNTAX category.

.p The right precision. The decimal digit string *p* specifies the number of digits after the radix character. If the value of the precision *p* is zero, no radix character appears. If this option is not included, a default specified by the current locale is used. The amount being formatted is rounded to the specified number of digits before formatting.

! A flag used to suppress the currency symbol from the output conversion.

Note: The code point for the ! character is determined according to the current LC_SYNTAX category.

The LC_MONETARY category of the program's locale affects the behavior of this function including the monetary radix character (which is different from the numeric radix character affected by the LC_NUMERIC category), the thousands (or alternative grouping) separator, the currency symbols and formats. The international currency symbol must be in accordance with those specified in ISO4217 Codes for the representation of currencies and funds.

Formatting choices are indicated in the LC_MONETARY category for the output of both national and international monetary quantities. The national format is determined by the settings of p_cs_precedes, n_cs_precedes, p_sign_posn, n_sign_posn, p_sep_by_space, and n_sep_by_space. An equivalent set of members for international formats are added to conform with the ISO/IEC standard. See "locale.h" on page 40 for more information on international formats.

The following tables show expected results for the various combinations of sep_by_space and sign_posn. All examples are based on a positive monetary quantity of 123.00, positive sign of '+', and currency symbol of '$'. Note that formatting rules are equivalent for negative and non-negative values as well as for national and international formats.

Table 53. Monetary formats when cs_precedes = 1

sep_by_space	**sign_posn**				
	0	1	2	3	4
0	($123.00)	+$123.00	$123.00+	+$123.00	$+123.00
1	($ 123.00)	+$ 123.00	$ 123.00+	+$ 123.00	$+ 123.00
2	($123.00)	+ $123.00	$123.00 +	+ $123.00	$ +123.00

Table 54. Monetary formats when cs_precedes = 0

sep_by_space	**sign_posn**				
	0	1	2	3	4
0	(123.00$)	+123.00$	123.00$+	123.00+$	123.00$+
1	(123.00 $)	+123.00 $	123.00 $+	123.00 +$	123.00 $+
2	(123.00$)	+ 123.00$	123.00$ +	123.00+ $	123.00$ +

cs_precedes

0 The currency symbol follows the value.

1 The currency symbol precedes the value.

sep_by_space

0 No space separates the currency symbol and value.

1 If the currency symbol and sign string are adjacent, a space separates them from the value; otherwise, a space separates the currency symbol from the value.

2 If the currency symbol and sign string are adjacent, a space separates them; otherwise, a space separates the sign string from the value.

sign_posn

0 Parentheses surround the quantity and currency_symbol or int_curr_symbol.

1 The sign string precedes the quantity and currency_symbol or int_curr_symbol.

2 The sign string succeeds the quantity and currency_symbol or int_curr_symbol.

3 The sign string immediately precedes the currency_symbol or int_curr_symbol.

4 The sign string immediately succeeds the currency_symbol or int_curr_symbol.

Returned value

If the total number of resulting bytes including the terminating NULL character is not more than *maxsize*, strfmon() returns the number of bytes placed into the array pointed to by *s*, not including the terminating NULL character.

If unsuccessful, the contents of the array are indeterminate, strfmon() returns -1, and sets errno to one of the following values:

Error Code
Description

E2BIG Conversion stopped due to lack of space in the buffer

Example

CELEBS41

```
/* CELEBS41 */
#include <localdef.h>
#include <monetary.h>
#include <stdio.h>
#include <stdlib.h>

int main(void)
{
   char   string[100];     /* hold the string returned from strfmon() */
   double money = 1234.56;

   if (setlocale(LC_ALL, "En_US") == NULL) {
      printf("Unable to setlocale().\n");
      exit(1);
   }

   strfmon(string, 100, "%i", money);
   printf("%s\n", string);
   strfmon(string, 100, "%n", money);
   printf("%s\n", string);
}
```

Example

The following example shows euro currency support:

```
/* EUROSAMP
   This example sets the locale to Fr_BE.IBM-1148
   and Fr_BE.IBM-1148@euro and prints a value with
   the locales national and international currency
   format.
*/

#include <localdef.h>
#include <locale.h>
#include <stdio.h>
#include <stdlib.h>

int main(void)
{
  char string[100];
  double money = 1234.56;

if (setlocale(LC_ALL,"Fr_BE.IBM-1148") == NULL) {
    printf("Unable to setlocale().\n");
    exit(1);
}

strfmon(string,100,"%i",money);
printf("%s\n",string);
strfmon(string,100,"%n",money);
printf("%s\n",string);

if (setlocale(LC_ALL,"Fr_BE.IBM-1148@euro") == NULL) {
    printf("Unable to setlocale().\n");
    exit(1);
}

strfmon(string,100,"%i",money);
printf("%s\n",string);
strfmon(string,100,"%n",money);
printf("%s\n",string);
}
```

Output

```
1.234,56 BEF
1.234,56 BF
1.234,56 EUR
1.234,56 <euro-sign>
```

Related information

- “monetary.h” on page 48
- “__isBFP() — Determine application floating-point format” on page 904

strftime() — Convert to formatted time

Standards

Standards / Extensions	C or C++	Dependencies
ISO C POSIX.1 XPG4 XPG4.2 C99 Single UNIX Specification, Version 3	both	

Format

```
#include <time.h>

size_t strftime(char * __restrict__ dest, size_t maxsize,
                const char * __restrict__ format, const struct tm * __restrict__ timeptr);
```

General description

Places characters into the array pointed to by *dest* according to the string pointed to by *format*. The format string is a multibyte character string which contains:

- Conversion specification characters
- Ordinary multibyte characters, which are copied into an array unchanged.

The characters that are converted are determined by the LC_CTYPE category of the current locale and by the values in the time structure pointed to by *timeptr*. The time structure pointed to by *timeptr* is usually obtained by calling the gmtime() or localtime() function.

Table 55. Conversion Specifiers Used by strftime()

Specifier	Meaning
%a	Replace with abbreviated weekday name of locale.
%A	Replace with full weekday name of locale.
%b	Replace with abbreviated month name of locale.
%B	Replace with full month name of locale.
%c	Replace with date and time of locale.
%C	Replace with locale's century number (year divided by 100 and truncated).
%d	Replace with day of the month (01-31).
%D	Insert date in mm/dd/yy form, regardless of locale.
%e	Insert day of the month as a decimal number (01–31). Under C POSIX only, it's a 2-character, right-justified, blank-filled field.
%E[cCxyY]	If the alternative date/time format is not available, the %E descriptors are mapped to their unextended counterparts. For example, %EC is mapped to %C.
%Ec	Replace with the locale's alternative date and time representation.
%EC	Replace with the name of the base year (period) in the locale's alternative representation.
%Ex	Replace with the locale's alternative date representation.
%EX	Replace with the locale's alternative time representation.
%Ey	Replace with the offset from %EC (year only) in the locale's alternative representation.
%EY	Replace with the full alternative year representation.
%F	Replace with the ISO 8601:2000 standard date format, equivalent to %Y-%m-%d. Values are taken from struct tm members, tm_year, tm_mon, and tm_mday.
%g	Replace with the last two digits of the week-based year as a decimal number (00-99).
%G	Replace with the week-based year as a four digit decimal.
%h	Replace with locale's abbreviated month name. This is the same as %b. %b.

Table 55. Conversion Specifiers Used by strftime() (continued)

Specifier	Meaning
%H	Replace with hour (24-hour clock) as a decimal number (00-23).
%I	Replace with hour (12-hour clock) as a decimal number (01-12).
%j	Replace with day of the year (001-366).
%m	Replace with month (01-12).
%M	Replace with minute (00-59).
%n	Replace with a newline.
%O[deHImMSUwWy]	If the alternative date/time format is not available, the %E descriptors are mapped to their unextended counterparts. For example, %Od is mapped to %d.
%Od	Replace with the day of month, using the locale's alternative numeric symbols, filled as needed with leading zeros if there is any alternative symbol for zero, otherwise with leading spaces.
%Oe	Replace with the day of the month, using the locale's alternative symbols, filled as needed with leading spaces.
%OH	Replace with the hour (24-hour clock) using the locale's alternative symbols.
%OI	Replace with the hour (12-hour clock) using the locale's alternative symbols.
%Om	Replace with the month using the locale's alternative numeric symbols.
%OM	Replace with the minutes using the locale's alternative numeric symbols.
%OS	Replace with the seconds using the locale's alternative numeric symbols.
%Ou	Replace with the weekday as a number in the locale's alternative representation (Monday=1).
%OU	Replace with the week number of the year (Sunday as the first day of the week, rules corresponding to %U) using the locale's alternative numeric symbols.
%OV	Replace with the week number of the year (Monday as the first day of the week, rules corresponding to %V) using the locale's alternative numeric symbols.
%Ow	Replace with the weekday (Sunday=0) using the locale's alternative numeric symbols.
%OW	Replace with the week number of the year (Monday as the first day of the week) using the locale's alternative numeric symbols.
%Oy	Replace with the year (offset from %C) in the locale's alternative representation and using the locale's alternative numeric symbols.
%p	Replace with the locale's equivalent of AM or PM.
%r	Replace with a string equivalent to %I:%M:%S %p; or use `t_fmt_ampm` from LC_TIME, if present.
%R	Replace with time in 24 hour notation (%H:%M).
%S	Replace with seconds as a decimal number (00-60).
%t	Replace with a tab.
%T	Replace with a string equivalent to %H:%M:%S.

Table 55. Conversion Specifiers Used by strftime() (continued)

Specifier	Meaning
%u	Replace with the weekday as a decimal number (1 to 7), with 1 representing Monday.
%U	Replace with week number of the year (00-53) where Sunday is the first day of the week. The first Sunday of January is the first day of week 1; days in the new year before this are in week 0.
%V	Replace with week number of the year (01-53) where Monday is the first day of the week. If the week containing 1 January has four or more days in the new year, then it is considered week 1. Otherwise, it is the last week of the previous year, and the next week is week 1. Both January 4th and the first Thursday of January are always in week 1.
%w	Replace with weekday (0-6) where Sunday is 0.
%W	Replace with week number of the year (00-53) where Monday is the first day of the week.
%x	Replace with date representation of locale.
%X	Replace with time representation of locale.
%y	Replace with year without the century (00-99).
%Y	Replace with year with century.
%z	Replace with the offset from UTC in ISO8601:2000 standard format (+hhmm or -hhmm). For example, "-0430" means 4 hours 30 minutes behind UTC (west of Greenwich). If tm_isdst is zero, the standard time offset is used. If tm_isdst is greater than zero, the daylight savings time offset is used. If tm_isdst is negative, or if no timezone can be determined, then no characters are returned.
%Z	Replace with name of time zone, or no characters if time zone is not available.
%%	Replace with %.

If data has the form of a directive, but is not one of the above, the characters following the % are copied to the output.

The behavior is undefined when objects being copied overlap. *maxsize* specifies the maximum number of characters that can be copied into the array.

If strftime() is called by a non-POSIX application, it obtains appropriate time zone name information from LC_TOD locale category. Time zone name defaults to STD for Standard time name, DST for Daylight Savings time name, or UTC for Coordinated Universal Time (UTC), name, as appropriate, if time zone name information is unspecified in the current LC_TOD locale category.

Note: The strftime() function requires time zone name information to convert the %Z conversion specifier. It is obtained as follows:

- The strftime() functions calls the tzset() function to obtain time zone information by parsing the TZ (POSIX) or _TZ (non_POSIX) environment variable or from the current LC_TOD locale category. If the `tm` structure input to strftime() was produced by calling localtime(), strftime() converts %Z to the Standard or Daylight Savings name characters specified by the TZ environment variable or LC_TOD category (if TZ cannot be found or parsed).

- When neither TZ nor _TZ is defined, the current locale is interrogated for time zone information. If neither TZ nor _TZ is defined and LC_TOD time zone information is not present in the current locale, a default value is applied to local time. POSIX programs simply default to Coordinated Universal Time (UTC), while non-POSIX programs establish an offset from UTC based on the setting of the system clock. For more information about customizing a time zone to work with local time, see “Customizing a time zone” in *z/OS XL C/C++ Programming Guide*.

The tm_isdst flag in the time structure input to strftime() determines whether %Z is replaced by the Standard or Daylight Savings name characters. If Standard or Daylight Savings name characters are not available in the current LC_TOD locale category or from parsing TZ, strftime() uses the characters STD for Standard or DST for Daylight Savings time name.

If the tm structure input to strftime() was produced by the gmtime() function, strftime() replaces %Z by UCTNAME characters specified in the current LC_TOD locale category or by UTC if UCTNAME is not specified.

Returned value

If successful, strftime() returns the number of characters (bytes) placed into the array, not including the terminating NULL character.

If unsuccessful, strftime() returns 0 and the content of the string is indeterminate.

Example

CELEBS42

```
/* CELEBS42

   This example places characters into the array dest and prints
   the resulting string.

 */
#include <stdio.h>
#include <time.h>

int main(void)
{
  char dest[70];
  int ch;
  time_t temp;
  struct tm *timeptr;

  temp = time(NULL);
  timeptr = localtime(&temp);
  ch = strftime(dest,sizeof(dest)-1,"Today is %A,"
             " %b %d. \n Time: %I:%M %p", timeptr);
  printf("%d characters placed in string to make: \n \n %s", ch, dest);
}
```

Output

```
44 characters placed in string to make:

 Today is Friday, Jun 16.
 Time: 03:07 PM
```

Related information

- "time.h" on page 79
- "asctime(), asctime64() — Convert time to character string" on page 182
- "asctime_r(), asctime64_r() — Convert date and time to a character string" on page 184
- "ctime(), ctime64() — Convert time to character string" on page 359
- "ctime_r(), ctime64_r() — Convert time value to date and time character string" on page 362
- "gmtime(), gmtime64() — Convert time to broken-down UTC time" on page 810
- "gmtime_r(), gmtime64_r() — Convert a time value to broken-down UTC time" on page 812
- "localdtconv() — Date and time formatting convention inquiry" on page 981
- "localtime(), localtime64() — Convert time and correct for local time" on page 984
- "localtime_r(), localtime64_r() — Convert time value to broken-down local time" on page 986
- "mktime(), mktime64() — Convert local time" on page 1078
- "time(),time64() — Determine current UTC time" on page 1878
- "tzset() — Set the time zone" on page 1936

strlen() — Determine string length

Standards

Standards / Extensions	C or C++	Dependencies
ISO C POSIX.1 XPG4 XPG4.2 C99 Single UNIX Specification, Version 3	both	

Format

```
#include <string.h>

size_t strlen(const char *string);
```

General description

The strlen() built-in function determines the length of string pointed to by *string*, excluding the terminating NULL character.

Returned value

strlen() returns the length of *string*.

Example

CELEBS43

```
/* CELEBS43

   This example determines the length of the string that is
   passed to main.

 */
#include <stdio.h>
```

```
#include <string.h>

int main(int argc, char **argv)
{

  if ( argc != 2 )
    printf( "Usage: %s string\n", argv[0] );
  else
    printf( "Input string has a length of %i\n", strlen( argv[1] ));
}
```

Output

If the input is the string: "How long is this string?", then the expected output is:

```
Input string has a length of 24
```

Related information

- “string.h” on page 72
- “mblen() — Calculate length of multibyte character” on page 1040
- “strncat() — Concatenate strings” on page 1750
- “strncmp() — Compare strings” on page 1751
- “strncpy() — Copy string” on page 1753
- “wcslen() — Calculate length of wide-character string” on page 2013

strncasecmp() — Case-insensitive string comparison

Standards

Standards / Extensions	C or C++	Dependencies
XPG4.2 Single UNIX Specification, Version 3	both	

Format

```
#define  _XOPEN_SOURCE_EXTENDED 1
#include <strings.h>

int strncasecmp(const char *string1, const char *string2, size_t n);
```

General description

The strncasecmp() function compares, while ignoring differences in case, the string pointed to by *string1* to the string pointed to by *string2*. At most *n* characters will be compared.

The string arguments to the function should contain a NULL character (\0) marking the end of the string.

The strncasecmp() function is locale-sensitive.

Returned value

strncasecmp() returns a value indicating the relationship between the strings, while ignoring case, as follows:

Value **Meaning**

< 0 String pointed to by *string1* is less than string pointed to by *string2*.

= 0 String pointed to by *string1* is equal to string pointed to by *string2*.

> 0 String pointed to by *string1* is greater than string pointed to by *string2*.

There are no errno values defined.

Related information

- “strings.h” on page 72
- “strcasecmp() — Case-insensitive string comparison” on page 1727
- “strcspn() — Compare strings” on page 1735
- “strncmp() — Compare strings” on page 1751

strncat() — Concatenate strings

Standards

Standards / Extensions	C or C++	Dependencies
ISO C POSIX.1 XPG4 XPG4.2 C99 Single UNIX Specification, Version 3	both	

Format

```
#include <string.h>

char *strncat(char * __restrict__string1, const char * __restrict__string2, size_t count);
```

General description

The strncat() built-in function appends the first *count* characters of *string2* to *string1* and ends the resulting string with a NULL character (\0). If *count* is greater than the length of *string2*, strncat() appends only the maximum length of *string2* to *string1*. The first character of the appended string overwrites the terminating NULL character of the string pointed to by *string1*.

If copying takes place between overlapping objects, the behavior is undefined.

Returned value

strncat() returns the value *string1*, the concatenated string.

Example

CELEBS44

```
/* CELEBS44

   This example demonstrates the difference between &strcat. and
   &strncat..
   &strcat. appends the entire second string to the first,
   whereas &strncat. appends only the specified number of
   characters in the second string to the first.

  */
#include <stdio.h>
#include <string.h>
```

```
#define SIZE 40


int main(void)
{
  char buffer1[SIZE] = "computer";
  char * ptr;

  /* Call strcat with buffer1 and " program" */

  ptr = strcat( buffer1, " program" );
  printf( "strcat : buffer1 = \"%s\"\n", buffer1 );

  /* Reset buffer1 to contain just the string "computer" again */

  memset( buffer1, '\0', sizeof( buffer1 ));
  ptr = strcpy( buffer1, "computer" );

  /* Call strncat with buffer1 and " program" */
  ptr = strncat( buffer1, " program", 3 );
  printf( "strncat: buffer1 = \"%s\"\n", buffer1 );
}
```

Output

```
strcat : buffer1 = "computer program"
strncat: buffer1 = "computer pr"
```

Related information

- "string.h" on page 72
- "strcat() — Concatenate strings" on page 1728
- "strncmp() — Compare strings"
- "strncpy() — Copy string" on page 1753
- "strpbrk() — Find characters in string" on page 1754
- "strrchr() — Find last occurrence of character in string" on page 1759
- "strspn() — Search string" on page 1761

strncmp() — Compare strings

Standards

Standards / Extensions	C or C++	Dependencies
ISO C POSIX.1 XPG4 XPG4.2 C99 Single UNIX Specification, Version 3	both	

Format

```
#include <string.h>

int strncmp(const char *string1, const char *string2, size_t count);
```

General description

The strncmp() built-in function compares at most the first *count* characters of the string pointed to by *string1* to the string pointed to by *string2*.

The string arguments to the function should contain a NULL character (\0) marking the end of the string.

The relation between the strings is determined by the sign of the difference between the values of the leftmost first pair of characters that differ. The values depend on character encoding. This function is *not* locale sensitive.

Returned value

strncmp() returns a value indicating the relationship between the substrings, as follows:

Value	Meaning
< 0	String pointed to by *substring1* less than string pointed to by *substring2*
= 0	String pointed to by *substring1* equivalent to string pointed to by *substring2*
> 0	String pointed to by *substring1* greater than string pointed to by *substring2*

Example

CELEBS45

```
/* CELEBS45

   This example demonstrates the difference between &strcmp.
   and &strncmp..

 */
#include <stdio.h>
#include <string.h>

#define SIZE 10

int index = 3;

int main(void)
{
  int  result;
  char buffer1[SIZE] = "abcdefg";
  char buffer2[SIZE] = "abcfg";
  void print_result( int, char *, char * );

  result = strcmp( buffer1, buffer2 );
  printf( "  strcmp: compares each character\n");
  print_result( result, buffer1, buffer2 );

  result = strncmp( buffer1, buffer2, index);
  printf( "\nstrncmp: compares only the first %i characters\n", index );
  print_result( result, buffer1, buffer2 );
}

void print_result( int res, char * p_buffer1, char * p_buffer2 )
{
  if ( res == 0 )
    printf( "first %i characters of \"%s\" is identical to \"%s\"\n",
             index,  p_buffer1, p_buffer2);
  else if ( res < 0 )
    printf( "\"%s\" is less than \"%s\"\n", p_buffer1, p_buffer2 );
  else
    printf( "\"%s\" is greater than \"%s\"\n", p_buffer1, p_buffer2 );
}
```

Output

```
strcmp: compares each character
"abcdefg" is less than "abcfg"

strncmp: compares only the first 3 characters
first 3 characters of "abcdefg" is identical to "abcfg"
```

Related information

- “string.h” on page 72
- “memcmp() — Compare bytes” on page 1062
- “strcmp() — Compare strings” on page 1731
- “strcspn() — Compare strings” on page 1735
- “strncat() — Concatenate strings” on page 1750
- “strncpy() — Copy string”
- “strpbrk() — Find characters in string” on page 1754
- “strrchr() — Find last occurrence of character in string” on page 1759
- “strspn() — Search string” on page 1761

strncpy() — Copy string

Standards

Standards / Extensions	C or C++	Dependencies
ISO C XPG4 XPG4.2 C99 Single UNIX Specification, Version 3	both	

Format

```
#include <string.h>

char *strncpy(char * __restrict__string1, const char * __restrict__string2, size_t count);
```

General description

The strncpy() built-in function copies at most *count* characters of *string2* to *string1*. If *count* is less than or equal to the length of *string2*, a NULL character (\0) is *not* appended to the copied string. If *count* is greater than the length of *string2*, the *string1* result is padded with NULL characters (\0) up to length *count*.

If copying takes place between objects that overlap, the behavior is undefined.

Returned value

strncpy() returns *string1*.

Example

CELEBS46

```
/* CELEBS46

   This example demonstrates the difference between &strcpy.
   and &strncpy..

 */
#include <stdio.h>
#include <string.h>
```

```
#define SIZE    40

int main(void)
{
  char source[ SIZE ] = "123456789";
  char source1[ SIZE ] = "123456789";
  char destination[ SIZE ] = "abcdefg";
  char destination1[ SIZE ] = "abcdefg";
  char * return_string;
  int    index = 5;

  /* This is how strcpy works */
  printf( "destination is originally = '%s'\n", destination );
  return_string = strcpy( destination, source );
  printf( "After strcpy, destination becomes '%s'\n\n", destination );


  /* This is how strncpy works */
  printf( "destination1 is originally = '%s'\n", destination1 );
  return_string = strncpy( destination1, source1, index );
  printf( "After strncpy, destination1 becomes '%s'\n", destination1 );
}
```

Output

```
destination is originally = 'abcdefg'
After strcpy, destination becomes '123456789'

destination1 is originally = 'abcdefg'
After strncpy, destination1 becomes '12345fg'
```

Related information

- “string.h” on page 72
- “memcpy() — Copy buffer” on page 1063
- “strcpy() — Copy string” on page 1734
- “strncat() — Concatenate strings” on page 1750
- “strncmp() — Compare strings” on page 1751
- “strpbrk() — Find characters in string”
- “strrchr() — Find last occurrence of character in string” on page 1759
- “strspn() — Search string” on page 1761

strpbrk() — Find characters in string

Standards

Standards / Extensions	C or C++	Dependencies
ISO C POSIX.1 XPG4 XPG4.2 C99 Single UNIX Specification, Version 3	both	

Format

```
#include <string.h>

char *strpbrk(const char *string1, const char *string2);
```

General description

Locates the first occurrence in the string pointed to by *string1* of any character from the string pointed to by *string2*.

Returned value

If successful, strpbrk() returns a pointer to the character.

If *string1* and *string2* have no characters in common, strpbrk() returns a NULL pointer.

Example

CELEBS47

```
/* CELEBS47

   This example returns a pointer to the first occurrence in the
   array string of either a or b.

 */
#include <stdio.h>
#include <string.h>

int main(void)
{
   char *result, *string = "A Blue Danube";
   char *chars = "ab";

   result = strpbrk(string, chars);
   printf("The first occurrence of any of the characters \"%s\" in "
          "\"%s\" is \"%s\"\n", chars, string, result);
}
```

Output

```
The first occurrence of any of the characters "ab" in "A Blue Danube"
is "anube"
```

Related information

- "string.h" on page 72
- "strchr() — Search for character" on page 1729
- "strcspn() — Compare strings" on page 1735
- "strncmp() — Compare strings" on page 1751
- "strrchr() — Find last occurrence of character in string" on page 1759
- "strspn() — Search string" on page 1761

strptime() — Date and time conversion

Standards

Standards / Extensions	C or C++	Dependencies
ISO C POSIX.1 XPG4 XPG4.2 C99 Single UNIX Specification, Version 3	both	

Format

```
#include <time.h>

char *strptime(const char * __restrict__ buf, const char * __restrict__ fmt,
               struct tm * __restrict__ tm);
```

General description

The strptime() function converts the character string pointed to by *buf* to values that are stored in the tm structure pointed to by *tm*, using the format specified by *fmt*. Only the fields in the tm structure for which there is a corresponding format item in *fmt* are updated by the strptime() function. Therefore, for a valid and conforming tm structure, provide strptime() with a format and data that completely specify the date and time being converted.

To avoid infringing on the user's name space, this nonstandard function has two names. One name is prefixed with two underscore characters, and one name is not. The name without the prefix underscore characters is exposed only when you use LANGLVL(EXTENDED).

To use this function, you must either invoke the function using its external entry point name (that is, the name that begins with two underscore characters), or compile with LANGLVL(EXTENDED). When you use LANGLVL(EXTENDED) any relevant information in the header is also exposed.

The format is composed of zero or more directives. Each directive is composed of one of the following: one or more white space characters (as specified by the isalnum() to isxdigit() function), an ordinary character other than a percent sign (%) or white space character, or a conversion specification. Each conversion specification is composed of a percent sign followed by a conversion character that specifies the replacement required. There must be a white space character or other nonalphanumeric character between any two conversion specifications.

Table 56. Conversion Specifiers Used by strptime()

Specifier	Meaning
%a	Day of week, using locale's abbreviated or full weekday name.
%A	Day of week, using locale's abbreviated or full weekday name.
%b	Month, using locale's abbreviated or full month name.
%B	Month, using locale's abbreviated or full month name.
%c	Date and time, using locale's date and time.
%C	Century number (year divided by 100 and truncated to an integer).
%d	Day of the month (1 - 31; leading zeros permitted but not required; day will not be checked for correctness for the month specified).
%D	Date as %m/%d/%y.
%e	Day of the month (1 - 31; leading zeros permitted but not required; day will not be checked for correctness for the month specified).
%h	Month, using locale's abbreviated or full month name.
%H	Hour (0 - 23; leading zeros permitted but not required).
%I	Hour (0 - 12; leading zeros permitted but not required).
%j	Day number of the year (001 - 366).
%m	Month number (1 - 12; leading zeros are permitted but not required).

Table 56. Conversion Specifiers Used by strptime() (continued)

Specifier	Meaning
%M	Minute (0 - 59; leading zeros are permitted but not required).
%n	Newline character.
%p	Locale's equivalent of AM or PM.
%r	12-hour clock time using the AM/PM notation if t_fmt_ampm is not an empty string in the LC_TIME portion of the current locale; in the POSIX locale, this is equivalent to %I : %M : %S %p.
%R	Time in 24 hour notation (11/19/01M).
%S	Seconds (0 - 60; leading zeros are permitted but not required).
%t	Tab character.
%T	Time as %H:%M:%S.
%U	Week number of the year (0 - 53; where Sunday is the first day of the week; leading zeros are permitted but not required).
%w	Weekday (0 - 6; where Sunday is 0; leading zeros are permitted but not required).
%W	Week number of the year (0 - 53; where Monday is the first day of the week; leading zeros are permitted but not required).
%x	Date, using locale's date format.
%X	Time, using locale's time format.
%y	Year within century. When a century is not otherwise specified, values in the range 69 - 99 refer to years in the twentieth century (1969 to 1999 inclusive); values in the range 00 - 68 refer to years in the twenty-first century (2000 to 2068 inclusive). Leading zeros are permitted but not required.
%Y	Year, including century. When the value of the _EDC_STRPTM_STD environment variable is set to 1, at most 4 digits will be consumed. When the value of _EDC_STRPTM_STD is set to other values or unset, more than 4 digits might be consumed, and if the generated value is greater than 9999, this function fails.
%Z	Time zone name.
%%	Replace with %.

Modified directives: Some directives can be modified by the E or O modifier characters to indicate that an alternative format or specification should be used rather than the one normally used by the unmodified directive. If the alternative format or specification does not exist in the current locale, the behavior will be as if the unmodified directive were used.

Table 57. Modified Directives Used by strptime()

Specifier	Meaning
%Ec	Replace with the locale's alternative date and time representation.
%EC	Replace with the name of the base year (period) in the locale's representation.
%Ex	Replace with the locale's alternative date representation.
%EX	Replace with the locale's alternative time representation.
%Ey	Replace with the offset from %EC (year only) in the locale's representation.

Table 57. Modified Directives Used by strptime() (continued)

Specifier	Meaning
%EY	Replace with the full alternative year representation.
%Od	Replace with the day of month, using the locale's alternative numeric symbols, filled as needed with leading zeros if there is any alternative symbol for zero, otherwise with leading spaces.
%Oe	Replace with the day of the month, using the locale's alternative numeric symbols, filled as needed with leading spaces.
%OH	Replace with the hour (24-hour clock) using the locale's alternative numeric symbols.
%OI	Replace with the hour (12-hour clock) using the locale's alternative numeric symbols.
%Om	Replace with the month using the locale's alternative numeric symbols.
%OM	Replace with the minutes using the locale's alternative numeric symbols.
%OS	Replace with the seconds using the locale's alternative numeric symbols.
%OU	Replace with the week number of the year (Sunday as the first rules corresponding to %U) using the locale's alternative numeric symbols.
%Ow	Replace with the weekday (Sunday=0) using the locale's alternative numeric symbols.
%OW	Replace with the week number of the year (Monday as the first day of the week) using the locale's alternative numeric symbols.
%Oy	Replace with the year (offset from %C) in the locale's alternative representation and using the locale's alternative numeric symbols.

A directive composed of white space characters is executed by scanning input up to the first character that is not white space (which remains unscanned) or until no more characters can be scanned.

A directive that is an ordinary character is executed by scanning the next character from the buffer. If the character scanned from the buffer differs from the one comprising the directive, the directive fails, and the differing and subsequent characters remain unscanned.

A series of directives composed or %n, %t, white space characters or any combination is executed by scanning up to the first character that is not white space (which remains unscanned), or until no more characters can be scanned.

Any other conversion specification is executed by scanning characters until a character matching the next directive is scanned, or until no more characters can be scanned. These characters, excepting the one matching the next directive, are then compared to the locale values associated with the conversion specifier. If a match is found, values for the appropriate `tm` structure members are set to values corresponding to the locale information. Case is ignored when matching items in *buf*, such as month or weekday names. If no match is found, strptime() fails and no more characters are scanned.

Returned value

If successful, strptime() returns a pointer to the character following the last character parsed.

If unsuccessful, strptime() returns a NULL pointer.

Example

CELEBS48

```
/* CELEBS48 */
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <localdef.h>

int main(void)
{
   struct tm xmas;

   if (strptime("12/25/93 13:30:00", "%D %T", &xmas) == NULL) {
      printf("strptime() failed.\n");
      exit(1);
   }

   printf("tm_sec  =%3d\n", xmas.tm_sec  );
   printf("tm_min  =%3d\n", xmas.tm_min  );
   printf("tm_hour =%3d\n", xmas.tm_hour );
   printf("tm_mday =%3d\n", xmas.tm_mday );
   printf("tm_mon  =%3d\n", xmas.tm_mon  );
   printf("tm_year =%3d\n", xmas.tm_year );
   printf("tm_wday =%3d\n", xmas.tm_wday );
   printf("tm_yday =%3d\n", xmas.tm_yday );
}
```

Output

```
tm_sec  =  0
tm_min  = 30
tm_hour = 13
tm_mday = 25
tm_mon  = 11
tm_year = 93
tm_wday =  0
tm_yday =358
```

Related information

- “time.h” on page 79

strrchr() — Find last occurrence of character in string

Standards

Standards / Extensions	C or C++	Dependencies
ISO C POSIX.1 XPG4 XPG4.2 C99 Single UNIX Specification, Version 3	both	

Format

```
#include <string.h>

char *strrchr(const char *string, int c);
```

General description

The strrchr() function finds the last occurrence of *c* (converted to a char) in *string*. The ending NULL character is considered part of the *string*.

Returned value

If successful, strrchr() returns a pointer to the last occurrence of *c* in *string*.

If the given character is not found, strrchr() returns a NULL pointer.

Example

CELEBS49

```
/* CELEBS49

   This example compares the use of &strchr. and &strrchr..
   It searches the string for the first and last occurrence of
   p in the string.

 */
#include <stdio.h>
#include <string.h>

#define SIZE 40

int main(void)
{
  char buf[SIZE] = "computer program";
  char * ptr;
  int    ch = 'p';

  /* This illustrates strchr */
  ptr = strchr( buf, ch );
  printf( "The first occurrence of %c in '%s' is '%s'\n", ch, buf, ptr );

  /* This illustrates strrchr */
  ptr = strrchr( buf, ch );
  printf( "The last occurrence of %c in '%s' is '%s'\n", ch, buf, ptr );
}
```

Output

```
The first occurrence of p in 'computer program' is 'puter program'
The last occurrence of p in 'computer program' is 'program'
```

Related information

- "string.h" on page 72
- "memchr() — Search buffer" on page 1061
- "strchr() — Search for character" on page 1729
- "strcspn() — Compare strings" on page 1735
- "strncmp() — Compare strings" on page 1751
- "strpbrk() — Find characters in string" on page 1754
- "strspn() — Search string" on page 1761

strspn() — Search string

Standards

Standards / Extensions	C or C++	Dependencies
Language Environment ISO C XPG4 XPG4.2 C99 Single UNIX Specification, Version 3	both	

Format

```
#include <string.h>

size_t strspn(const char *string1, const char *string2);
```

General description

Calculates the length of the maximum initial portion of the string pointed to by *string1* that consists entirely of the characters contained in the string pointed to by *string2*.

Returned value

strspn() returns the length of the substring found.

Example

CELEBS50

```
/* CELEBS50

   This example finds the first occurrence in the array string
   of a character that is neither an a, b, nor c. Because the
   string in this example is cabbage, &strspn. returns 5, the
   length of the segment of cabbage before a character that is
   not an a, b or c.

 */
#include <stdio.h>
#include <string.h>

int main(void)
{
  char * string = "cabbage";
  char * source = "abc";
  int index;

  index = strspn( string, "abc" );
  printf( "The first %d characters of \"%s\" are found in \"%s\"\n",
              index, string, source );
}
```

Output

```
The first 5 characters of "cabbage" are found in "abc"
```

Related information

- “string.h” on page 72
- “strcat() — Concatenate strings” on page 1728

- “strchr() — Search for character” on page 1729
- “strcmp() — Compare strings” on page 1731
- “strcpy() — Copy string” on page 1734
- “strcspn() — Compare strings” on page 1735
- “strpbrk() — Find characters in string” on page 1754
- “strrchr() — Find last occurrence of character in string” on page 1759

strstr() — Locate substring

Standards

Standards / Extensions	C or C++	Dependencies
ISO C XPG4 XPG4.2 C99 Single UNIX Specification, Version 3	both	

Format

```
#include <string.h>

char *strstr(const char *string1, const char *string2);
```

General description

Finds the first occurrence of the string pointed to by *string2* (excluding the NULL character) in the string pointed to by *string1*.

Returned value

If successful, strstr() returns a pointer to the beginning of the first occurrence of *string2* in *string1*.

If *string2* does not appear in *string1*, strstr() returns NULL.

If *string2* points to a string with zero length, strstr() returns *string1*.

Example

CELEBS51

```
/* CELEBS51

   This example locates the string haystack in the string "needle in a
   haystack".

 */
#include <stdio.h>
#include <string.h>

int main(void)
{
   char *string1 = "needle in a haystack";
   char *string2 = "haystack";
   char *result;
```

```
  result = strstr(string1,string2);
     /* Result = a pointer to "haystack" */
  printf("%s\n", result);
}
```

Output

```
haystack
```

Related information

- “string.h” on page 72
- “strchr() — Search for character” on page 1729
- “strcmp() — Compare strings” on page 1731
- “strcspn() — Compare strings” on page 1735
- “strncmp() — Compare strings” on page 1751
- “strpbrk() — Find characters in string” on page 1754
- “strrchr() — Find last occurrence of character in string” on page 1759
- “strspn() — Search string” on page 1761

strtocoll() — Return collating element for string

Standards

Standards / Extensions	C or C++	Dependencies
Language Environment	both	

Format

```
#include <collate.h>

collel_t strtocoll(char *s);
```

General description

Determines whether the string pointed to by *s* represents the valid element as defined in the LC_COLLATE category of the current locale.

If a string pointed to by *s* contains only one character, the collating element representing this character always exists. Otherwise, a valid collating element exists if the LC_COLLATE category contains the definition of a sequence of characters that collate as one for the purpose of culture-sensitive string comparison. This many-characters-to-one-collating element relation is also called *many-to-one* mapping.

Returned value

The type `collel_t` represents the collating elements.

If many-to-one mapping is not defined in the LC_COLLATE of the current locale, strtocoll() returns `(collel_t)-1`.

Also, if the string is not a valid collating element or is of zero length, strtocoll() returns `(collel_t)-1`.

Example

CELEBS52

```
/* CELEBS52

   This example uses the strtocoll() function to get the
   collel_t value for the start and end collating-elements for
   the collrange() function.

 */
#include <stdio.h>
#include <stdlib.h>
#include <locale.h>
#include <collate.h>
#include <wchar.h>
#include <wctype.h>

main(int argc, char *argv[]) {
   collel_t s, e, *rp;
   int i;

   setlocale(LC_ALL, "");
   if ((s = strtocoll(argv[1])) == (collel_t)-1) {
      printf("%s collating element not defined\n", argv[1]);
      exit(1);
   }
   if ((e = strtocoll(argv[2])) == (collel_t)-1) {
      printf("%s collating element not defined\n", argv[2]);
      exit(1);
   }
   if ((i = collrange(s, e, &rp)) == -1) {
      printf("Invalid range for %s to %s\n", argv[1], argv[2]);
      exit(1);
   }
   for (; i-- > 0; rp++) {
      if (ismccollel(*rp))
         printf("'%s' ", colltostr(*rp));
      else if (iswprint(*rp))
         printf("'%lc' ", *rp);
      else
         printf("'%x' ", *rp);
   }
}
```

Related information

- "collate.h" on page 19
- "cclass() — Return characters in a character class" on page 243
- "collequiv() — Return a list of equivalent collating elements" on page 295
- "collorder() — Return list of collating elements" on page 297
- "collrange() — Calculate the range list of collating elements" on page 298
- "colltostr() — Return a string for a collating element" on page 299
- "getmccoll() — Get next collating element from string" on page 734
- "getwmccoll() — Get next collating element from wide string" on page 803
- "ismccollel() — Identify a multicharacter collating element" on page 914
- "maxcoll() — Return maximum collating element" on page 1039

strtod() — Convert character string to double

Standards

Standards / Extensions	C or C++	Dependencies
ISO C XPG4 XPG4.2 C99 Single UNIX Specification, Version 3	both	

Format

```
#include <stdlib.h>

double strtod(const char * __restrict__nptr, char ** __restrict__endptr);
```

General description

Converts a part of a character string, pointed to by *nptr*, to a double. The parameter *nptr* points to a sequence of characters that can be interpreted as a numerical value of the type *double*.

See the "*fscanf* Family of Formatted Input Functions" on fscanf(), scanf(), sscanf() — Read and format data for a description of special infinity and NaN sequences recognized by z/OS formatted input functions, including atof() and strtod() in IEEE Binary Floating-Point mode.

The strtod() function breaks the string into three parts:
1. A sequence of white space characters (as specified for the current locale, see isspace())
2. A subject sequence interpreted as a floating-point constant or representing infinity or a NAN.
3. A sequence of unrecognized characters (including a NULL character).

The subject string is the longest string that matches the expected form.

The expected form of the subject sequence is an optional plus or minus sign, then one of the following:
- A non-empty sequence of decimal digits optionally containing a radix character, then an optional exponent part. Where radix character is the character that separates the integer part of a number from the fractional part.
- A 0x or 0X, then a non-empty sequence of hexadecimal digits optionally containing a radix character, then an optional binary exponent part. Where radix character is the character that separates the integer part of a number from the fractional part.
- One of INF or INFINITY, ignoring case.
- One of NANQ or NANQ(n-char-sequence), ignoring case.
- One of NANS or NANS(n-char-sequence), ignoring case.
- One of NAN or NAN(n-char-sequence), ignoring case.

The pointer to the last string successfully converted is stored in the object pointed to by *endptr*, provided that *endptr* is not a NULL pointer. If the subject string is empty or it does not have the expected form, then no conversion is performed. The value of *nptr* is stored in the object pointed to by *endptr*.

Returned value

If successful, strtod() returns the value of the floating-point number.

The double value is hexadecimal floating-point or IEEE Binary Floating-Point format depending on the floating-point mode of the thread invoking the strtod() function. This function uses __isBFP() to determine the floating-point mode of the invoking thread.

In an overflow, strtod() returns ±HUGE_VAL. In an underflow, it returns 0. If no conversion is performed, strtod() returns 0. In both cases, errno is set to ERANGE, depending on the base of the value.

Example

CELEBS53

```
/* CELEBS3

   This example converts a string to a double value.
   It prints out the converted value and the substring that
   stopped the conversion.

 */
#include <stdlib.h>
#include <stdio.h>

int main(void)
{
   char *string, *stopstring;
   double x;

   string = "3.1415926This stopped it";
   x = strtod(string, &stopstring);
   printf("string = %s\n", string);
   printf("   strtod = %f\n", x);
   printf("   Stopped scan at %s\n\n", stopstring);

   string = "100ergs";
   x = strtod(string, &stopstring);
   printf("string = \"%s\"\n", string);
   printf("   strtod = %f\n", x);
   printf("   Stopped scan at \"%s\"\n\n", stopstring);
}
```

Output

```
string = 3.1415926This stopped it
   strtod = 3.141593
   Stopped scan at This stopped it

string = 100ergs
   strtod = 100.000000
   Stopped scan at ergs
```

Related information

- “stdlib.h” on page 70
- “atof() — Convert character string to double” on page 203
- “atoi() — Convert character string to integer” on page 204
- “atol() — Convert character string to long” on page 204
- “fscanf(), scanf(), sscanf() — Read and format data” on page 625

- “__isBFP() — Determine application floating-point format” on page 904
- “strtof() — Convert character string to float” on page 1769
- “strtol() — Convert character string to long” on page 1775
- “strtold() — Convert character string to long double” on page 1777
- “strtoul() — Convert string to unsigned integer” on page 1780

strtod32(), strtod64(), strtod128() — Convert character string to decimal floating point

Standards

Standards / Extensions	C or C++	Dependencies
C/C++ DFP	both	z/OS V1.8

Format

```
#define __STDC_WANT_DEC_FP__
#include <stdlib.h>

_Decimal32 strtod32(const char * __restrict__ nptr, char ** __restrict__ endptr);
_Decimal64 strtod64(const char * __restrict__ nptr, char ** __restrict__ endptr);
_Decimal128 strtod128(const char * __restrict__ nptr, char ** __restrict__ endptr);
```

General description

The strtod32() strtod64(), and strtod128() functions convert the initial portion of the string pointed to by *nptr* to _Decimal32, _Decimal64, and _Decimal128 representation, respectively.

First, they decompose the input string into three parts:

1. An initial, possibly empty, sequence of white-space characters (as specified by the isspace() function) .
2. A subject sequence resembling a floating-point constant or representing an infinity or NaN.
3. A final string of one or more unrecognized characters, including the terminating null character of the input string.

Then, they attempt to convert the subject sequence to a floating-point number, and return the result.

The expected form of the subject sequence is an optional plus or minus sign, then one of the following:

- a nonempty sequence of decimal digits optionally containing a decimal-point character, then an optional exponent part
- INF or INFINITY, ignoring case
- NAN, NAN(n-char-sequence), NANQ, NANQ(n-char-sequence), NANS, or NANS(n-char-sequence), ignoring case in the NAN, NANQ, or NANS part, where n-char-sequence is one or more decimal numeric digits.

Note: If the input string is not one of these forms (for example “INFINITE”), the output results are undefined.

The subject sequence is defined as the longest initial subsequence of the input string, starting with the first non-white-space character, that is of the expected form. The subject sequence contains no characters if the input string is not of the expected form.

A character sequence NAN, NANQ, NAN(n-char-sequence), or NANQ(n-char-sequence) is interpreted as a quiet NAN. A character sequence of NANS, or NANS(n-char-sequence), is interpreted as a signalling NaN. A character sequence INF or INFINITY is interpreted as an infinity. A character sequence NAN, NAN(),or NAN(n-char-sequence) is interpreted as a quiet NAN. A character sequence of NANS, NANS(),or NANS(n-char-sequence), is interpreted as a signalling NaN.

A pointer to the final string is stored in the object pointed to by *endptr*, provided that *endptr* is not a null pointer.

The converted value keeps the same precision as the input if possible, and the value may be denormalized. Otherwise, rounding may occur. Rounding happens after any negation.

In other than the “C” locale, additional locale-specific subject sequence forms are accepted.

If the subject sequence is empty or does not have the expected form, no conversion is performed; the value of *nptr* is stored in the object pointed to by *endptr*, provided that *endptr* is not a null pointer.

Argument	Description
nptr	Input pointer to start of the string to be converted
endptr	NULL, or a pointer to a output pointer field that is filled in with the address of the first character in the input string that is not used in the conversion.

Note: To use IEEE decimal floating-point, the hardware must have the Decimal Floating-Point Facility installed.

Returned value

These functions return the converted value, if any. If no conversion could be performed, the value +0.E0DF, +0.E0DD,or +0.E0DL is returned. If the correct value is outside the range of representable values, plus or minus HUGE_VAL_D32, HUGE_VAL_D64, or HUGE_VAL_D128 is returned (according to the return type and sign of the value), and errno is set to ERANGE. If the result underflows, these functions return a value whose magnitude is no greater than the smallest normalized positive number in the return type. No signal is raised at the point of returning a signaling NaN.

errno	Description
ERANGE	The input string represents a value too large to fit in the output Decimal Floating Point type.

Example

See “strtod() — Convert character string to double” on page 1765 for an example.

Related information

- “math.h” on page 44
- “fscanf(), scanf(), sscanf() — Read and format data” on page 625
- “strtod() — Convert character string to double” on page 1765
- “wcstod32(), wcstod64(), wcstod128() — Convert wide-character string to decimal floating point” on page 2027

strtof() — Convert character string to float

Standards

Standards / Extensions	C or C++	Dependencies
C99 Single UNIX Specification, Version 3 C++ TR1 C99	both	z/OS V1R7

Format

```
#define _ISOC99_SOURCE
#include <stdlib.h>

float strtof(const char *__restrict__ nptr, char **__restrict__ endptr);
```

General description

strtof() converts a part of a character string, pointed to by *nptr*, to float. The parameter *nptr* points to a sequence of characters that can be interpreted as a numerical value of the type float.

The strtof() function breaks the string into three parts:

1. An initial, possibly empty, sequence of white-space characters (as specified by isspace()).
2. A subject sequence interpreted as a floating-point constant or representing infinity or a NAN.
3. A final string of one or more unrecognized characters, including the terminating null byte of the input string.

The function then attempts to convert the subject string into the floating-point number, and returns the result.

The expected form of the subject sequence is an optional plus or minus sign, then one of the following:

- A non-empty sequence of decimal digits optionally containing a radix character, then an optional exponent part. A radix character is the character that separates the integer part of a number from the fractional part.
- A 0x or 0X, a non-empty sequence of hexadecimal digits optionally containing a radix character, then a base 2 decimal exponent part with a p or P as prefix, a plus or minus sign, then a sequence of at least one decimal digit. (Example [-]0xh.hhhhp+/-d). A radix character is the character that separates the integer part of a number from the fractional part.
- One of INF or INFINITY, ignoring case.

- One of NANQ or NANQ(n-char-sequence), ignoring case.
- One of NANS or NANS(n-char-sequence), ignoring case.
- One of NAN or NAN(n-char-sequence), ignoring case.

See the "scanf Family of Formatted Input Functions" for a description of special infinity and NaN sequences recognized by z/OS formatted input functions in IEEE Binary Floating-Point mode.

The pointer to the last string successfully converted is stored in the object pointed to by *endptr*, provided that *endptr* is not a NULL pointer. If the subject string is empty or it does not have the expected form, then no conversion is performed. The value of *nptr* is stored in the object pointed to by *endptr*.

Returned value

If successful, strtof() returns the value of the floating-point number.

The float value is hexadecimal floating-point or IEEE Binary Floating-Point format depending on the floating-point mode of the thread invoking the strtof() function. This function uses __isBFP() to determine the floating-point mode of the invoking thread.

In an overflow, strtof() returns +/-HUGE_VALF. In an underflow, it returns 0. If no conversion is performed, strtof() returns 0. In both cases, errno is set to ERANGE, depending on the base of the value.

Related information

- “stdlib.h” on page 70
- “atof() — Convert character string to double” on page 203
- “atoi() — Convert character string to integer” on page 204
- “atol() — Convert character string to long” on page 204
- “fscanf(), scanf(), sscanf() — Read and format data” on page 625
- “__isBFP() — Determine application floating-point format” on page 904
- “strtod() — Convert character string to double” on page 1765
- “strtold() — Convert character string to long double” on page 1777
- “strtol() — Convert character string to long” on page 1775
- “strtoul() — Convert string to unsigned integer” on page 1780

strtoimax() — Convert character string to intmax_t integer type

Standards

Standards / Extensions	C or C++	Dependencies
C99 Single UNIX Specification, Version 3 C++ TR1 C99	both	z/OS V1R7

Format

```
#define _ISOC99_SOURCE
#include <inttypes.h>

intmax_t strtoimax(const char * __restrict__nptr, char ** __restrict__ endptr, int base);
```

Compile requirement: Function strtoimax() requires LONGLONG to be available.

General description

The strtoimax() function converts the string nptr to an intmax_t integer type. Valid input values for base are 0 and in the range 2-36. The strtoimax() function is equivalent to strtol() and strtoll() with the only difference being that the return value is of type intmax_t. See strtoll() for more information.

Returned value

If successful, strtoimax() returns the converted intmax_t value represented in the string.

If unsuccessful, strtoimax() returns 0 if no conversion could be performed. If the correct value is outside the range of representable values, strtoimax() returns INTMAX_MAX or INTMAX_MIN, according to the sign of the value. If the value of base is not supported, strtoimax() returns 0.

If unsuccessful strtoimax() sets errno to one of the following values:

Error Code
Description

EINVAL
The value of *base* is not supported.

ERANGE
The conversion caused an overflow.

Example

```
#define _ISOC99_SOURCE
#include <inttypes.h>
#include <stdlib.h>
#include <stdio.h>

int main(void)
     {
        intmax_t j;
        int base = 10;
        char *nptr, *endptr;
        nptr = "10345134932abc";
        printf("nptr = %s\n", nptr);
        j = strtoimax(nptr, &endptr, base);
        printf("strtoimax = %jd (base %d)\n", j, base);
        printf("Stopped scan at %s\n\n", endptr);
     }



Output

nptr = 10345134932abc
strtoimax = 10345134932(base 10)
Stopped scan at abc
```

Related information

- inttypes.h
- “strtol() — Convert character string to long” on page 1775
- “strtoul() — Convert string to unsigned integer” on page 1780
- “strtoll() — Convert string to signed long long” on page 1778
- “strtoull() — Convert string to unsigned long long” on page 1782

- "strtoumax() — Convert character string to uintmax_t integer type" on page 1784
- wcstoimax()
- wcstoumax()

strtok() — Tokenize string

Standards

Standards / Extensions	C or C++	Dependencies
ISO C XPG4 XPG4.2 C99 Single UNIX Specification, Version 3	both	

Format

```
#include <string.h>

char *strtok(char * __restrict__string1, const char * __restrict__string2);
```

General description

Breaks a character string, pointed to by *string1*, into a sequence of tokens. The tokens are separated from one another by the characters in the string pointed to by *string2*.

The token starts with the first character not in the string pointed to by *string2*. If such a character is not found, there are no tokens in the string. strtok() returns a NULL pointer. The token ends with the first character contained in the string pointed to by *string2*. If such a character is not found, the token ends at the terminating NULL character. Subsequent calls to strtok() will return the NULL pointer. If such a character *is* found, then it is overwritten by a NULL character, which terminates the token.

If the next call to strtok() specifies a NULL pointer for *string1*, the tokenization resumes at the first character following the found and overwritten character from the previous call. For example:

```
/* Here are two calls  */
strtok(string," ")
strtok(NULL," ")


/* Here is the string they are processing */
                   abc defg hij
    first call finds  ↑
                       ↑ second call starts
```

Returned value

The first time strtok() is called, it returns a pointer to the first token in *string1*. In later calls with the same token string, strtok() returns a pointer to the next token in the string. A NULL pointer is returned when there are no more tokens. All tokens are NULL-terminated.

Example

CELEBS54

```
/* CELEBS54
 *
 * strtok() example:
 *
 * This example parses tokens separated by commas, blanks and semicolons,
 * from a string until no tokens are left.  As the string is parsed,
 * pointers to the the following tokens are returned by strtok(),
 * and these tokens are written to stdout:
 *
 *   a
 *   string
 *   of
 *   tokens
 *
 * The final call to strtok() returns NULL indicating that
 * there are no more tokens.
 *
 * Note that as the string is tokenized, it will be overwritten.
 *
 */

#include <stdio.h>
#include <string.h>

int main(void)
{
   char *token, string[] = "a string, of,; ;;;,tokens\0,after null terminator";

   token = strtok(string, ", ;");
   do
   {
      printf("token: \"%s\"\n", token);
   }
   while (token = strtok(NULL, ", ;"));

}
```

Output

```
token: "a string"
token: " of"
token: " "
token: "tokens"
```

Related information

- "string.h" on page 72
- "strcat() — Concatenate strings" on page 1728
- "strchr() — Search for character" on page 1729
- "strcmp() — Compare strings" on page 1731
- "strcpy() — Copy string" on page 1734
- "strcspn() — Compare strings" on page 1735
- "strspn() — Search string" on page 1761
- "strtok_r() — Split string into tokens" on page 1774

strtok_r() — Split string into tokens

Standards

Standards / Extensions	C or C++	Dependencies
Single UNIX Specification, Version 2 Single UNIX Specification, Version 3	both	OS/390 V2R8

Format

```
#define _XOPEN_SOURCE 500
#include <string.h>

char *strtok_r(char *s, const char *sep, char **lasts);
```

General description

The function strtok_r() considers the NULL-terminated string *s* as a sequence of zero or more text tokens separated by spans of one or more characters from the separator string *sep*. The argument *lasts* points to a user-provided pointer which points to stored information necessary for strtok_r() to continue scanning the same string.

In the first call to strtok_r(), *s* points to a NULL-terminated string, *sep* to a NULL-terminated string of separator characters and the value pointed to by *lasts* is ignored. The function strtok_r() returns a pointer to the first character of the first token, writes a NULL character into *s* immediately following the returned token, and updates the pointer to which *lasts* points.

In subsequent calls, *s* is a NULL pointer and *lasts* will be unchanged from the previous call so that subsequent calls will move through the string *s*, returning successive tokens until no tokens remain. The separator string *sep* may be different from call to call. When no token remains in *s*, a NULL pointer is returned.

Returned value

If successful, strtok_r() returns a pointer to the token found.

When no token is found, strtok_r() returns a NULL pointer.

Related information

- "string.h" on page 72
- "strcat() — Concatenate strings" on page 1728
- "strchr() — Search for character" on page 1729
- "strcmp() — Compare strings" on page 1731
- "strcpy() — Copy string" on page 1734
- "strcspn() — Compare strings" on page 1735
- "strspn() — Search string" on page 1761
- "strtok() — Tokenize string" on page 1772

strtol() — Convert character string to long

Standards

Standards / Extensions	C or C++	Dependencies
ISO C XPG4 XPG4.2 C99 Single UNIX Specification, Version 3	both	

Format

```
#include <stdlib.h>

long int strtol(const char * __restrict__nptr, char ** __restrict__endptr, int base);
```

General description

Converts *nptr*, a character string, to a long int value.

The function decomposes the entire string into three parts:

1. A sequence of characters, which in the current locale are defined as white space characters. This part may be empty.
2. A sequence of characters interpreted as integer in some base notation. This is the *subject sequence*.
3. A sequence of unrecognized characters.

The base notation is determined by *base*, if base is greater than zero. If base is zero, the base notation is determined by the format of the sequence of characters that follow an optional plus—or optional minus—sign.

10 Sequence starts with nonzero decimal digit.

8 Sequence starts with 0, followed by a sequence of digits with values from 0 to 7.

16 Sequence starts with either `0x` or `0X`, followed by digits, and letters `A` through `F` or `a` through `f`.

If the base is greater than zero, the subject sequence contains decimal digits and letters, possibly preceded by either a plus or a minus sign. The letters `a` (or `A`) through `z` (or `Z`) represent values from 10 through 36, but only those letters whose value is less than the value of the base are allowed.

When you use the strtol() function, *nptr* should point to a string with the following form:

```
►►──┬─────────────┬──┬───┬──┬────┬──┬────────┬─────────────────────────────►◄
    └─white space─┘  ├─+─┤  ├─0──┤  └─digits─┘
                     └─-─┘  ├─0x─┤
                            └─0X─┘
```

The pointer to the converted characters, even if conversion was unsuccessful, is stored in the object pointed to by *endptr*, as long as *endptr* is not a NULL pointer.

Returned value

If successful, strtol() returns the converted long int value.

If unsuccessful, strtol() returns 0 if no conversion could be performed. If the correct value is outside the range of representable values, strtol() returns LONG_MAX or LONG_MIN, according to the sign of the value. If the value of base is not supported, strtol() returns 0.

If unsuccessful strtol() sets errno to one of the following values:

Error Code
Description

EINVAL
The value of *base* is not supported.

ERANGE
The conversion caused an overflow.

Example

CELEBS55

```
/* CELEBS55

   This example converts the strings to a long.
   It prints out the converted value and the substring that
   stopped the conversion.

 */
#include <stdlib.h>
#include <stdio.h>

int main(void)
{
   char *string, *stopstring;
   long l;
   int bs;

   string = "10110134932";
   printf("string = %s\n", string);
   for (bs = 2; bs <= 8; bs *= 2)
   {
      l = strtol(string, &stopstring, bs);
      printf("   strtol = %ld (base %d)\n", l, bs);
      printf("   Stopped scan at %s\n\n", stopstring);
      }
}
```

Output

```
string = 10110134932
   strtol = 45 (base 2)
   Stopped scan at 34932

   strtol = 4423 (base 4)
   Stopped scan at 4932

   strtol = 2134108 (base 8)
   Stopped scan at 932
```

Related information

- “stdlib.h” on page 70
- “atof() — Convert character string to double” on page 203
- “atoi() — Convert character string to integer” on page 204
- “atol() — Convert character string to long” on page 204
- “fscanf(), scanf(), sscanf() — Read and format data” on page 625
- “strtod() — Convert character string to double” on page 1765
- “strtoul() — Convert string to unsigned integer” on page 1780

strtold() — Convert character string to long double

Standards

Standards / Extensions	C or C++	Dependencies
C99 Single UNIX Specification, Version 3 C++ TR1 C99	both	z/OS V1R7

Format

```
#define _ISOC99_SOURCE
#include <stdlib.h>

long double strtold(const char *__restrict__ nptr, char **__restrict__ endptr);
```

General description

strtold() converts a part of a character string, pointed to by *nptr*, to long double. The parameter *nptr* points to a sequence of characters that can be interpreted as a numerical value of the type long double.

The strtold() function breaks the string into three parts:

1. An initial, possibly empty, sequence of white-space characters (as specified by isspace()).
2. A subject sequence interpreted as a floating-point constant or representing infinity or NaN.
3. A final string of one or more unrecognized characters, including the terminating null byte of the input string.

The function then attempts to convert the subject string into the floating-point number, and returns the result.

The expected form of the subject sequence is an optional plus or minus sign, then one of the following:

- A non-empty sequence of decimal digits optionally containing a radix character, then an optional exponent part. A radix character is the character that separates the integer part of a number from the fractional part.
- A 0x or 0X, a non-empty sequence of hexadecimal digits optionally containing a radix character, then a base 2 decimal exponent part with a p or P as prefix, a plus or minus sign, then a sequence of at least one decimal digit. (Example [-]0xh.hhhhp+/-d). A radix character is the character that separates the integer part of a number from the fractional part.
- One of INF or INFINITY, ignoring case.
- One of NANQ or NANQ(n-char-sequence), ignoring case.

- One of NANS or NANS(n-char-sequence), ignoring case.
- One of NAN or NAN(n-char-sequence), ignoring case.

See the "scanf Family of Formatted Input Functions" for a description of special infinity and NaN sequences recognized by z/OS formatted input functions in IEEE Binary Floating-Point mode.

The pointer to the last string successfully converted is stored in the object pointed to by *endptr*, provided that *endptr* is not a NULL pointer. If the subject string is empty or it does not have the expected form, then no conversion is performed. The value of *nptr* is stored in the object pointed to by *endptr*.

Returned value

If successful, strtold() returns the value of the floating-point number.

The long double value is hexadecimal floating-point or IEEE Binary Floating-Point format depending on the floating-point mode of the thread invoking the strtold() function. This function uses __isBFP() to determine the floating-point mode of the invoking thread.

In an overflow, strtold() returns +/-HUGE_VALL. In an underflow, it returns 0. If no conversion is performed, strtold() returns 0. In both cases, errno is set to ERANGE, depending on the base of the value.

Related information

- “stdlib.h” on page 70
- “atof() — Convert character string to double” on page 203
- “atoi() — Convert character string to integer” on page 204
- “atol() — Convert character string to long” on page 204
- “fscanf(), scanf(), sscanf() — Read and format data” on page 625
- “__isBFP() — Determine application floating-point format” on page 904
- “strtod() — Convert character string to double” on page 1765
- “strtof() — Convert character string to float” on page 1769
- “strtol() — Convert character string to long” on page 1775
- “strtoul() — Convert string to unsigned integer” on page 1780

strtoll() — Convert string to signed long long

Standards

Standards / Extensions	C or C++	Dependencies
Language Environment C99 Single UNIX Specification, Version 3 C++ TR1 C99	both	OS/390 V2R9

Format

```
#include <stdlib.h>

long long strtoll(const char * __restrict__ nptr, char ** __restrict__ endptr, int base);
```

Compile requirement: Use of this function requires the long long data type. See *z/OS XL C/C++ Language Reference* for information on how to make long long available.

General description

Converts *nptr*, a character string, to a signed long long value.

The function decomposes the entire string into three parts:

1. A sequence of characters, which in the current locale are defined as white space characters. This part may be empty.
2. A sequence of characters interpreted as an unsigned integer in some base notation. This is the *subject sequence*.
3. A sequence of unrecognized characters.

The base notation is determined by *base*, if *base* is greater than zero. If *base* is zero, the base notation is determined by the format of the sequence of characters that follow an optional plus or optional minus sign.

10 Sequence starts with nonzero decimal digit.

8 Sequence starts with 0, followed by a sequence of digits with values from 0 to 7.

16 Sequence starts with either `0x` or `0X`, followed by digits, and letters `A` through `F` or `a` through `f`.

If the base is greater than zero, the subject sequence contains decimal digits and letters, possibly preceded by either a plus or a minus sign. The letters `a` (or `A`) through `z` (or `Z`) represent values from 10 through 36, but only those letters whose value is less than the value of the base are allowed.

When you are using strtoll(), *nptr* should point to a string with the following form:

```
►►──┬─────────────┬──┬─────┬──┬──────┬──┬────────┬──────────────────────────────────────────►◄
    └─white space─┘  ├─ + ─┤  ├─0────┤  └─digits─┘
                     └─ - ─┘  ├─0x───┤
                              └─0X───┘
```

The pointer to the converted characters, even if conversion was unsuccessful, is stored in the object pointed to by *endptr*, as long as *endptr* is not a NULL pointer.

Returned value

If successful, strtoll() returns the converted signed long long value, represented in the string.

If unsuccessful, strtoll() returns 0 if no conversion could be performed. If the correct value is outside the range of representable values, strtoll() returns `LLONG_MAX` (`LONGLONG_MAX`) or `LLONG_MIN` (`LONGLONG_MIN`), according to the sign of the value. If the value of base is not supported, strtoll() returns 0.

If unsuccessful strtoll() sets errno to one of the following values:

Error Code
Description

EINVAL
The value of *base* is not supported.

ERANGE
The conversion caused an overflow.

Related information

- "stdlib.h" on page 70
- "atof() — Convert character string to double" on page 203
- "atoi() — Convert character string to integer" on page 204
- "atol() — Convert character string to long" on page 204
- "fscanf(), scanf(), sscanf() — Read and format data" on page 625
- "strtod() — Convert character string to double" on page 1765
- "strtoul() — Convert string to unsigned integer"

strtoul() — Convert string to unsigned integer

Standards

Standards / Extensions	C or C++	Dependencies
Language Environment ISO C XPG4 XPG4.2 C99 Single UNIX Specification, Version 3	both	

Format

```
#include <stdlib.h>

unsigned long int strtoul(const char * __restrict__ string1,
                          char ** __restrict__ string2, int base);
```

General description

Converts *string1*, a character string, to an unsigned long int value.

The function decomposes the entire string into three parts:

1. A sequence of characters, which in the current locale are defined as white space characters. This part may be empty.
2. A sequence of characters interpreted as an unsigned integer in some base notation. This is the *subject sequence*.
3. A sequence of unrecognized characters.

The base notation is determined by *base*, if base is greater than zero. If base is zero, the base notation is determined by the format of the sequence of characters that follow an optional plus or optional minus sign.

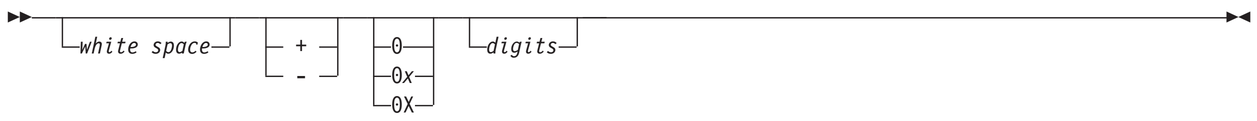
10
Sequence starts with nonzero decimal digit.

8
Sequence starts with 0, followed by a sequence of digits with values from 0 to 7.

16
Sequence starts with either 0x or 0X, followed by digits, and letters A through F or a through f.

If the base is greater than zero, the subject sequence contains decimal digits and letters, possibly preceded by either a plus or a minus sign. The letters a (or A) through z (or Z) represent values from 10 through 36, but only those letters whose value is less than the value of the base are allowed. The function stops reading the string at the first character that it cannot recognize as part of a number. This character can be the first numeric character greater than or equal to the *base*. The strtoul() function sets *string2* to point to the end of the resulting output string if a conversion is performed and provided that *string2* is not a NULL pointer.

When you are using the strtoul() function, *string1* should point to a string with the following form:



If *base* is in the range of 2-36, it becomes the base of the number. If *base* is 0, the prefix determines the base (8, 16, or 10): the prefix 0 means base 8; the prefix 0x or 0X means base 16; using any other digit without a prefix means decimal.

The pointer to the converted characters, even if conversion was unsuccessful, is stored in the object pointed to by *string2*, as long as *string2* is not a NULL pointer.

Returned value

If successful, strtoul() returns the converted unsigned long int value, represented in the string.

If unsuccessful, strtoul() returns 0 if no conversion could be performed. If the correct value is outside the range of representable values, strtoul() returns ULONG_MAX. If the value of base is not supported, strtoul() returns 0.

If unsuccessful strtoul() sets errno to one of the following values:

Error Code
Description

EINVAL
The value of *base* is not supported.

ERANGE
The conversion caused an overflow.

Example

CELEBS56

```
/* CELEBS56

   This example converts the string to an unsigned long value.
   It prints out the converted value and the substring that
   stopped the conversion.

 */
#include <stdio.h>
```

```
#include <stdlib.h>

#define BASE 2

int main(void)
{
   char *string, *stopstring;
   unsigned long ul;

   string = "1000e13 e";
   printf("string = %s\n", string);
   ul = strtoul(string, &stopstring, BASE);
   printf("   strtoul = %ld (base %d)\n", ul, BASE);
   printf("   Stopped scan at %s\n\n", stopstring);
}
```

Output

```
string = 1000e13 e
   strtoul = 8 (base 2)
   Stopped scan at e13 e
```

Related information

- “stdlib.h” on page 70
- “atof() — Convert character string to double” on page 203
- “atoi() — Convert character string to integer” on page 204
- “atol() — Convert character string to long” on page 204
- “fscanf(), scanf(), sscanf() — Read and format data” on page 625
- “strtod() — Convert character string to double” on page 1765
- “strtol() — Convert character string to long” on page 1775

strtoull() — Convert string to unsigned long long

Standards

Standards / Extensions	C or C++	Dependencies
Language Environment C99 Single UNIX Specification, Version 3 C++ TR1 C99	both	OS/390 V2R9

Format

```
#include <stdlib.h>

unsigned long long strtoull(register const char * __restrict__ nptr,
                            char ** __restrict__ endptr, int base);
```

Compile requirement: Use of this function requires the long long data type. See *z/OS XL C/C++ Language Reference* for information on how to make long long available.

General description

Converts *nptr*, a character string, to an unsigned long long value.

The function decomposes the entire string into three parts:

1. A sequence of characters, which in the current locale are defined as white space characters. This part may be empty.

2. A sequence of characters interpreted as an unsigned integer in some base notation. This is the *subject sequence*.
3. A sequence of unrecognized characters.

The base notation is determined by *base*, if base is greater than zero. If base is zero, the base notation is determined by the format of the sequence of characters that follow an optional plus or optional minus sign.

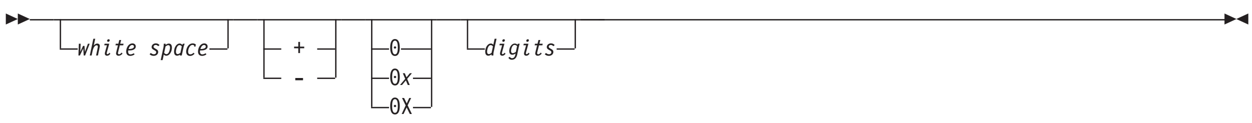
10 Sequence starts with nonzero decimal digit.

8 Sequence starts with 0, followed by a sequence of digits with values from 0 to 7.

16 Sequence starts with either `0x` or `0X`, followed by digits, and letters `A` through `F` or `a` through `f`.

If the base is greater than zero, the subject sequence contains decimal digits and letters, possibly preceded by either a plus or a minus sign. The letters `a` (or `A`) through `z` (or `Z`) represent values from 10 through 36, but only those letters whose value is less than the value of the base are allowed. The function stops reading the string at the first character that it cannot recognize as part of a number. This character can be the first numeric character greater than or equal to the *base*. The strtoull() function sets *endptr* to point to the end of the resulting output string if a conversion is performed and provided that *endptr* is not a NULL pointer.

When you are using the strtoull() function, *nptr* should point to a string with the following form:



If *base* is in the range of 2-36, it becomes the base of the number. If *base* is 0, the prefix determines the base (8, 16 or 10): the prefix 0 means base 8; the prefix `0x` or `0X` means base 16; using any other digit without a prefix means decimal.

The pointer to the converted characters, even if conversion was unsuccessful, is stored in the object pointed to by *endptr*, as long as *endptr* is not a NULL pointer.

Returned value

If successful, strtoull() returns the converted unsigned long long value, represented in the string.

If unsuccessful, strtoull() returns 0 if no conversion could be performed. If the correct value is outside the range of representable values, strtoull() returns `ULLONG_MAX` (`ULONGLONG_MAX`). If the value of base is not supported, strtoull() returns 0.

If unsuccessful strtoull() sets errno to one of the following values:

Error Code
Description

EINVAL
The value of *base* is not supported.

ERANGE
The conversion caused an overflow.

Related information

- "stdlib.h" on page 70
- "atof() — Convert character string to double" on page 203
- "atoi() — Convert character string to integer" on page 204
- "atol() — Convert character string to long" on page 204
- "fscanf(), scanf(), sscanf() — Read and format data" on page 625
- "strtod() — Convert character string to double" on page 1765
- "strtoul() — Convert string to unsigned integer" on page 1780

strtoumax() — Convert character string to uintmax_t integer type

Standards

Standards / Extensions	C or C++	Dependencies
C99 Single UNIX Specification, Version 3 C++ TR1 C99	both	z/OS V1R7

Format

```
#define _ISOC99_SOURCE
#include <inttypes.h>

uintmax_t strtoumax(const char * __restrict__ nptr,
                    char ** __restrict__ endptr, int base);
```

Compile requirement: Function strtoumax() requires long long to be available.

General description

The strtoumax() function converts the string nptr to an uintmax_t integer type. Valid input values for base are 0 and in the range 2-36. The strtoumax() function is equivalent to strtoul() and strtoull(). The only difference being that the return value is of type uintmax_t. See strtoull for more information.

Returned value

If successful, strtoumax() returns the converted uintmax_t value, represented in the string.

If unsuccessful, strtoumax() returns 0 if no conversion could be performed. If the correct value is outside the range of representable values, strtoumax() returns `UINTMAX_MAX`. If the value of base is not supported, strtoumax() returns 0.

If unsuccessful strtoumax() sets errno to one of the following values:

Error Code
Description

EINVAL
The value of *base* is not supported.

ERANGE
The conversion caused an overflow.

Example

```
#define _ISOC99_SOURCE
#include <inttypes.h>
#include <stdlib.h>
#include <stdio.h>

      int main(void)
       {
          uintmax_t j;
          int base = 10;
          char *nptr, *endptr;
          nptr = "20690239864abc";
          printf("string = %s\n", nptr);
          j = strtoumax(nptr, &endptr, base);
          printf("strtoumax = %ju (base %d)\n", j, base);
          printf("Stopped scan at %s\n\n", endptr);
       }

Output

string = 20690239864abc
strtoumax = 20690239864 (base 10)
Stopped scan at abc
```

Related information

- inttypes.h
- strtoimax()
- strtol()
- strtoul()
- strtoll()
- strtoull()
- wcstoimax()
- wcstoumax()

strxfrm() — Transform string

Standards

Standards / Extensions	C or C++	Dependencies
ISO C XPG4 XPG4.2 C99 Single UNIX Specification, Version 3	both	

Format

```
#include <string.h>

size_t strxfrm(char * __restrict__ s1, const char * __restrict__ s2, size_t n);
```

General description

Transforms the string pointed to by *s2* and places the resulting string into the array pointed to by *s1*. The transformation is determined by the program's locale. The transformed string is not necessarily readable, but can be used with the strcmp() or strncmp() functions.

The transformation is such that, if strcmp() or strncmp() were applied to the two transformed strings, the results would be the same as applying the strcoll() function to the two corresponding untransformed strings.

No more than *n* bytes are placed into the area pointed to by *s1*, including the terminating NULL byte. If *n* is zero, *s1* is allowed to be a NULL pointer.

Returned value

If successful, strxfrm() returns the length of the transformed string (excluding the NULL byte). When *n* is zero and *s1* is a NULL pointer, the length returned is the number of bytes minus one required to contain the transformed string.

If unsuccessful, strxfrm() returns (size_t)-1 and sets errno to indicate the error.

Notes:

1. The string returned by strxfrm() contains the weights for each order of the characters within the string. As a result, the string returned may be longer than the input string, and does not contain printable characters.
2. strxfrm() issues a malloc() when the LC_COLLATE category specifies *backward* on the *order_start* keyword, the *substitute* keyword is specified, or the locale has one-to-many mapping. The strxfrm() function will fail if the malloc() fails.
3. If the locale supports double-byte characters (`MB_CUR_MAX` specified as 4), the strxfrm() function validates the multibyte characters, whereas previously the strxfrm() function did not validate the string. The strxfrm() function will fail if the string contains invalid multibyte characters.
4. If MB_CUR_MAX is defined as 4, and no collation is defined for DBCS chars in the current locale, the DBCS characters will collate after the single-byte characters.

Example

CELEBS57

```
/* CELEBS57

   This example prompts the user to input a string of characters, then
   uses strxfrm() to transform the string and return its length.

 */
#include <collate.h>
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
int main(void)
{
  char *string1="string1", *string2="string2";
  char *newstring1, *newstring2;
  int length1, length2;

  length1=strxfrm(NULL, string1, 0);
  length2=strxfrm(NULL, string2, 0);
  if (((newstring1=(char *)calloc(length1+1, 1))==NULL) ||
      ((newstring2=(char *)calloc(length2+1, 1))==NULL))
  {
    printf("insufficient memory\n");
    exit(99);
  }
  if ((strxfrm(newstring1, string1, length1+1) != length1) ||
      (strxfrm(newstring2, string2, length2+1) != length2))
  {
    printf("error in string processing\n");
    exit(99);
  }
  if (strcoll(string1, string2) != strcmp(newstring1, newstring2))
  {
    printf("wrong results\n");
    exit(99);
  }
  printf("correct results\n");
  exit(0);
}
```

Related information

- "string.h" on page 72
- "localeconv() — Query numeric conventions" on page 982
- "setlocale() — Set locale" on page 1552
- "strcmp() — Compare strings" on page 1731
- "strcoll() — Compare strings" on page 1732
- "strncmp() — Compare strings" on page 1751

__superkill() — Sends "super" SIGKILL to terminate target process

Standards

Standards / Extensions	C or C++	Dependencies
z/OS UNIX	both	z/OS V1R6 POSIX(ON)

Format

```
#define _POSIX_SOURCE
#include <signal.h>

int  __superkill(pid_t pid);
```

General description

The __superkill() function generates a more robust version of the SIGKILL signal to the process with *pid* as the process ID. The SIGKILL will be able to break through almost all of the current signal deterrents that can be an obstacle to the normal delivery of a SIGKILL and the resulting termination of the target process.

Function restrictions include:

- Cannot do a __superkill() to a group or specifying PID -1. An attempt to do so will result in a EINVAL/JrNoGroups.
- The superkill will be ignored if the target process has blocked all signals, in which case the __superkill() will not fail but simply be ignored (refer to BPX1SDD syscall in Chapter 2 . Callable services descriptions, *SA23-2281*). Under a multithread environment, as long as BPX1SDD is called on the initial thread, __superkill() will be ignored. The sigprocmask() function cannot be used to block __superkill().
- A regular SIGKILL must be sent, at least 3 seconds, to a process before a superkill. Otherwise the attempt will result in EINVAL/JRSigkillNotSent.
- Runtime option POSIX(ON) is required to be set for this function to work properly.

If the environment is valid, then the target process will be abended with a '422'x abend reason code x'0109' reason code. The abend code will be sent to the first dubbed thread in the process. Under Language Environment, this is almost always the initial processing thread (IPT).

Returned value

When successful, the target process is terminated. Upon failure, __superkill() returns -1 and sets errno to one of the following values:

Value Description

ENIVAL
PID is -1 or a group process ID or the superkill was not sent 3 seconds after the regular SIGKILL.

EPERM
The caller does not have the permission to send the signal to any process that was specified by the process ID parameter.

ESRCH
No process or process groups that correspond to the process ID are found.

svc99() — Access supervisor call

Standards

Standards / Extensions	C or C++	Dependencies
Language Environment	both	

Format

```
#include <stdio.h>

int svc99(__S99parms *string);
```

General description

Provides access to SVC99 on z/OS, which provides ability to:
- Dynamically allocate or deallocate a resource
- Dynamically concatenate or deconcatenate data sets
- Dynamically retrieve information on data sets

To avoid infringing on the user's name space, this nonstandard function has two names. One name is prefixed with two underscore characters, and one name is not. The name without the prefix underscore characters is exposed only when you use LANGLVL(EXTENDED).

To use this function, you must either invoke the function using its external entry point name (for example, the name that begins with two underscore characters), or compile with LANGLVL(EXTENDED). When you use LANGLVL(EXTENDED) any relevant information in the header is also exposed.

The __S99parms structure must be in 31-bit addressable storage. A call to svc99() with 64-bit addressable storage will result in -1 return code.

The __S99TXTPP element needs to be a 32-bits wide pointer to 31-bit addressable storage containing an array of text unit pointers. Each of the text unit pointers must be a 32-bits wide pointer, each pointing to 31-bit addressable storage containing a text unit, or can be a NULL pointer. The last text unit pointer must have its high bit (traditional 31-bit amode high bit) turned on to denote the end of text units has been reached.

The __S99S99X element needs to be a 32-bits wide pointer to 31-bit addressable storage containing the __S99rbx structure, when needed. This is consistent with the __dyn_t structure element __rbx requirement outlined above.

The `__S99parms` structure is defined in `stdio.h`. It has been changed to include the address of the Request Block Extension. The Request Block Extension and the Error Message Parameter list can be used to process the messages returned by SVC99 when an error occurs. To use this feature, you must allocate and initialize these structures.

Table 58. Elements Contained by __S99parms Structure

Field	Type	Value Stored
__S99RBLN	unsigned char	SVC99 length of request block
__S99VERB	unsigned char	SVC99 verb code
__S99FLAG1	unsigned short	SVC99 Flags 1 field
__S99ERROR	unsigned short	SVC99 error code field
__S99INFO	unsigned short	SVC99 information code
__S99TXTPP	void * __ptr32	SVC99 pointer to a list of text unit pointers
__S99S99X	void * __ptr32	SVC99 pointer to the Request Extension Block
__S99FLAG2	unsigned int	SVC99 Flags 2 field for APF authorized programs

Returned value

If the input pointer is NULL, svc99() returns 0 if running under CICS and nonzero otherwise. The nonzero value indicates that svc99() is supported under the current operating system (that is, z/OS non-CICS). If the input is not NULL, svc99() returns -1 if running under CICS (to indicate an error), otherwise it returns a code that results from svc99().

If the input is not NULL, and svc99() is not supported on the system, it returns -1.

Example

CELEBS58

```
/* CELEBS58

   This example uses the svc99() function to dynamically allocate a data
   set called USERID.EXAMPLE.

 */
#define MASK 0x80000000
#define _EXT

#include <stdio.h>
#include <string.h>
```

```
int main(void)
{
    int rc;
    struct __S99struc parmlist;
    char *s[10] = {                        /* array of text pointers */
    /* text units follow */
        "\0\x02\0\x01\0\x0E""USERID.EXAMPLE", /* DSN=EXAMPLE */
        "\0\x05\0\x01\0\x01\x02",             /* DISP=(,CATLG) */
        "\0\x07\0\0",                         /* SPACE=(TRK,.. */
        "\0\x0A\0\x01\0\x03\0\0\x14",         /* primary=20 */
        "\0\x0B\0\x01\0\x03\0\0\x01",         /* secondary=1 */
        "\0\x15\0\x01\0\x05SYSDA",            /* UNIT=SYSDA */
        "\0\x30\0\x01\0\x02\0\x50",           /* BLKSIZE=80 */
        "\0\x3C\0\x01\0\x02\x40\0",           /* DSORG=PS */
        "\0\x42\0\x01\0\x02\0\x50",           /* LRECL=80 */
        "\0\x49\0\x01\0\x01\x80"};            /* RECFM=F */

    memset(&parmlist, 0, sizeof(parmlist));
    parmlist.__S99RBLN = 20;
    parmlist.__S99VERB = 1;           /* verb for dsname allocation       */
    parmlist.__S99FLAG1 = 0x4000;     /* do not use existing allocation   */
    parmlist.__S99TXTPP = s;          /* pointer to pointer to text units */
    s[9] = (char *)((long unsigned) (s[9]) | MASK);

    rc = svc99(&parmlist);
    if (rc != 0)
       printf(" Error code = %d   Information code = %d\n",
           parmlist.__S99ERROR, parmlist.__S99INFO);
}
```

If your user ID starts with one of the letters A-F, you must add two double quotation marks (") before the user ID so that the first letter of the user ID is interpreted as a character rather than as a hexadecimal digit.

The preceding example can be made more readable by using symbolic names and data structures as demonstrated in the example below. The members IEFZB4DB, IEFZB4D0 and IEFZB4D2 of SYS1.MACLIB contain symbolic names that will be familiar to most assembler language programmers.

This next example uses symbolic names taken from these members to define, in z/OS XL C/C++ the text unit representing primary=20 or s[3]. Similar definitions can be made for the remaining text units but will not be given here.

```
#include <stdio.h>
#include <string.h>

#define MASK 0x80000000
#define CHAR_BIT 4
/* Defines one text unit with an integer of size 'bytes' */

#define __S99TUNIT_INT(bytes) struct {
    short unsigned __S99TUKEY;    /* KEY                         */
    short unsigned __S99TUNUM;    /* NO. OF LENGTH+PARM ENTRIES */
    struct {                      /* TEXT ENTRY OF LENGTH+PARM  */
      short unsigned __S99TULNG;  /* LENGTH OF 1ST (ONLY) PARM  */
      unsigned int   __S99TUPAR :     /*          PARAMETER     */
                                  (bytes) * CHAR_BIT;
     } __S99TUENT;
    }
/* initialize by: __S99TUNUM = 1;                                */
/*               __S99TUENT.__S99TULNG = <bytes>;                */
/*               __S99TUENT.__S99TUPAR = <value>;                */

#define __DALPRIME 0x000A           /*PRIMARY SPACE QUANTITY     */
```

```
static const __S99TUNIT_INT(3) primary = {__DALPRIME, 1, 3, 20};

int main(void)
{
    int rc;
    struct __S99struc parmlist;
    memset(&parmlist, 0, sizeof(parmlist));
    void *s[10] = {            /* array of text pointers */
    /* text units follow */
                     .    ,                  /* DSN=EXAMPLE  */
                     .    ,                  /* DISP=(,CATLG)*/
                     .    ,                  /* SPACE=(TRK,..*/
                  &primary,                  /* primary=20   */
                     .    ,                  /* secondary=1  */
                     .    ,                  /* UNIT=SYSDA   */
                     .    ,                  /* BLKSIZE=80   */
                     .    ,                  /* DSORG=PS     */
                     .    ,                  /* LRECL=80     */
                     .    };                 /* RECFM=F      */

    parmlist.__S99RBLN = 20;
    parmlist.__S99VERB = 01;      /* verb for dsname allocation       */
    parmlist.__S99FLAG1 = 0x4000; /* do not use existing allocation   */
    parmlist.__S99TXTPP = s;      /* pointer to pointer to text units */
    s[9] = (char *)((long unsigned) (s[9]) | MASK);

    rc = svc99(&parmlist);
    if (rc != 0)
       printf(" Error code = %d   Information code = %d\n",
           parmlist.__S99ERROR, parmlist.__S99INFO);
}
```

Related information

- "stdio.h" on page 68
- "dynalloc() — Allocate a data set" on page 407
- "dynfree() — Deallocate a data set" on page 415
- "dyninit() — Initialize __dyn_t structure" on page 416
- Requesting Dynamic Allocation Functions in *z/OS MVS Programming: Authorized Assembler Services Guide*

swab() — Copy and swap bytes

Standards

Standards / Extensions	C or C++	Dependencies
XPG4 XPG4.2 Single UNIX Specification, Version 3	both	

Format

```
#define _XOPEN_SOURCE
#include <unistd.h>

void swab(const void *__restrict__src, void *__restrict__dest, ssize_t nbytes);
```

General description

The swab() function copies *nbytes* bytes, which are pointed to by *src* to the object pointed to by *dest*, exchanging adjacent bytes. The *nbytes* argument should be even. If *nbytes* is odd, swab() copies and exchanges *nbytes*-1 bytes and the disposition of the last byte is left unchanged in the target area. If *nbytes* is zero or negative, no copying is performed.

Returned value

swab() returns no values.

Related information

- "unistd.h" on page 82

swapcontext() — Save and restore user context

Standards

Standards / Extensions	C or C++	Dependencies
XPG4.2 Single UNIX Specification, Version 3	both	POSIX(ON)

Format

```
#define _XOPEN_SOURCE_EXTENDED 1
#include <ucontext.h>

int swapcontext(ucontext_t *__restrict__ oucp, const ucontext_t *__restrict__ ucp);
```

General description

The swapcontext() function saves the current user context in the context structure pointed to by *oucp* and restores the user context structure pointed to by *ucp*. swapcontext() is equivalent to getcontext() with the *oucp* argument followed by setcontext() with the *ucp* argument.

Control does not return from the initial invocation of swapcontext(). However, if the saved context is not modified using makecontext(), and a subsequent setcontext() or swapcontext() is issued using the saved context, swapcontext() returns with a 0 return value.

Notes:

1. If the *ucontext* pointed to by *ucp* that is input to swapcontext(), has not been modified by makecontext(), you must ensure that the function that saved that context by calling either getcontext() or swapcontext() does not return before you call swapcontext() to restore that context. Calling swapcontext() after the function that saved the context returns causes unpredictable program behavior.
2. If swapcontext() is used to jump back into an XPLINK routine, any alloca() requests issued by the XPLINK routine after the earlier getcontext() was called and before swapcontext() is called are backed out. All storage obtained by these alloca() requests is freed before the XPLINK routine is resumed.
3. If swapcontext() is used to jump back into a non-XPLINK routine, alloca() requests made after getcontext() and before swapcontext() are not backed out.
4. Do not issue swapcontext() from any type of condition handling routine (for eample, a signal catcher, a Language Environment user condition handler or an exception handler).
5. If *ucp* is pointing to a user context of a different execution stack from the current, the user context should be either a freshly modified one (by

makecontext()) or the most recently saved one (by getcontext() or swapcontext()) when running on its stack.

6. If *ucp* is pointing to a user context of a different execution stack from the current, the current stack is never collapsed and any resource associated with it is never freed after swapcontext() being called.

This function is supported only in a POSIX program.

The <ucontext.h> header file defines the ucontext_t type as a structure that includes the following members:

```
mcontext_t       uc_mcontext     A machine-specific representation
                                 of the saved context.
ucontext_t      *uc_link         Pointer to the context that will
                                 be resumed when this context returns.
sigset_t         uc_sigmask      The set of signals that are blocked
                                 when this context is active.
stack_t          uc_stack        The stack used by this context.
```

Special behavior for C++: If getcontext() and swapcontext() are used to transfer control in a z/OS XL C++ program, the behavior in terms of the destruction of automatic objects is undefined. This applies to both z/OS XL C++ and z/OS XL C++ ILC modules. The use of getcontext() and swapcontext() in conjunction with try(), catch(), and throw() is also undefined.

Do not issue getcontext() in a C++ constructor or destructor, since the saved context would not be usable in a subsequent setcontext() or swapcontext() after the constructor or destructor returns.

Special behavior for XPLINK-compiled C/C++: Restrictions concerning setjmp.h and ucontext.h:

1. All XPLINK programs compiled with the V2R10 or later C compilers that are to run with Language Environment V2R10 or later libraries and use the **jmp_buf**, **sigjmp_buf** or **ucontext_t** types must not be compiled with C headers from Language Environment V2R9 or earlier.
2. Non-XPLINK functions compiled with any level of Language Environment headers must not define **jmp_buf**, **sigjmp_buf** or **ucontext_t** data items and pass them to XPLINK functions that call getcontext(), longjmp(), _longjmp(), setjmp(), _setjmp(), setcontext(), sigsetjmp(), or swapcontext() with these passed-in data items.
3. When __XPLINK__ is defined, the Language Environment V2R10 and later headers define a larger **jmp_buf**, **sigjmp_buf** or **ucontext_t** area that is required by setjmp(), getcontext(), and related functions when they are called from an XPLINK routine. If __XPLINK__ is not defined, the Language Environment V2R10 and later headers define a shorter **jmp_buf**, **sigjmp_buf** or **ucontext_t** area. The Language Environment headers before V2R10 also define the shorter version of these data areas. If an XPLINK function calls setjmp(), getcontext() or similar functions with a short **jmp_buf**, **sigjmp_buf** or **ucontext_t** area, a storage overlay or program check may occur when the C library tries to store past the end of the passed-in (too short) data area.

Returned value

If successful, swapcontext() does not return from the initial invocation. If the unmodified saved context is later restored, swapcontext() returns 0.

If unsuccessful, swapcontext() returns -1.

There are no errno values defined.

Example

This example uses two contexts. It creates the first, *fcontext*, in main with the getcontext() statement, and invokes the function *func*. It invokes the function with the swapcontext() statement, saving the context at that point in the second context, *mcontext*. The function returns to the point of the swapcontext() using the setcontext() statement and specifying *mcontext* as the context.

```
/* This example shows the usage of swapcontext().                    */

#define _XOPEN_SOURCE_EXTENDED 1
#include <stdlib.h>
#include <stdio.h>
#include <ucontext.h>
#include <errno.h>

 #ifdef _LP64
  #define STACK_SIZE 2097152+16384 /* large enough value for AMODE 64 */
 #else
  #define STACK_SIZE 16384  /* AMODE 31 addressing*/
 #endif
void func(int);

ucontext_t fcontext,mcontext;
int x = 0;

int main(void) {

  int  value = 1;

  getcontext(&fcontext);
  if ((fcontext.uc_stack.ss_sp = (char *) malloc(STACK_SIZE)) != NULL) {
    fcontext.uc_stack.ss_size = STACK_SIZE;
    fcontext.uc_stack.ss_flags = 0;
   errno = 0;
    makecontext(&fcontext,func,1,value);
   if (errno != 0){
    perror("Error reported by makecontext()");
    return -1;    /* Error occurred exit */
 }
   }
  else {
    perror("not enough storage for stack");
    abort();
  }
  printf("context has been built\n");
  swapcontext(&mcontext,&fcontext);
  if (!x) {
    perror("incorrect return from swapcontext");
    abort();
  }
  else {
    printf("returned from function\n");
  }

}

void func(int arg) {

  printf("function called with value %d\n",arg);
  x++;
  printf("function returning to main\n");
  setcontext(&mcontext);

}
```

Output

```
context has been built
function called with value 1
function returning to main
returned from function
```

Related information

- "ucontext.h" on page 82
- "getcontext() — Get user context" on page 694
- "longjmp() — Restore stack environment" on page 1005
- "_longjmp() — Nonlocal goto" on page 1008
- "makecontext() — Modify user context" on page 1028
- "setcontext() — Restore user context" on page 1524
- "setjmp() — Preserve stack environment" on page 1546
- "_setjmp() — Set jump point for a nonlocal goto" on page 1549
- "siglongjmp() — Restore the stack environment and signal mask" on page 1640
- "sigsetjmp() — Save stack environment and signal mask" on page 1659

swprintf() — Format and write wide characters

The information for this function is included in "fwprintf(), swprintf(), wprintf() — Format and write wide characters" on page 672.

swscanf() — Read a wide-character string

The information for this function is included in "fwscanf(), swscanf(), wscanf() — Convert formatted wide-character input" on page 679 section .

symlink() — Create a symbolic link to a path name

Standards

Standards / Extensions	C or C++	Dependencies
POSIX.1a XPG4.2 Single UNIX Specification, Version 3	both	

Format

```
#define _POSIX1_SOURCE 2
#include <unistd.h>

int symlink(const char *pathname, const char *slink);
```

General description

Creates the symbolic link named by *slink* with the file specified by *pathname*. File access checking is not performed on the file *pathname*, and the file need not exist. In addition, a symbolic link can cross file system boundaries.

A symbolic link path name is resolved in this fashion:

- When a component of a path name refers to a symbolic link rather than to a directory, the path name contained in the symbolic link is resolved.
- If the path name in the symbolic link begins with / (slash), the symbolic link path name is resolved relative to the process root directory.

  If the path name in the symbolic link does not start with / (slash), the symbolic link path name is resolved relative to the directory that contains the symbolic link.
- If the symbolic link is the last component of a path name, it may or may not be resolved. Resolution depends on the function using the path name. For example, rename() does not resolve a symbolic link when it appears as the final component of either the new or old path name. However, `open` does resolve a symbolic link when it appears as the last component.
- If the symbolic link is not the last component of the original path name, remaining components of the original path name are resolved relative to the symbolic link.
- When a / (slash) is the last component of a path name and it is preceded by a symbolic link, the symbolic link is always resolved.

Because the mode of a symbolic link cannot be changed, its mode is ignored during the lookup process. Any files and directories to which a symbolic link refers are checked for access permission.

Returned value

If successful, symlink() returns 0.

If unsuccessful, symlink() returns -1, does not affect any file it names, and sets errno to one of the following values:

Error Code
    Description

EACCES
    A component of the *slink* path prefix denies search permission, or write permission is denied in the parent directory of the symbolic link to be created.

EEXIST
    The file named by *slink* already exists.

EINVAL
    This may be returned for either of these reasons:

- There is a NULL character in *pathname*.
- *slink* has a slash as its last component, which indicates that the preceding component will be a directory. A symbolic link cannot be a directory.

EIO **Added for XPG4.2:** An I/O error occurred while reading from the file system.

ELOOP
    A loop exists in symbolic links. This error is issued if more than POSIX_SYMLOOP symbolic links are encountered during resolution of the *slink* argument.

ENAMETOOLONG
    pathname is longer than **PATH_MAX** characters, or some component of *pathname* is longer than **NAME_MAX** characters while

_POSIX_NO_TRUNC is in effect. For symbolic links, the length of the path name string substituted for a symbolic link exceeds **PATH_MAX**. The **PATH_MAX** and **NAME_MAX** values can be determined with pathconf().

ENOENT
Added for XPG4.2: A component of *slink* does not name an existing file or *slink* is an empty string.

ENOSPC
The new symbolic link cannot be created because there is no space left on the file system that will contain the symbolic link.

ENOTDIR
A component of the path prefix of *slink* is not a directory.

EROFS
The file *slink* cannot reside on a read-only system.

Example

```
/* This example works only under z/OS XL C, not z/OS XL C++  */
#define _POSIX1_SOURCE 2
#include <fcntl.h>
#include <sys/stat.h>
#include <sys/types.h>
#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>

main() {
  char fn[]="test.file";
  char sln[]="test.symlink";
  int fd;

  if ((fd = creat(fn, S_IWUSR)) < 0)
    perror("creat() error");
  else {
    close(fd);
    puts("before symlink()");
    system("ls -il test.*");
    if (symlink(fn, sln) != 0) {
      perror("symlink() error");
      unlink(fn);
    }
    else {
      puts("after symlink()");
      system("ls -il test.*");
      unlink(fn);
      puts("after first unlink()");
      system("ls -il test.*");
      unlink(sln);
    }
  }
}
```

Output

```
before symlink()
 4030 --w-------   1 MVSUSR1  SYS1     0 Apr 20 13:57 test.file

after symlink()
 4030 --w-------   1 MVSUSR1  SYS1     0 Apr 20 13:57 test.file
 4031 l---------   1 MVSUSR1  SYS1     9 Apr 20 13:57 test.symlink -> test.file
after first unlink()
 4031 l---------   1 MVSUSR1  SYS1     9 Apr 20 13:57 test.symlink -> test.file
```

Related information

- "unistd.h" on page 82
- "link() — Create a link to a file" on page 970
- "readlink() — Read the value of a symbolic link" on page 1387
- "unlink() — Remove a directory entry" on page 1962

sync() — Schedule file system updates

Standards

Standards / Extensions	C or C++	Dependencies
XPG4.2 Single UNIX Specification, Version 3	both	

Format

```
#define _XOPEN_SOURCE_EXTENDED 1
#include <unistd.h>

void sync(void);
```

General description

The sync() function causes all information in memory that updates file systems to be scheduled for writing out to all file systems.

The writing, although scheduled, is not necessarily complete upon return from sync().

Returned value

sync() returns no values.

No errors are defined.

Related information

- "unistd.h" on page 82
- "fsync() — Write changes to direct-access storage" on page 655

sysconf() — Determine system configuration options

Standards

Standards / Extensions	C or C++	Dependencies
POSIX.1 XPG4 XPG4.2 Single UNIX Specification, Version 2 Single UNIX Specification, Version 3	both	

Format

```
#define _POSIX_SOURCE
#include <unistd.h>

long sysconf(int name);
```

General description

Determines the value of a configurable system option.

`int` *name*

Specifies the system configuration option to be obtained. The value of *name* can be any one of the following set of symbols defined in the unistd.h header file, each corresponding to a system configuration option:

The following are available when _POSIX_SOURCE is defined.

_SC_ARG_MAX
Represents **ARG_MAX**, as defined by the values returned by sysconf(), the maximum number of bytes of arguments and environment data that can be passed in an exec function.

_SC_CHILD_MAX
Represents CHILD_MAX, as defined by the values returned by sysconf(), the maximum number of processes that a real user ID (UID) may have running simultaneously.

_SC_CLK_TCK
Represents the CLK_TCK macro defined in the time.h header file: the number of clock ticks in a second.

_SC_JOB_CONTROL
Represents the _POSIX_JOB_CONTROL macro that can be defined in the unistd.h header file. This indicates that certain job control operations are implemented by this version of the operating system. If _POSIX_JOB_CONTROL is defined, various functions (for example, setpgid()) have more functionality than when the macro is not defined.

_SC_NGROUPS_MAX
Represents **NGROUPS_MAX**, as defined by the values returned by sysconf(), the maximum number of supplementary group IDs (GIDs) that can be associated with a process.

_SC_OPEN_MAX
Represents **OPEN_MAX**, as defined by the values returned by sysconf(), the maximum number of files that a single process can have open at one time.

_SC_SAVED_IDS
Represents the _POSIX_SAVED_IDS macro, which may be defined in unistd.h header file, indicating that this POSIX implementation has a saved set UID and a saved set GID. This symbol affects the behavior of such functions as setuid() and setgid().

_SC_STREAM_MAX
Represents the _STREAM_MAX macro, which may be defined in the unistd.h header file, indicating the maximum number of streams that a process can have open at one time.

_SC_THREADS_MAX_NP
Represents the THREAD_MAX macro, as defined by the values returned by sysconf(), the maximum number of concurrent threads processed by pthread_create(), including running, queued, and exited undetached threads in the caller's process.

_SC_THREAD_TASKS_MAX_NP
Represents the THREAD_TASKS_MAX macro, as defined by the values returned by sysconf(), the maximum number of MVS tasks simultaneously in use for threads processed by pthread_create() in the caller's process.

_SC_TTY_GROUP
Retrieves the group number associated with the TTYGROUP() initialization parameter.

_SC_TZNAME_MAX
Represents the _TZNAME_MAX macro, which may be defined in the unistd.h header file, indicating the maximum length of the name of a time zone.

_SC_VERSION
Represents the _POSIX_VERSION macro, which will be defined in the unistd.h header file, indicating the version of the POSIX.1 standard that the system conforms to.

In addition to the symbols exposed by _POSIX_SOURCE, the following are visible when _XOPEN_SOURCE is defined:

_SC_XOPEN_CRYPT
Represents _XOPEN_CRYPT, the implementation supports the X/Open Encryption Option Group.

_SC_XOPEN_VERSION
Represents _XOPEN_VERSION, integer value indicating version of the X/Open Portability Guide to which the implementation conforms.

_SC_PASS_MAX

Represents PASS_MAX, as defined by the values returned by sysconf(), the maximum number of bytes allowed in a password, PassTicket, or password phrase.

Note:

It was part of the Legacy Feature in Single UNIX Specification, Version 2, but has been withdrawn and is not supported as part of Single UNIX Specification, Version 3.

If it is necessary to continue using this function in an application written for Single UNIX Specification, Version 3, define the feature test macro _UNIX03_WITHDRAWN before including any standard system headers. The macro exposes all interfaces and symbols removed in Single UNIX Specification, Version 3.

In addition to the symbols exposed by _XOPEN_SOURCE, the following are visible when _XOPEN_SOURCE_EXTENDED is defined to be 1:

_SC_PAGE_SIZE
Returns the current page size in bytes.

_SC_PAGESIZE
Returns the current page size in bytes.

In addition to the symbols exposed by _POSIX_SOURCE, the following are visible when _POSIX_C_SOURCE is defined to be 2:

_SC_2_C_BIND
Represents _POSIX2_C_BIND, the implementation supports the C-Language Binding option.

_SC_2_C_DEV
Represents _POSIX2_C_DEV, the implementation supports the C-Language Development Utilities option.

_SC_2_LOCALEDEF
Represents _POSIX2_LOCALEDEF, the implementation supports the creation of locales by the localedef utility.

_SC_2_UPE
Represents _POSIX2_UPE, the implementation supports the User Portability Utilities option. .

_SC_2_VERSION
Represents _POSIX2_VERSION, integer value indicating version of the Shell and Utilities to which the implementation conforms.

In addition to the symbols exposed by _POSIX_C_SOURCE defined to be 2, the following are visible when _POSIX_C_SOURCE is defined to be 200112L:

_SC_HOST_NAME_MAX
Represents HOST_NAME_MAX, Maximum length of a host name (not including the terminating null) as returned from the gethostname() function.

_SC_IPV6
Represents _POSIX_IPV6, the implementation supports the IPv6 option.

_SC_LOGIN_NAME_MAX
Represents LOGIN_NAME_MAX, Maximum length of a login name.

_SC_READER_WRITER_LOCKS
Represents _POSIX_READER_WRITER_LOCKS, the implementation supports the Read-Write Locks option. This is always set to a value greater than zero if the Threads option is supported.

_SC_REGEXP
Represents _POSIX_REGEXP, the implementation supports the Regular Expression Handling option.

_SC_SHELL
Represents _POSIX_SHELL, the implementation supports the POSIX shell.

_SC_SYMLOOP_MAX
Represents SYMLOOP_MAX, maximum number of symbolic links that can be reliably traversed in the resolution of a path name in the absence of a loop.

_SC_THREAD_ATTR_STACKSIZE
Represents _POSIX_THREAD_ATTR_STACKSIZE, the implementation supports the Thread Stack Size Attribute option.

_SC_THREAD_KEYS_MAX
Represents PTHREAD_KEYS_MAX, maximum number of data keys that can be created by a process.

_SC_THREAD_PROCESS_SHARED
Represents _POSIX_THREAD_PROCESS_SHARED, the implementation supports the Thread Process-Shared Synchronization option.

_SC_THREAD_SAFE_FUNCTIONS
Represents _POSIX_THREAD_SAFE_FUNCTIONS, the implementation supports the Thread-Safe Functions option.

_SC_THREAD_STACK_MIN
Represents PTHREAD_STACK_MIN, minimum size in bytes of thread stack storage.

_SC_THREAD_THREADS_MAX
Represents PTHREAD_THREADS_MAX, maximum number of threads that can be created per process.

_SC_THREADS
Represents _POSIX_THREADS, the implementation supports the Threads option.

_SC_TTY_NAME_MAX
Represents TTY_NAME_MAX, maximum length of terminal device name.

_SC_V6_ILP32_OFF32
Represents _POSIX_V6_ILP32_OFF32, the implementation provides a C-language compilation environment with 32-bit int, long, pointer, and off_t types.

_SC_V6_ILP32_OFFBIG
Represents _POSIX_V6_ILP32_OFFBIG, the implementation provides a C-language compilation environment with 32-bit int, long, and pointer types and an off_t type using at least 64 bits.

_SC_V6_LP64_OFF64
Represents _POSIX_V6_LP64_OFF64, the implementation provides a C-language compilation environment with 32-bit int and 64-bit long, pointer, and off_t types.

_SC_V6_LPBIG_OFFBIG
Represents _POSIX_V6_LPBIG_OFFBIG, the implementation provides a C-language compilation environment with an int type using at least 32 bits and long, pointer, and off_t types using at least 64 bits.

_SC_XOPEN_LEGACY
Represents _XOPEN_LEGACY, the implementation supports the Legacy Option Group.

The following symbols are available under _XOPEN_SOURCE 500:

_SC_GETPW_R_SIZE_MAX
Maximum size of getpwuid_r() and getpwnam_r() data buffers

_SC_GETGR_R_SIZE_MAX
Maximum size of getgrgid_r() and getgrnam_r() data buffers

Returned value

If successful, sysconf() returns the value associated with the specified option.

If the variable corresponding to *name* exists but is not supported by the system, sysconf() returns -1 but does not change the value of errno. If sysconf() fails in some other way, it returns -1.

If unsuccessful, sysconf() sets errno to one of the following values:

Error Code
Description

EINVAL
The value specified for the *name* argument is incorrect.

Example

CELEBS61

```
/* CELEBS61

   This example determines the value of ARG_MAX.

 */
#define _POSIX_SOURCE
#include <stdio.h>
#include <unistd.h>
#include <errno.h>

main() {
  long result;

  errno = 0;
  puts("examining ARG_MAX limit");
  if ((result = sysconf(_SC_ARG_MAX)) == -1)
    if (errno == 0)
      puts("ARG_MAX is not supported.");
    else perror("sysconf() error");
  else
    printf("ARG_MAX is %ld\n", result);
}
```

Output

```
examining ARG_MAX limit
ARG_MAX is 1048576
```

Related information

- “unistd.h” on page 82
- “clock() — Determine processor time” on page 285
- “exec functions” on page 436

syslog() — Send a message to the control log

Standards

Standards / Extensions	C or C++	Dependencies
XPG4.2 Single UNIX Specification, Version 3	both	

Format

```
#define _XOPEN_SOURCE_EXTENDED 1
#include <syslog.h>

void syslog(int priority, const char *message, ... /* argument */);
```

General description

The syslog() function sends a message to an implementation-specific logging facility, which loads it in an appropriate system log, writes it to the system console, forwards it to a list of users, or forwards it to the logging facility on another host over the network. The logged message includes a message header and a message body. The message header consists of a facility indicator, a severity indicator, a timestamp, a tag string, and optionally the process ID. The process ID is surrounded by square brackets. The code point values for the square brackets are taken from code page IBM-1047. The value for the left square bracket is 0xAD. The value for the right square bracket is 0xBD.

The message body is generated from the *message* and following arguments in the same manner as if these were arguments to the printf() function, except that occurrences of *%m* in the format string pointed to by the *message* argument are replaced by the error message string associated with the current value of errno. A trailing newline character is added if needed.

Note: If the total length of the format string and the parameters is greater than 4096 bytes, then the results are undefined.

Values of the *priority* argument are formed by ORing together a severity level values and an option facility value. If no facility value is specified, the current default facility value is used. Possible values of severity level include:

LOG_ALERT
A condition that should be corrected immediately, such as a corrupted system database.

LOG_CRIT
Critical conditions, such as hard device errors.

LOG_DEBUG
Messages that contain information normally of use only when debugging a program.

LOG_EMERG
A Panic condition. This is normally broadcast to all processes.

LOG_ERR
Errors.

LOG_INFO
Informational messages.

LOG_NOTICE
Conditions that are not error conditions, but that may require special handling.

LOG_WARNING
Warning messages.

The facility indicates the application or system component generating the message. Possible facility values include:

LOG_USER
Message generated by random processes. This is the default facility identifier if none is specified.

LOG_LOCAL0
Reserved for local use.

LOG_LOCAL1
Reserved for local use.

LOG_LOCAL2
Reserved for local use.

LOG_LOCAL3
Reserved for local use.

LOG_LOCAL4
Reserved for local use.

LOG_LOCAL5
Reserved for local use.

LOG_LOCAL6
Reserved for local use.

LOG_LOCAL7
Reserved for local use.

Returned value

syslog() returns no values.

Related information

- “syslog.h” on page 72
- “closelog() — Close the control log” on page 292
- “fprintf(), printf(), sprintf() — Format and write data” on page 588
- “openlog() — Open the system control log” on page 1162
- “setlogmask() — Set the mask for the control log” on page 1561

system() — Execute a command

Standards

Standards / Extensions	C or C++	Dependencies
ISO C POSIX.1 XPG4 XPG4.2 IEEE Std 1003.1, 2004 Edition	both	

Format

```
#include <stdlib.h>

int system(const char *string);
```

General description

The system() function has two different behaviors. These behaviors are designated as ANSI system() and POSIX system(). The ANSI system() behavior is based on the ISO C standard definition of the function, whereas the POSIX system() behavior is based on the POSIX standard definition. The ANSI system() behavior is used when a) running POSIX(OFF) or b) when running POSIX(ON) and environment variable __POSIX_SYSTEM is set to NO. Otherwise the POSIX system() behavior is used.

Restriction:

- The system() function is not supported under CICS. If the *string* argument is NULL, system() returns 0 since there is no command processor under CICS, otherwise it returns -1.

ANSI system() function:

Note: In this topic, MVS specifically refers to MVS batch (excluding batch TSO/E), whereas TSO/E includes both batch TSO/E (IKJEFT01 as the program specified on the JCL EXEC statement) and interactive TSO/E (which is TSO/E at a terminal).

Using the ANSI system() function, you can call commands, EXECs, CLISTs, or executable modules under MVS and TSO/E. You cannot use the ANSI system() function to invoke z/OS UNIX services shell programs.

The *string* argument can take one of the two formats:

command-line

A string with TSO/E command line syntax:

```
command_name parm1 parm2 ...
```

Example:

```
system("user_pgm1 1234 abcd xyz");
```

If the *string* argument is in the command-line format, the system() function passes the given string to the command processor, if available, for execution.

named-program

A string, of the following form, with no embedded blanks except in the PARM area.

```
"PGM=program_name[,PARM='....']"
```

Example:

```
system("PGM=user_pgm1, PARM='1234 abcd xyz'");
```

If the *string* argument is a named-program format, the system() function calls program_name with the parameters following “PARM=”, if any.

The two formats are supported under both MVS and TSO/E, but not all targets can be called from all environments. For example, TSO/E commands cannot be called in an MVS environment. As a result, the two formats are equivalent under MVS, but are different under TSO/E. The details of each are provided below. For maximum portability when invoking executable modules under the different environments, use the named-program format.

If the specified executable module is a z/OS XL C or z/OS XL C++ module, full initialization and termination will be performed: including, but not limited to, automatic closing of files and releasing of fetched modules. In addition, if the ANSI system() call uses the named-program format under either MVS or TSO/E, or the command-line format under MVS, information can be passed across the program boundary using memory files. Memory files are not removed until either the highest level (root) program in the call chain terminates or the clrmemf() function is used. Standard streams are also shared in this environment.

ANSI MVS considerations: Under MVS, the ANSI system() function accepts either command-line or named-program format strings. However, the command-line string is restricted to specifying only executable modules (that is, TSO/E commands, EXECs, and CLISTs cannot be specified). Because of this restriction, both formats provide the same functionality.

In the case of either a command-line or named-program string, the ANSI system() function will search the usual MVS sources (STEPLIB/JOBLIB concatenation, Link Pack Area (LPA), Extended Link Pack Area (ELPA), and the link libraries) for the specified program name. The LINK SVC is used to give control to the program.

Under MVS, using ATTACH instead of ANSI system() will prevent you from sharing memory files or standard streams between the programs.

ANSI TSO/E considerations: Under TSO/E, the ANSI system() function accepts either command-line or named-program format strings.

Command-line format strings are presented to the TSO/E command processor and can be used to execute user modules, TSO/E commands, EXECs or CLISTs. If there is any ambiguity as to what is to be run when a command-line format string is supplied, the hierarchy is:

1. TSO/E command or user module
2. CLIST or EXEC

Therefore, if a command-line format string is used and a CLIST or EXEC exists with the same name as a TSO/E command or user module, the TSO/E EXEC command must be used to specifically invoke the CLIST or EXEC.

If the command-line format string is used to call a user module, it effectively uses ATTACH to execute the program. As with MVS, when using ATTACH, memory files and standard streams are not shared between the programs. This is the reason that named-program format strings should be used for maximum portability. The named-program format provides the same memory file and standard stream sharing in both the MVS and TSO/E environments.

Notes:

1. If an executable module is placed in the STEPLIB or ISPLLIB (under ISPF), TSO/E will allow the module to be activated as a command. Recall from the discussion above, that, if a CLIST or EXEC has the same name, the module would be activated first, due to the hierarchy rules. Activating a module as a command involves a different input interface. If required, you can use the CALL command to activate a module that is not prepared to take the TSO/E command input format. z/OS XL C and z/OS XL C++ modules can be called as TSO/E commands.
2. A module that exists in the Link Pack Area (LPA) or Extended Link Pack Area (ELPA) and not in the STEPLIB concatenation (ISPLLIB on ISPF) will not be activated as a TSO/E command, and is treated as an executable module.

Named-program format strings are not presented to the TSO/E command processor and can only be used to execute user modules.

POSIX system() function: Using the POSIX system() function, you can call z/OS UNIX services shell programs. You cannot use the POSIX system() function to call commands, EXECs, CLISTs, or executable modules under MVS and TSO/E.

The POSIX system() function passes *string* to the **sh** shell command for execution. The environment is established by the runtime library through a spawn() of the shell.

The POSIX system() function ignores the SIGINT and SIGQUIT signals, and blocks the SIGCHILD signal while it waits for the command specified by *string* argument to end.

Special considerations for POSIX C: The system() function has these additional considerations:

- A program running with POSIX(ON) can call another program that will also use POSIX(ON) only if the calling program uses the POSIX system() function to invoke the called program out of the shell. Using the ANSI system() function to call a program that will also use POSIX(ON) will result in message CEE3648S being issued, followed by ABENDU4093-AC.
- A program running with POSIX(ON) can receive signals other than SIGINT, SIGQUIT, or SIGCHLD while the POSIX system() function is waiting for the shell command to complete. If there is a signal catcher registered for the signal, it will be invoked immediately. If the signal catcher calls siglongjmp() or setcontext() to pass control back to the application, the SIGINT and SIGQUIT signals will remain ignored, and the SIGCHLD signal will remain blocked.
- If the calling program is the child of a forked process, it cannot use the ANSI system() function to run TSO/E commands since the TSO/E address space is not available.
- If the calling program was invoked using one of the exec or spawn functions, it cannot use the ANSI system() function to run TSO/E commands since the TSO/E address space is not available.
- The system() function is not thread-safe. It cannot be called simultaneously from more than one thread. A multithreaded application must ensure that no more than one system() call is ever outstanding from the various threads. Results are undefined if this restriction is violated..
- When using a signal handler and setting the default handler for SIGCHLD to be SIG_IGN, an errno value of ECHILD will be returned. This is due to the speed at which the process executes and finishes before system() can call waitpid() to wait for the child process to end. In this case, the ECHILD can be ignored because the process has already completed successfully and returned.

Note: If an application invokes a z/OS UNIX service shell command or utility that performs terminal I/O, the command may fail due to the z/OS UNIX services shell file descriptors not being initialized. z/OS UNIX files for terminal I/O must be defined. An example of how these can be defined in a C application are as follows:

```
stdin = fopen("/tmp/sys.stdin","r");
stdout = fopen("/tmp/sys.stdout","w");
stderr = fopen("/tmp/sys.stderr","w");
```

See z/OS XL C/C++ applications with z/OS UNIX System Services C functions in topic 1.9 for more information about using POSIX support.

Mixed environments across an ANSI system() call: The mixing of z/OS Language Environment, C/370 Library Version 1 or Version 2, and System Programming C (SPC) environments across a system() call is not supported. Whichever of these environments is active when the first system() call is made is the only one that is tolerated in the system() call chain. Results are undefined if this restriction is violated.

Returned value

If the *string* argument is a NULL pointer, the system() function returns nonzero if a command processor exists, or 0 if one does not exist.

ANSI MVS considerations: The returned value from ANSI system() will be that from the user module, if successfully called. If system() cannot call the specified module, the returned value is -1 and errno is set appropriately.

ANSI TSO/E considerations: The returned value from ANSI system() will be nonzero if the command processor cannot execute the command or user module. The macros __abendcode and __rsncode will contain the abend code and reason code from a failing TSO/E command, EXEC, or CLIST.

POSIX considerations: If the *string* argument is a NULL pointer, the POSIX system() function returns nonzero. If the *string* argument is not NULL, the POSIX system() function returns the termination status of the command language interpreter in the format specified by waitpid(). If a child process cannot be created, or if the termination status for the command language interpreter cannot be obtained, system() returns -1.

Note: When using a signal handler and setting the default handler for SIGCHLD to be SIG_IGN, an errno value of ECHILD will be returned. This is due to the speed at which the process executes and finishes before system() can call waitpid() to wait for the child process to end. In this case, the ECHILD can be ignored because the process has already completed successfully and returned.

If system() returns -1, errno may be set to one of the following:

Error Code
    Description

EAGAIN
    There are insufficient resources to create another process, or the maximum number of processes you can run has been reached.

ECHILD
    The new process finished before system() could call waitpid() to wait for the child process to end. This error can be ignored because the child process has already completed successfully and returned.

ENOMEM
    The process requires more space than is available.

ENOSYS
    The ANSI system() function was requested from an AMODE 64 application.

Example

```
/* This example illustrates how to use system() to execute a command which
   returns the time.  The example works only under TSO.
 */
#include <stdlib.h>

int main(void)
{
    int rc;
    rc = system("time");
    exit(0);
}
```

```
/* This example may only be used in a POSIX program.  */
#include <stdlib.h>

int main(void)
{
    int result;

    result = system("date | tee result.log");
}
```

Related information

- "stdlib.h" on page 70
- "clrmemf() — Clear memory files" on page 293
- "signal() — Handle interrupts" on page 1643

t_accept() — Accept a connect request

Standards

Standards / Extensions	C or C++	Dependencies
XPG4.2	both	

Format

```
#define _XOPEN_SOURCE_EXTENDED 1
#include <xti.h>

int t_accept(int fd, int resfd, struct t_call *call);
```

General description

t_accept() is issued by a transport user to accept a connect request. The parameter *fd* identifies the local transport endpoint where the connect indication arrived. *resfd* specifies the local transport endpoint where the connection is to be established, and *call* contains information required by the transport provider to complete the connection. The parameter *call* points to a *t_call* structure which contains the following members:

```
struct netbuf   addr;
struct netbuf   opt;
struct netbuf   udata;
int             sequence;
```

In *call*, *addr* is the protocol address of the calling transport user. *opt* indicates any options associated with the connection. *udata* points to any user data to be returned to the caller, and *sequence* is the value returned by t_listen() that uniquely associates the response with a previously received connect indication. The address of the caller, *addr* may be NULL (length zero). Where *addr* is not NULL, then it may optionally be checked by XTI.

A transport user may accept a connection on either the same, or on a different, local transport endpoint than the one on which the connect indication arrived. Before the connection can be accepted on the same endpoint (*resfd==fd*), the user must have responded to any previous connect indications received on that transport endpoint (using t_accept() or t_snddis()). Otherwise, t_accept() fails and sets *t_errno* to TINDOUT.

If a different transport endpoint is specified (*resfd*!=*fd*), then the user may or may not choose to bind the endpoint before the t_accept() is issued. If the endpoint is not bound before the t_accept(), then the transport provider will automatically bind it to the same protocol address *fd* is bound to. If the transport user chooses to bind the endpoint it must be bound to a protocol address with a *qlen* of zero and must be in the T_IDLE state before the t_accept() is issued.

The call to t_accept() will fail with *t_errno* set to TLOOK if there are indications (for example, connect or disconnect) waiting to be received on the endpoint *fd*.

Return of user data over a connection accept is not supported under TCP, so the *udata* field is always meaningless.

When the user does not indicate any option (*call->opt.len* == 0) it is assumed that the connection is to be accepted unconditionally. The transport provider may choose options other than the defaults to ensure that the connection is accepted successfully.

Due to implementation restrictions, behavior is undefined if a different process accepts a connection pending on an endpoint than obtained it (with t_listen).

Valid states: *fd*: T_INCON *resfd* (*fd*!=*resfd*): T_IDLE

Returned value

If successful, t_accept() returns 0.

If unsuccessful, t_accept() returns -1 and sets t_errno to one of the following values:

Error Code
Description

TACCES
The user does not have permission to accept a connection on the responding transport endpoint or to use the specified options.

TBADADDR
The specified protocol address was in an incorrect format or contained illegal information.

TBADDATA
The amount of user data specified was not within the bounds allowed by the transport provider.

TBADF
The file descriptor *fd* or *resfd* does not refer to a transport endpoint.

TBADOPT
The specified options were in an incorrect format or contained illegal information.

TBADSEQ
An invalid sequence number was specified.

TINDOUT
The function was called with *fd==resfd* but there are outstanding connection indications on the endpoint. Those other connection indications must be handled either by rejecting them using t_snddis (3) or accepting them on a different endpoint using t_accept (3).

TLOOK
An asynchronous event has occurred on the transport endpoint referenced by fd and requires immediate attention.

TNOTSUPPORT
This function is not supported by the underlying transport provider.

TOUTSTATE
The function was called in the wrong sequence on the transport endpoint referenced by fd, or the transport endpoint referred to by resfd is not in the appropriate state.

TPROTO
This error indicates that a communication problem has been detected between XTI and the transport provider for which there is no other suitable XTI (*t_errno*).

TPROVMISMATCH
The file descriptors fd and resfd do not refer to the same transport provider.

TRESADDR
This transport provider requires both *fd* and *resfd* to be bound to the same address. This error results if they are not.

TRESQLEN
The endpoint referenced by *resfd* (where *resfd* != *fd*) was bound to a protocol address with a *qlen* that is greater than 0.

TSYSERR
A system error has occurred during execution of this function.

Related information

- "xti.h" on page 87
- "t_connect() — Establish a connection with another transport user" on page 1838
- "t_getstate() — Get the current state" on page 1876
- "t_listen() — Listen for a connect indication" on page 1883
- "t_open() — Establish a transport endpoint" on page 1899
- "t_optmgmt() — Manage options for a transport endpoint" on page 1901
- "t_rcvconnect() — Receive the confirmation from a connect request" on page 1910

takesocket() — Acquire a socket from another program

Standards

Standards / Extensions	C or C++	Dependencies
z/OS UNIX	both	

Format

```
#define _OPEN_SYS_SOCK_EXT
#include <sys/types.h>
#include <socket.h>

int takesocket(struct clientid *clientid, int sdesc);
```

General description

The takesocket() function acquires a socket from another program. Typically, the other program passes its client ID and socket descriptor, and/or process id (PID), to your program through your program's startup parameter list.

Parameter
Description

clientid A pointer to the *clientid* of the application from which you are taking a socket.

sdesc The descriptor of the socket to be taken.

If your program is using the PID to ensure integrity between givesocket() and takesocket(), before issuing the takesocket() call, your program should set the *c_pid.pid* field of the clientid structure to the PID of the giving program (that is, program that issued the givesocket() call). This identifies the process from which the socket is to be taken. If the *c_reserved.type* field of the clientid structure was set to SO_CLOSE on the givesocket() call, *c_close.SockToken* of clientid structure should be used as input to takesocket() instead of the normal socket descriptor. See "givesocket() — Make the specified socket available" on page 804 for a description of the clientid structure.

Returned value

If successful, takesocket() returns the new socket descriptor.

If unsuccessful, takesocket() returns -1 and sets errno to one of the following values:

Error Code
Description

EACCES
The other application did not give the socket to your application.

EBADF
The *sdesc* parameter does not specify a valid socket descriptor owned by the other application, or the socket has already been taken.

EFAULT
Using the *clientid* parameter as specified would result in an attempt to access storage outside the caller's address space.

EINVAL
The *clientid* parameter does not specify a valid client identifier. Either the client process cannot be found, or the client exists, but has no outstanding givesockets.

EMFILE
The socket descriptor table is already full.

Related information

- "sys/socket.h" on page 74
- "sys/types.h" on page 75
- "getclientid() — Get the identifier for the calling application" on page 691
- "__getclientid() — Get the PID identifier for the calling application" on page 692
- "givesocket() — Make the specified socket available" on page 804

t_alloc() — Allocate a library structure

Standards

Standards / Extensions	C or C++	Dependencies
XPG4.2	both	

Format

```
#define _XOPEN_SOURCE_EXTENDED 1
#include <xti.h>

char *t_alloc(int fd, int struct_type, int fields);
```

General description

Dynamically allocates memory for the various transport function argument structures as specified below. t_alloc() allocates memory for the specified structure, and memory for buffers referenced by the structure.

The structure to allocate is specified by *struct_type* and must be one of the following:

```
        T_BIND      struct t_bind
        T_CALL      struct t_call
        T_OPTMGMT   struct t_optmgmt
        T_DIS       struct t_discon
        T_UNITDATA  struct t_unitdata
        T_UDERROR   struct t_uderr
        T_INFO      struct t_info
```

where each of these structures may subsequently be used as an argument to one or more transport functions.

Each of the above structures, except T_INFO, contains at least one field of type struct netbuf. For each field of this type, the user may specify that the buffer for that field should be allocated as well. The length of the buffer allocated will be equal to or greater than the appropriate size as returned in the *info* argument of t_open() or t_getinfo() . The relevant fields of the *info* argument are described in the following list. The *fields* argument specifies which buffers to allocate, where the argument is the bitwise OR of any of the following:

T_ADDR
The *addr* field of the t_bind, t_call, t_unitdata or t_uderr structures.

T_OPT
The *opt* field of the t_optmgmt, t_call, t_unitdata or t_uderr structures.

T_UDATA
The *udata* field of the t_call, t_discon or t_unitdata structures.

T_ALL
All relevant fields of the given structure. Fields which are not supported by the transport provider specified by fd will not be allocated.

For each relevant field specified in *fields*, t_alloc() allocates memory for the buffer associated with the field, and initializes the *len* field to zero and the *buf* pointer and *maxlen* field accordingly. Irrelevant or unknown values passed in fields are ignored. Since the length of the buffer allocated will be based on the same size information that is returned to the user on a call to t_open() and t_getinfo(), *fd* must refer to the transport endpoint through which the newly allocated structure will be passed. In this way the appropriate size information can be accessed. If the size value associated with any specified field is -1 or -2 (see t_open() or t_getinfo()), t_alloc() will be unable to determine the size of the buffer to allocate and will fail, setting *t_errno* to `TSYSERR` and errno to `EINVAL`. For any field not specified in *fields*, *buf* will be set to the NULL pointer and *len* and *maxlen* will be set to zero.

Use of t_alloc() to allocate structures helps ensure the compatibility of user programs with future releases of the transport interface functions.

Valid states: All - except for T_UNINIT

Returned value

If successful, t_alloc() returns a pointer to the newly allocated structure.

If unsuccessful, t_alloc() returns a NULL pointer and sets errno to one of the following values:

Error Code
Description

TBADF
The specified file descriptor does not refer to a transport endpoint.

TNOSTRUCTYPE
Unsupported struct_type requested. This can include a request for a structure type which is inconsistent with the transport provider type specified, that is, connection-oriented or connectionless.

TPROTO
This error indicates that a communication problem has been detected between XTI and the transport provider for which there is no other suitable XTI (t_errno).

TSYSERR
A system error has occurred during execution of this function.

Related information

- “xti.h” on page 87
- “t_free() — Free a library structure” on page 1871
- “t_getinfo() — Get protocol-specific service information” on page 1874
- “t_open() — Establish a transport endpoint” on page 1899

tan(), tanf(), tanl() — Calculate tangent

Standards

Standards / Extensions	C or C++	Dependencies
ISO C POSIX.1 XPG4 XPG4.2 ISO/ANSI C++ C99 Single UNIX Specification, Version 3 C++ TR1 C99	both	

Format

```
#include <math.h>

double tan(double x);
float tan(float x);                 /* C++ only */
long double tan(long double x);     /* C++ only */
float tanf(float x);
long double tanl(long double x);
```

General description

Calculates the tangent of *x*, where *x* is expressed in radians. If *x* is large, a partial loss of significance in the result can occur.

Note: These functions work in both IEEE Binary Floating-Point and hexadecimal floating-point formats. See "IEEE binary floating-point" on page 94 for more information about IEEE Binary Floating-Point.

Returned value

Returns the calculated tangent of *x*.

If the correct value would cause an underflow, 0 is returned. If the result overflows, ±HUGE_VAL is returned. For both an underflow and an overflow, the value ERANGE is stored in errno.

Special behavior for XPG4.2: The following error is added:

Error Code
Description

EDOM
The argument exceeded an internal limit for the function (approximately 2^{50}).

Example

CELEBT01

```
/* CELEBT01

   This example computes x as the tangent of PI/4.

 */
#include <math.h>
#include <stdio.h>

int main(void)
{
   double pi, x;

   pi = 3.1415926;
   x = tan(pi/4.0);

   printf("tan( %lf ) is %lf\n", pi/4, x);
}
```

Output

```
tan( 0.785398 ) is 1.000000
```

Related information

- “math.h” on page 44
- “acos(), acosf(), acosl() — Calculate arccosine” on page 135
- “acosh(), acoshf(), acoshl() — Calculate hyperbolic arccosine” on page 139
- “asin(), asinf(), asinl() — Calculate arcsine” on page 185
- “asinh(), asinhf(), asinhl() — Calculate hyperbolic arcsine” on page 188
- “atan(), atanf(), atanl(), atan2(), atan2f(), atan2l() — Calculate arctangent” on page 192
- “atanh(), atanhf(), atanhl() — Calculate hyperbolic arctangent” on page 195
- “cos(), cosf(), cosl() — Calculate cosine” on page 328
- “cosh(), coshf(), coshl() — Calculate hyperbolic cosine” on page 331
- “sin(), sinf(), sinl() — Calculate sine” on page 1672
- “sinh(), sinhf(), sinhl() — Calculate hyperbolic sine” on page 1674
- “tanh(), tanhf(), tanhl() — Calculate hyperbolic tangent” on page 1818

tand32(), tand64(), tand128() - Calculate tangent

Standards

Standards / Extensions	C or C++	Dependencies
C/C++ DFP	both	z/OS V1.10

Format

```
#define __STDC_WANT_DEC_FP__
#include <math.h>

_Decimal32  tand32(_Decimal32 x);
_Decimal64  tand64(_Decimal64 x);
_Decimal128 tand128(_Decimal128 x);
_Decimal32  tan(_Decimal32 x);     /* C++ only */
_Decimal64  tan(_Decimal64 x);     /* C++ only */
_Decimal128 tan(_Decimal128 x);    /* C++ only */
```

General description

Calculates the tangent of *x*, where *x* is expressed in radians. If x is large, a partial loss of significance in the result can occur.

These functions work in IEEE decimal floating-point format. See “IEEE decimal floating-point” on page 95 for more information.

Note: To use IEEE decimal floating-point, the hardware must have the Decimal Floating-Point Facility installed.

Returned value

Returns the calculated tangent of *x*.

If the correct value would cause underflow, zero is returned. If the result overflows, ±HUGE_VAL_D32, ±HUGE_VAL_D64, or ±HUGE_VAL_D128 is returned. For both underflow and overflow, the value ERANGE is stored in errno.

Example

CELEBT22

```
/* CELEBT22

   This example illustrates the tand64() function.

   This example computes x as the tangent of PI/4.

*/

#define __STDC_WANT_DEC_FP__
#include <math.h>
#include <stdio.h>

int main(void)
{
   _Decimal64 pi, x;

   pi = 3.1415926DD;
   x = tand64(pi/4.0DD);

   printf("tand64( %Df ) is %Df\n", pi/4.0DD, x);
}
```

Related information

- “math.h” on page 44
- “atanh(), atanhf(), atanhl() — Calculate hyperbolic arctangent” on page 195
- “acosd32(), acosd64(), acosd128() - Calculate arccosine” on page 137
- “acoshd32(), acoshd64(), acoshd128() - Calculate hyperbolic arccosine” on page 140
- “asind32(), asind64(), asind128() - Calculate arcsine” on page 187
- “asinhd32(), asinhd64(), asinhd128() - Calculate hyperbolic arcsine” on page 189
- “atand32(), atand64(), atand128(), atan2d32(), atan2d64(), atan2d128() - Calculate arctangent” on page 194
- “atanhd32(), atanhd64(), atanhd128() - Calculate hyperbolic arctangent” on page 197
- “cosd32(), cosd64(), cosd128() — Calculate cosine” on page 329
- “coshd32(), coshd64(), coshd128() - Calculate hyperbolic cosine” on page 332
- “sind32(), sind64(), sind128() — Calculate sine” on page 1673
- “sinhd32(), sinhd64(), sinhd128() - Calculate hyperbolic sine” on page 1676
- “tanhd32(), tanhd64(), tanhd128() - Calculate hyperbolic tangent” on page 1820

tanh(), tanhf(), tanhl() — Calculate hyperbolic tangent

Standards

Standards / Extensions	C or C++	Dependencies
ISO C POSIX.1 XPG4 XPG4.2 ISO/ANSI C++ C99 Single UNIX Specification, Version 3 C++ TR1 C99	both	

Format

```
#include <math.h>

double tanh(double x);
```

```
float tanh(float x);                /* C++ only */
long double tanh(long double x);    /* C++ only */
float tanhf(float x);
long double tanhl(long double x);
```

General description

Calculates the hyperbolic tangent of *x*, where *x* is expressed in radians. The result of the function cannot have a range error.

Note: These functions work in both IEEE Binary Floating-Point and hexadecimal floating-point formats. See "IEEE binary floating-point" on page 94 for more information about IEEE Binary Floating-Point.

Returned value

Returns the calculated value of the hyperbolic tangent of *x*.

If the result underflows, the function returns 0 and sets the errno to ERANGE.

Example

CELEBT02

```
/* CELEBT02

   This example computes x as the hyperbolic tangent of PI/4.

 */
#define _POSIX_SOURCE
#include <math.h>
#include <stdio.h>

int main(void)
{
   double pi, x;

   pi = 3.1415926;
   x = tanh(pi/4);

   printf("tanh( %lf ) = %lf\n", pi/4, x);
}
```

Output

```
tanh( 0.785398 ) = 0.655794
```

Related information

- "math.h" on page 44
- "acos(), acosf(), acosl() — Calculate arccosine" on page 135
- "acosh(), acoshf(), acoshl() — Calculate hyperbolic arccosine" on page 139
- "asin(), asinf(), asinl() — Calculate arcsine" on page 185
- "asinh(), asinhf(), asinhl() — Calculate hyperbolic arcsine" on page 188
- "atan(), atanf(), atanl(), atan2(), atan2f(), atan2l() — Calculate arctangent" on page 192
- "atanh(), atanhf(), atanhl() — Calculate hyperbolic arctangent" on page 195
- "cos(), cosf(), cosl() — Calculate cosine" on page 328
- "cosh(), coshf(), coshl() — Calculate hyperbolic cosine" on page 331
- "sin(), sinf(), sinl() — Calculate sine" on page 1672
- "sinh(), sinhf(), sinhl() — Calculate hyperbolic sine" on page 1674
- "tan(), tanf(), tanl() — Calculate tangent" on page 1815

tanhd32(), tanhd64(), tanhd128() - Calculate hyperbolic tangent

Standards

Standards / Extensions	C or C++	Dependencies
C/C++ DFP	both	z/OS V1.10

Format

```
#define __STDC_WANT_DEC_FP__
#include <math.h>

_Decimal32  tanhd32(_Decimal32 x);
_Decimal64  tanhd64(_Decimal64 x);
_Decimal128 tanhd128(_Decimal128 x);
_Decimal32  tanh(_Decimal32 x);     /* C++ only */
_Decimal64  tanh(_Decimal64 x);     /* C++ only */
_Decimal128 tanh(_Decimal128 x);    /* C++ only */
```

General description

Calculates the hyperbolic tangent of *x*, where *x* is expressed in radians.

Returned value

Returns the calculated value of the hyperbolic tangent of *x*.

If the result underflows, the function returns 0 and sets the errno to ERANGE.

Example

CELEBT23

```
/* CELEBT23

   This example illustrates the tanhd64() function.

   This example computes x as the hyperbolic tangent of PI/4.

*/

#define __STDC_WANT_DEC_FP__
#include <math.h>
#include <stdio.h>

int main(void)
{
   _Decimal64 pi, x;

   pi = 3.1415926DD;
   x = tanhd64(pi/4.0DD);

   printf("tanhd64( %Df ) = %Df\n", pi/4.0DD, x);
}
```

Related information

- “math.h” on page 44
- “atanh(), atanhf(), atanhl() — Calculate hyperbolic arctangent” on page 195
- “acosd32(), acosd64(), acosd128() - Calculate arccosine” on page 137
- “acoshd32(), acoshd64(), acoshd128() - Calculate hyperbolic arccosine” on page 140

- "asind32(), asind64(), asind128() - Calculate arcsine" on page 187
- "asinhd32(), asinhd64(), asinhd128() - Calculate hyperbolic arcsine" on page 189
- "atand32(), atand64(), atand128(), atan2d32(), atan2d64(), atan2d128() - Calculate arctangent" on page 194
- "atanhd32(), atanhd64(), atanhd128() - Calculate hyperbolic arctangent" on page 197
- "cosd32(), cosd64(), cosd128() — Calculate cosine" on page 329
- "coshd32(), coshd64(), coshd128() - Calculate hyperbolic cosine" on page 332
- "sind32(), sind64(), sind128() — Calculate sine" on page 1673
- "sinhd32(), sinhd64(), sinhd128() - Calculate hyperbolic sine" on page 1676
- "tand32(), tand64(), tand128() - Calculate tangent" on page 1817

t_bind() — Bind an address to a transport endpoint

Standards

Standards / Extensions	C or C++	Dependencies
XPG4.2	both	

Format

```
#define _XOPEN_SOURCE_EXTENDED 1
#include <xti.h>

int t_bind(int fd, struct t_bind *req, struct t_bind *ret);
```

General description

Associates a protocol address with the transport endpoint specified by *fd* and activates that transport endpoint. In connection mode, the transport provider may begin enqueuing incoming connect indications, or servicing a connection request on the transport endpoint. In connectionless mode, the transport user may send or receive data units through the transport endpoint.

The *req* and *ret* arguments point to a t_bind structure containing the following members:

```
        struct netbuf   addr;
        unsigned        qlen;
```

The *addr* field of the t_bind structure specifies a protocol address, and the *qlen* field is used to indicate the maximum number of outstanding connect indications.

The parameter *req* is used to request that an address, represented by the netbuf structure, be bound to the given transport endpoint. The parameter *len* specifies the number of bytes in the address, and *buf* points to the address buffer. The parameter *maxlen* has no meaning for the *req* argument. On return, *ret* contains the address that the transport provider actually bound to the transport endpoint. This is the same as the address specified by the user in *req*. In *ret*, the user specifies *maxlen*, which is the maximum size of the address buffer, and *buf* which points to the buffer where the address is to be placed. On return, *len* specifies the number of bytes in the bound address, and *buf* points to the bound address. If *maxlen* is not large enough to hold the returned address, an error results.

If the requested address is not available, t_bind() returns -1 with *t_errno* set as appropriate. If no address is specified in *req* (the *len* field of *addr* in *req* is zero or *req* is NULL), the transport provider will assign an appropriate address to be bound, and will return that address in the *addr* field of *ret*. If the transport provider could not allocate an address, t_bind() fails with t_errno set to TNOADDR.

The parameter *req* may be a NULL pointer if the user does not wish to specify an address to be bound. Here, the value of *qlen* is assumed to be zero, and the transport provider assigns an address to the transport endpoint. Similarly, *ret* may be a NULL pointer if the user does not care what address was bound by the provider and is not interested in the negotiated value of *qlen*. It is valid to set *req* and *ret* to the NULL pointer for the same call, in which case the provider chooses the address to bind to the transport endpoint and does not return that information to the user.

The *qlen* field specifies the number of outstanding connect indications that the transport provider should support for the given transport endpoint. An outstanding connect indication is one that has been passed to the transport user by the transport provider, but which has not been accepted or rejected. A value of *qlen* greater than 0 is only meaningful when issued by a passive transport user that expects other users to call it. The value of *qlen* will be negotiated by the transport provider and will always be negotiated to 1 (one) from any nonzero value. On return, the *qlen* field in *ret* will contain the negotiated value.

If *fd* refers to a connection-oriented service, then multiple endpoints my be bound to the same protocol address by way of connections accepted on an endpoint using t_accept. The TCP transport provider will not permit the user to explicitly bind multiple endpoints to the same address. It is also not possible to bind an endpoint to more than one protocol address. If a user attempts to explicitly bind multiple endpoints to a protocol address, the second and subsequent binds will fail with t_errno set to TADDRBUSY. When a user accepts a connection on the transport endpoint that is being used as the listening endpoint, the bound protocol address will be found to be busy for the duration of the connection, until a t_unbind() or t_close() call has been issued. No other transport endpoints may be bound for listening on that same protocol address while that initial listening endpoint is active (in the data transfer phase or in the T_IDLE state). This prevents more than one transport endpoint bound to the same protocol address from accepting connect indications.

Valid states: T_UNBND

Returned value

If successful, t_bind() returns 0.

If unsuccessful, t_bind() returns -1 and sets errno to one of the following values:

Error Code
  Description

TACCES
  The user does not have permission to use the specified address.

TADDRBUSY
  The requested address is in use.

TBADADDR
  The specified protocol address was in an incorrect format or contained illegal information.

TBADF
The specified file descriptor does not refer to a transport endpoint.

TBUFOVFLW
The number of bytes allowed for an incoming argument (*maxlen*) is greater than 0 but not sufficient to store the value of that argument. The provider's state will change to T_IDLE and the information to be returned in *ret* will be discarded.

TNOADDR
The transport provider could not allocate an address.

TOUTSTATE
The function was issued in the wrong sequence.

TPROTO
This error indicates that a communication problem has been detected between XTI and the transport provider for which there is no other suitable XTI (*t_errno*).

TSYSERR
A system error has occurred during execution of this function.

Related information

- "xti.h" on page 87
- "t_alloc() — Allocate a library structure" on page 1814
- "t_close() — Close a transport endpoint" on page 1837
- "t_open() — Establish a transport endpoint" on page 1899
- "t_optmgmt() — Manage options for a transport endpoint" on page 1901
- "t_unbind() — Disable a transport endpoint" on page 1934

tcdrain() — Wait until output has been transmitted

Standards

Standards / Extensions	C or C++	Dependencies
POSIX.1 XPG4 XPG4.2 Single UNIX Specification, Version 3	both	

Format

```
#define _POSIX_SOURCE
#include <termios.h>

int tcdrain(int fildes);
```

General description

The tcdrain() function waits until all output sent to *fildes* has actually been sent to the terminal device.

If tcdrain() is called from a background process group against the caller's controlling terminal, a SIGTTOU signal may be generated depending how the process is handling SIGTTOUs:

Processing for SIGTTOU	System Behavior
Default or signal handler	The SIGTTOU signal is generated, and the function is not performed. tcdrain() returns -1 and sets errno to EINTR.
Ignored or blocked	The SIGTTOU signal is not sent, and the function continues normally.

Returned value

If successful, tcdrain() returns 0.

If unsuccessful, tcdrain() returns -1 and sets errno to one of the following values:

Error Code
Description

EBADF
fildes is not a valid open file descriptor.

EINTR
A signal interrupted tcdrain().

EIO The process group of the process issuing the function is an orphaned, background process group, and the process issuing the function is not ignoring or blocking SIGTTOU.

ENOTTY
fildes is not associated with a terminal.

Example

CELEBT03

```
/* CELEBT03  */

#define _POSIX_SOURCE
#include <termios.h>
#include <stdio.h>
#include <time.h>
#include <fcntl.h>
#include <unistd.h>
#include <sys/stat.h>

main() {
  char Master[]="master.tty";
  char Slave[]="slave.tty";
  char text[]="text to be written to tty";
  char data[80];
  int  master, slave;
  time_t T;

  if (mknod(Master, S_IFCHR|S_IRUSR|S_IWUSR, 0x00010000 + 10) !=0)
    perror("mknod() error for master tty");
  else {
    if (mknod(Slave, S_IFCHR|S_IRUSR|S_IWUSR, 0x00020000 + 10) !=0)
      perror("mknod() error for slave tty");
    else {
      if ((master = open(Master, O_RDWR|O_NONBLOCK)) < 0)
        perror("open() error for master tty");
      else {
        if ((slave = open(Slave, O_RDWR|O_NONBLOCK)) < 0)
          perror("open() error for slave tty");
        else {
          if (fork() == 0) {
            if (write(slave, text, strlen(text)+1) == -1)
              perror("write() error");
            time(&T);
            printf("child has written to tty, tcdrain() started at %s",
                   ctime(&T));
            if (tcdrain(slave) != 0)
              perror("tcdrain() error");
            time(&T);
            printf("tcdrain() returned at %s", ctime(&T));
            exit(0);
          }
          time(&T);
          printf("parent is starting nap at %s", ctime(&T));
          sleep(5);
          time(&T);
          printf("parent is done with nap at %s", ctime(&T));
          if (read(master, data, sizeof(data)) == -1)
            perror("read() error");
          else printf("read '%s' from the tty\n", data);
          sleep(5);
          close(slave);
        }
        close(master);
      }
      unlink(Slave);
    }
    unlink(Master);
  }
}
```

Output

```
parent is starting nap at Fri Jun 16 12:46:28 2006
child has written to tty, tcdrain() started at Fri Jun 16 12:46:28 2006
parent is done with nap at Fri Jun 16 12:46:34 2006
read 'text to be written to tty' from the tty
tcdrain() returned at Fri Jun 16 12:46:34 2006
```

Related information

- “termios.h” on page 78
- “tcflow() — Suspend or resume data flow on a terminal”
- “tcflush() — Flush input or output on a terminal” on page 1828
- “tcgetattr() — Get the attributes for a terminal” on page 1830
- “tcgetpgrp() — Get the foreground process group ID” on page 1835
- “tcsetattr() — Set the attributes for a terminal” on page 1843
- “tcsendbreak() — Send a break condition to a terminal” on page 1842
- “tcsetpgrp() — Set the foreground process group ID” on page 1858

tcflow() — Suspend or resume data flow on a terminal

Standards

Standards / Extensions	C or C++	Dependencies
POSIX.1 XPG4 XPG4.2 Single UNIX Specification, Version 3	both	

Format

```
#define _POSIX_SOURCE
#include <termios.h>

int tcflow(int fildes, int action);
```

General description

Suspends or resumes transmission or reception of data on a terminal device.

`int` *fildes*
A file descriptor associated with a terminal device.

`int` *action*
Indicates the action you want to perform, represented by one of the following symbols defined in the termios.h include file:

Symbol
Meaning

TCOOFF
Suspends output.

TCOON
Resumes suspended output.

TCIOFF
Sends a STOP character to the terminal, to stop the terminal from sending any further input.

TCION
Sends a START character to the terminal, to tell the terminal that it can resume sending input.

If tcflow() is called from a background process group against the caller's controlling terminal, a SIGTTOU signal may be generated depending how the process is handling SIGTTOUs:

Processing for SIGTTOU	System Behavior
Default or signal handler	The SIGTTOU signal is generated, and the function is not performed. tcflow() returns -1 and sets errno to EINTR.
Ignored or blocked	The SIGTTOU signal is not sent, and the function continues normally.

Returned value

If successful, tcflow() returns 0.

If unsuccessful, tcflow() returns -1 and sets errno to one of the following values:

Error Code
Description

EBADF
fildes is not a valid open file descriptor.

EINTR
A signal interrupted tcflow().

EINVAL
action had an incorrect value.

EIO For either of the following reasons:

- TCIOFF or TCION was requested, but the other side of the pseudoterminal connection is closed.
- The process group of the process issuing the function is an orphaned, background process group, and the process issuing the function is not ignoring or blocking SIGTTOU.

ENOTTY
fildes is not associated with a terminal.

Example

CELEBT04

```
/* CELEBT04

    This example suspends and then resumes transmission.

 */
 #define _POSIX_SOURCE
 #include <termios.h>
 #include <stdio.h>
 #include <fcntl.h>
 #include <unistd.h>
 #include <sys/stat.h>

 main() {
   char Master[]="/dev/ptyp0010";
   char Slave[]="/dev/ttyp0010";
   char text[]="tesxt to be written to tty";
   char data[80];
   int master, slave;

   if ((master = open(Master, O_RDWR|O_NONBLOCK)) < 0) {
     perror("open() error for master tty");
     exit(1);
   }
   if ((slave = open(Slave, O_RDWR|O_NONBLOCK)) < 0) {
     perror("open() error for slave tty");
     exit(1);

   if (write(slave, text, strlen(text)+1) == -1) {
     perror("write() error");
     exit(1);
   }
   puts("output is suspended to tty");
   if (read(master, data, sizeof(data)) == -1)
     perror("read() error");
   else printf("read '%s' from the tty\n", data);
   if (tcflow(slave, TCOON) != 0)
     perror("tcflow() error");
   exit(1);
   }
   puts("output is resumed to tty");
   if (read(master, data, sizeof(data)) == -1) {
     perror("read() error");
     exit(1);
   }
   printf("read '%s' from the tty\n", data);
   close(slave);
   close(master);
 }
```

Output

```
output is suspended to tty
read() error: Resource temporarily unavailable
output is resumed to tty
read 'text to be written to tty' from the tty
```

Related information

- “termios.h” on page 78
- “tcdrain() — Wait until output has been transmitted” on page 1823
- “tcflush() — Flush input or output on a terminal”
- “tcgetattr() — Get the attributes for a terminal” on page 1830
- “tcgetpgrp() — Get the foreground process group ID” on page 1835
- “tcsendbreak() — Send a break condition to a terminal” on page 1842
- “tcsetattr() — Set the attributes for a terminal” on page 1843
- “tcsetpgrp() — Set the foreground process group ID” on page 1858

tcflush() — Flush input or output on a terminal

Standards

Standards / Extensions	C or C++	Dependencies
POSIX.1 XPG4 XPG4.2 Single UNIX Specification, Version 3	both	

Format

```
#define _POSIX_SOURCE
#include <termios.h>

int tcflush(int fildes, int where);
```

General description

Flushes input or output on a terminal.

int *fildes*;
Indicates a file descriptor associated with a terminal device.

int *where*;
Indicates whether the system is to flush input or output, represented by one of the following symbols defined in the termios.h header file.

Symbol
Meaning

TCIFLUSH
Flushes input data that has been received by the system but not read by an application.

TCOFLUSH
Flushes output data that has been written by an application but not sent to the terminal.

TCIOFLUSH
Flushes both input and output data.

If tcflush() is called from a background process group against the caller's controlling terminal, a SIGTTOU signal may be generated depending how the process is handling SIGTTOUs:

Processing for SIGTTOU	System Behavior
Default or signal handler	The SIGTTOU signal is generated, and the function is not performed. tcflush() returns -1 and sets errno to EINTR.
Ignored or blocked	The SIGTTOU signal is not sent, and the function continues normally.

Returned value

If successful, tcflush() returns 0.

If unsuccessful, tcflush() returns -1 and sets errno to one of the following values:

Error Code
 Description

EBADF
 fildes is not a valid open file descriptor.

EINTR
 A signal interrupted tcflush().

EINVAL
 where has an incorrect value.

EIO The process group of the process issuing the function is an orphaned, background process group, and the process issuing the function is not ignoring or blocking SIGTTOU.

ENOTTY
 fildes is not associated with a terminal.

Example

CELEBT05

```
/* CELEBT05

   This example flushes a string.

 */
#define _POSIX_SOURCE
#include <termios.h>
#include <stdio.h>
#include <fcntl.h>
#include <unistd.h>
#include <sys/stat.h>

main() {
  char Master[]="master.tty";
  char Slave[]="slave.tty";
  char text1[]="string that will be flushed from buffer";
  char text2[]="string that will not be flushed from buffer";
  char data[80];
  int  master, slave;

  if (mknod(Master, S_IFCHR|S_IRUSR|S_IWUSR, 0x00010000 + 10) != 0)
    perror("mknod() error for master tty");
  else {
    if (mknod(Slave, S_IFCHR|S_IRUSR|S_IWUSR, 0x00020000 + 10) != 0)
      perror("mknod() error for slave tty");
    else {
      if ((master = open(Master, O_RDWR|O_NONBLOCK)) < 0)
        perror("open() error for master tty");
      else {
        if ((slave = open(Slave, O_RDWR|O_NONBLOCK)) < 0)
          perror("open() error for slave tty");
        else {
          if (write(slave, text1, strlen(text1)+1) == -1)
            perror("write() error");
          else if (tcflush(slave, TCOFLUSH) != 0)
            perror("tcflush() error");
          else {
            puts("first string is written and tty flushed");
            puts("now writing string that will not be flushed");
            if (write(slave, text2, strlen(text2)+1) == -1)
              perror("write() error");
            else if (read(master, data, sizeof(data)) == -1)
              perror("read() error");
            else printf("read '%s' from the tty\n", data);
          }
          close(slave);
        }
        close(master);
      }
      unlink(Slave);
    }
    unlink(Master);
  }
}
```

Output

```
first string is written and tty flushed
now writing string that will not be flushed
read 'string that will not be flushed from buffer' from the tty
```

Related information

- “termios.h” on page 78
- “tcdrain() — Wait until output has been transmitted” on page 1823
- “tcflow() — Suspend or resume data flow on a terminal” on page 1825
- “tcgetattr() — Get the attributes for a terminal”
- “tcgetpgrp() — Get the foreground process group ID” on page 1835
- “tcsendbreak() — Send a break condition to a terminal” on page 1842
- “tcsetattr() — Set the attributes for a terminal” on page 1843
- “tcsetpgrp() — Set the foreground process group ID” on page 1858

tcgetattr() — Get the attributes for a terminal

Standards

Standards / Extensions	C or C++	Dependencies
POSIX.1 XPG4 XPG4.2 Single UNIX Specification, Version 3	both	

Format

```
#define _POSIX_SOURCE
#include <termios.h>

int tcgetattr(int fildes, struct termios *termptr);
```

General description

Gets a `termios` structure, which contains control information for a terminal associated with *fildes*. It stores that information in a memory location that *termptr* points to. The contents of a `termios` structure are described in “tcsetattr() — Set the attributes for a terminal” on page 1843.

tcgetattr() can run in either a foreground or background process; however, if the process is in the background, a foreground process may subsequently change the attributes.

tcgetattr() only works in an environment where either a controlling terminal exists, or stdin and stderr refer to tty devices. Specifically, it does not work in a TSO environment.

Note: The tcgetattr() function has a dependency on the level of the Enhanced ASCII Extensions. See “Enhanced ASCII support” on page 2113 for details.

Returned value

If successful, tcgetattr() returns 0.

If unsuccessful, tcgetattr() returns -1 and sets errno to one of the following values:

Error Code
Description

EBADF
fildes is not a valid open file descriptor.

ENOTTY
The file associated with *fildes* is not a terminal.

Example

CELEBT06

```
/* CELEBT06

   This example provides information about the attributes.

 */
#define _POSIX_SOURCE
#include <termios.h>
#include <stdio.h>
#include <unistd.h>

main() {
  struct termios term;

  if (tcgetattr(STDIN_FILENO, &term) != 0)
    perror("tcgetatt() error");
  else {
    if (term.c_iflag & BRKINT)
      puts("BRKINT is set");
    else
```

```
      puts("BRKINT is not set");
    if (term.c_cflag & PARODD)
      puts("Odd parity is used");
    else
      puts("Even parity is used");
    if (term.c_lflag & ECHO)
      puts("ECHO is set");
    else
      puts("ECHO is not set");
    printf("The end-of-file character is x'%02x'\n",
term.c_cc[VEOF]);
  }
}
```

Output

```
ECHO is set
The End Of File character is x'37'
```

Related information

- “termios.h” on page 78
- “cfgetispeed() — Determine the input baud rate” on page 254
- “cfgetospeed() — Determine the output baud rate” on page 256
- “cfsetispeed() — Set the input baud rate in the termios” on page 258
- “cfsetospeed() — Set the output baud rate in the termios” on page 260
- “tcdrain() — Wait until output has been transmitted” on page 1823
- “tcflow() — Suspend or resume data flow on a terminal” on page 1825
- “tcflush() — Flush input or output on a terminal” on page 1828
- “tcgetpgrp() — Get the foreground process group ID” on page 1835
- “tcsendbreak() — Send a break condition to a terminal” on page 1842
- “tcsetattr() — Set the attributes for a terminal” on page 1843
- “tcsetpgrp() — Set the foreground process group ID” on page 1858

__tcgetcp() — Get terminal code page names

Standards

Standards / Extensions	C or C++	Dependencies
z/OS UNIX	both	

Format

```
#define _OPEN_SYS_PTY_EXTENSIONS
#include <termios.h>

int __tcgetcp(int fildes, size_t termcplen, struct __termcp *termcpptr);
```

General description

The __tcgetcp() function gets the terminal session code page information contained in the termcp structure and the Code Page Change Notification (CPCN) capability for the terminal file.

The following arguments are used:

fildes The file descriptor of the terminal for which you want to get the code page names and CPCN capability.

termcplen
The length of the passed termcp structure.

termcpptr
A pointer to a __termcp structure.

__tcgetcp() stores the termcp information in a memory location pointed to by *termcpptr*. The return value contains the CPCN capability. The following CPCN capabilities are defined:

Symbol
Meaning

_CPCN_NAMES
Forward code page names only

Use the __tcsetcp() function to change the terminal session data conversion. The z/OS UNIX pseudotty device driver supports this CPCN capability.

_CPCN_TABLES
Forward code page names and tables

Use __tcsettables() to change the terminal session data conversion. The OCS remote-tty device driver supports this CPCN capability.

In the returned termcp structure, if the _TCCP_FASTP bit is set then the data conversion that is specified by the source and target code page names can be performed locally by the data conversion application. This is valid any time that a table-driven conversion can be performed. For example, the data conversion point (application) could use the z/OS UNIX iconv() service to build local data conversion tables and perform all data conversion using the local tables instead of using iconv() all in subsequent conversions. This provides for better-performing data conversion.

In the returned termcp structure, if the _TCCP_BINARY bit is set then no data conversion is being performed and the code page names contained in the termcp structure should be ignored.

__tcgetcp() can run in either a foreground or background process; however, if the process is in the background, a foreground process may subsequently change the terminal code pages.

Note: The __tcgetcp() function has a dependency on the level of the Enhanced ASCII Extensions. See “Enhanced ASCII support” on page 2113 for details.

Returned value

If successful, __tcgetcp() returns the termcp structure in a memory location pointed to by *termcpptr*. The return value contains the CPCN capability.

If unsuccessful, __tcgetcp() returns -1 and sets errno to one of the following values:

Error Code
Description

EBADF
fildes is not a valid open file descriptor.

EINVAL
The value of *termcplen* was invalid.

ENODEV
One of the following error conditions exist:

- The terminal device driver does not support CPCN functions.
- CPCN functions have not been enabled.

  For a z/OS UNIX pseudotty terminal device file, issue the __tcsetcp() function against the master pty first to enable CPCN support.

ENOTTY
The file associated with *fildes* is not a terminal device.

Example

The following example retrieves the current code pages used in the data conversion and CPCN capability. Here, the __tcgetcp() function is issued against a session using a pty terminal device; ISO8859-1 and IBM-1047 code pages are being used.

```
#define _OPEN_SYS_PTY_EXTENSIONS
#include <unistd.h>
#include <stdio.h>
#include <fcntl.h>
#include <termios.h>

void main(void)
{
  struct __termcp mytermcp;
  int rv;
  int cterm_fd;

  if ((cterm_fd = open("/dev/tty",O_RDWR)) == -1)
    printf("No controlling terminal established.\n");
  else {
    if ((rv = __tcgetcp(cterm_fd,sizeof(mytermcp),&mytermcp))== -1)
      perror("__tcgetcp() error");
    else {
      if (_CPCN_NAMES == rv)
        printf("Forward Code Page Names Only.\n");
      else
        printf("Forward Code Page Names and Tables.\n");
      if (_TCCP_BINARY == (mytermcp.__tccp_flags & _TCCP_BINARY))
        printf("Binary mode is in effect.\n");
      else {
        printf("ASCII code page name is %s.\n",
                mytermcp.__tccp_fromname);
        printf("EBCIDC code page name is %s.\n",
                mytermcp.__tccp_toname);
      }
    }
    close(cterm_fd);
  }
}  /* main */
```

Output

```
Forward code page names only.
ASCII code page name is ISO8859-1.
EBCDIC code page name is IBM-1047.
```

Related information

- “termios.h” on page 78
- “__tcsetcp() — Set terminal code page names” on page 1854
- “__tcsettables() — Set terminal code page names and conversion tables” on page 1860

tcgetpgrp() — Get the foreground process group ID

Standards

Standards / Extensions	C or C++	Dependencies
POSIX.1 XPG4 XPG4.2 Single UNIX Specification, Version 3	both	

Format

```
#define _POSIX_SOURCE
#include <unistd.h>

pid_t tcgetpgrp(int fildes);
```

General description

Gets the process group ID (PGID) of the foreground process group associated with the terminal referred to by *fildes*. tcgetpgrp() can run from a background process, but the information may subsequently be changed by a process in the foreground process group.

Returned value

If successful, tcgetpgrp() returns of the foreground process group's PGID.

If unsuccessful, tcgetpgrp() returns -1 and sets errno to one of the following values:

Error Code
Description

EBADF
fildes is not a valid open file descriptor.

ENOTTY
The calling process does not have a controlling terminal, or the file is not the controlling terminal.

Example

CELEBT07

```
/* CELEBT07

   This example gets the foreground PGID.

 */
#define _POSIX_SOURCE
#include <termios.h>
#include <unistd.h>
#include <sys/wait.h>       /*FIX: was #include <sys/wait.h> */
#include <stdio.h>

main() {
  pid_t pid;

  if ((pid = tcgetpgrp(STDOUT_FILENO)) < 0)
    perror("tcgetpgrp() error");
  else
    printf("the foreground process group id of stdout is %d\n",
           (int) pid);
}
```

Output

```
the foreground process group id of stdout is 4063240
```

Related information

- "unistd.h" on page 82
- "setpgid() — Set process group ID for job control" on page 1565
- "setsid() — Create session, set process group ID" on page 1576
- "tcdrain() — Wait until output has been transmitted" on page 1823
- "tcflow() — Suspend or resume data flow on a terminal" on page 1825
- "tcflush() — Flush input or output on a terminal" on page 1828
- "tcgetattr() — Get the attributes for a terminal" on page 1830
- "tcsendbreak() — Send a break condition to a terminal" on page 1842
- "tcsetattr() — Set the attributes for a terminal" on page 1843
- "tcsetpgrp() — Set the foreground process group ID" on page 1858

tcgetsid() — Get process group ID for session leader for controlling terminal

Standards

Standards / Extensions	C or C++	Dependencies
XPG4.2 Single UNIX Specification, Version 3	both	

Format

```
#define _XOPEN_SOURCE_EXTENDED 1
#include <termios.h>

pid_t tcgetsid(int fildes);
```

General description

The tcgetsid() obtains the process group ID of the session for which the terminal specified by *fildes* is the controlling terminal.

Returned value

If successful, tcgetsid() returns the process group ID associated with the terminal.

If unsuccessful, tcgetsid() returns (pid_t)-1 and sets errno to one of the following values:

Error Code
Description

EACCES
The *fildes* argument is not associated with a controlling terminal. If the environment variable _EDC_SUSV3 is set to 1, ENOTTY will be returned instead of EACCES.

EBADF
The *fildes* argument is not a valid file descriptor.

ENOTTY
The calling process does not have a controlling terminal, or the file is not the controlling terminal.

Note: Starting with z/OS V1.9, environment variable _EDC_SUSV3 can be used to control the behavior of tcgetsid() with respect to setting errno to ENOTTY instead of EACCES. By default, tcgetsid() will set EACESS when fildes is not associated with a controlling terminal. When _EDC_SUSV3 is set to 1, setenv() will set errno to ENOTTY in place of EACCES.

Related information

- "termios.h" on page 78

t_close() — Close a transport endpoint

Standards

Standards / Extensions	C or C++	Dependencies
XPG4.2	both	

Format

```
#define _XOPEN_SOURCE_EXTENDED 1
#include <xti.h>

int t_close(int fd);
```

General description

Informs the transport provider that the user is finished with the transport endpoint specified by *fd*, and frees any local library resources associated with the endpoint. t_close() also closes the file associated with the transport endpoint.

t_close() should be called from the T_UNBND state. However, t_close() does not check state information, so it may be called from any state to close a transport endpoint. If this occurs, the local library resources associated with the endpoint are freed automatically. In addition, close() is issued for that file descriptor. The close() will be abortive if there are no other descriptors in this, or in another process which references the transport endpoint, and in this case will break any transport connection that may be associated with that endpoint.

A t_close() issued on a connection endpoint may cause data previously sent, or data not yet received, to be lost. It is the responsibility of the transport user to ensure that data is received by the remote peer.

Valid states: All - except for T_UNINIT

Returned value

If successful, t_close() returns 0.

If unsuccessful, t_close() returns -1 and sets errno to one of the following values:

Error Code
Description

TBADF
The specified file descriptor does not refer to a transport endpoint.

TPROTO
This error indicates that a communication problem has been detected between XTI and the transport provider for which there is no other suitable XTI (t_errno).

Related information

- "xti.h" on page 87
- "t_getstate() — Get the current state" on page 1876
- "t_open() — Establish a transport endpoint" on page 1899
- "t_unbind() — Disable a transport endpoint" on page 1934

t_connect() — Establish a connection with another transport user

Standards

Standards / Extensions	C or C++	Dependencies
XPG4.2	both	

Format

```
#define _XOPEN_SOURCE_EXTENDED 1
#include <xti.h>

int t_connect(int fd, struct t_call *call, struct t_call *rcvcall);
```

General description

Enables a transport user to request a connection to the specified destination transport user. This function can only be issued in the T_IDLE state. The parameter *fd* identifies the local transport endpoint where communication will be established, while *sndcall* and *rcvcall* point to a t_call structure which contains the following members:

```
struct netbuf   addr;
struct netbuf   opt;
struct netbuf   udata;
int             sequence;
```

The parameter *sndcall* specifies information needed by the transport provider to establish a connection, and *rcvcall* specifies information that is associated with the newly established connection.

In *sndcall*, *addr* specifies the protocol address of the destination transport user. *opt* presents any protocol-specific information that might be needed by the transport provider. *udata* points to optional user data that may be passed to the destination transport user during connection establishment. *sequence* has no meaning for this function.

On return, in *rcvcall*, *addr* contains the protocol address associated with the responding transport endpoint. *opt* represents any protocol-specific information associated with the connection. *udata* points to optional user data that may be returned by the destination transport user during connection establishment. *sequence* has no meaning for this function.

The *opt* argument permits users to define the options that may be passed to the transport provider. See the discussion of supported options in t_optmgmt(). The user may choose not to negotiate protocol options by setting the *len* field of *opt* to zero. In this case, the provider may use default options.

If used, *sndcall->opt.buf* must point to a buffer with the corresponding options. The *maxlen* and *buf* fields of the netbuf structure pointed by *rcvcall->addr* and *rcvcall->opt* must be set before the call.

Since passing of userdata over a connection request is not supported under TCP, the *udata* argument is always meaningless.

On return, the *addr*, *opt* and *udata* fields of *rcvcall* will be updated to reflect values associated with the connection. Thus, the *maxlen* field of each argument must be set before issuing this function to indicate the maximum size of the buffer for each. However, *rcvcall* may be a NULL pointer, in which case no information is given to the user on return from t_connect().

By default, t_connect() executes in synchronous mode, and will wait for the destination user's response before returning control to the local user. A successful return (that is, return value of zero) indicates that the requested connection has been established. However, if O_NONBLOCK is set (using t_open() or fcntl()), t_connect() executes in asynchronous mode. In this case, the call will not wait for the remote user's response, but will return control immediately to the local user and return -1 with *t_errno* set to `TNODATA` to indicate that the connection has not yet been established. In this way, the function simply initiates the connection establishment procedure by sending a connect request to the destination transport user. The t_rcvconnect() function is used in conjunction with t_connect() to determine the status of the requested connection.

When a synchronous t_connect() call is interrupted by the arrival of a signal, the state of the corresponding transport endpoint is T_OUTCON, allowing a further call to either t_rcvconnect(), t_rcvdis() or t_snddis().

Valid states: T_IDLE

Returned value

If successful, t_connect() returns 0.

If unsuccessful, t_connect() returns -1 and sets errno to one of the following values:

Error Code
  Description

TACCES
  The user does not have permission to use the specified address or options.

TADDRBUSY
  This transport provider does not support multiple connections with the same local and remote addresses. This error indicates that a connection already exists.

TBADADDR
  The specified protocol address was in an incorrect format or contained illegal information.

TBADDATA
The amount of user data specified was not within the bounds allowed by the transport provider.

TBADF
The specified file descriptor does not refer to a transport endpoint.

TBADOPT
The specified protocol options were in an incorrect format or contained illegal information.

TBUFOVFLW
The number of bytes allocated for an incoming argument (*maxlen*) is greater than 0 but not sufficient to store the value of that argument. If executed in synchronous mode, the provider's state, as seen by the user, changes to T_DATAXFER, and the information to be returned in *rcvcall* is discarded.

TLOOK
An asynchronous event has occurred on this transport endpoint and requires immediate attention.

TNODATA
O_NONBLOCK was set, so the function successfully initiated the connection establishment procedure, but did not wait for a response from the remote user.

TNOTSUPPORT
This function is not supported by the underlying transport provider.

TOUTSTATE
The function was issued in the wrong sequence.

TPROTO
This error indicates that a communication problem has been detected between XTI and the transport provider for which there is no other suitable XTI (*t_errno*).

TSYSERR
A system error has occurred during execution of this function.

Related information

- "xti.h" on page 87
- "t_accept() — Accept a connect request" on page 1810
- "t_alloc() — Allocate a library structure" on page 1814
- "t_getinfo() — Get protocol-specific service information" on page 1874
- "t_listen() — Listen for a connect indication" on page 1883
- "t_open() — Establish a transport endpoint" on page 1899
- "t_optmgmt() — Manage options for a transport endpoint" on page 1901
- "t_rcvconnect() — Receive the confirmation from a connect request" on page 1910

tcperror() — Print the error messages of a socket function

Standards

Standards / Extensions	C or C++	Dependencies
z/OS UNIX	both	

Format

```
#define _OPEN_SYS_SOCK_EXT
#include <sys/socket.h>
#include <stdio.h>
#include <errno.h>

void tcperror(const char *s);
```

General description

When a socket call produces an error, the call returns a negative value and the variable errno is set to an error value found in ERRNO.H. The tcperror() call prints a short error message describing the last error that occurred. If *s* is non-NULL, tcperror() prints the string *s* followed by a colon, followed by a space, followed by the error message, and terminated with a newline character. If *s* is NULL or points to a NULL string, only the error message and the newline character are output.

The tcperror() function is equivalent to the perror() function in UNIX.

Parameter
Description

s A NULL or NULL-terminated character string.

Returned value

tcperror() returns no values.

Example

The following are examples of the tcperror() call.

Example 1:

```
   if   ((s=socket(AF_INET, SOCK_DGRAM, 0)) < 0) {
        tcperror("socket()");
        exit(2);
   }
```

If the socket() call produces the error ENOMEM, socket() returns a negative value and sets errno to ENOMEM. When tcperror() is called, it prints the string:

```
      socket():  not enough storage (ENOMEM)
```

Example 2:

```
   if ((s=socket(AF_INET, SOCK_DGRAM, 0)) < 0)
      tcperror(NULL);
```

If the socket() call produces the error ENOMEM, socket() returns a negative value and sets errno to ENOMEM. When tcperror() is called, it prints the string:

```
       Not enough storage (ENOMEM)
```

Related information

- “errno.h” on page 23
- “stdio.h” on page 68
- “sys/socket.h” on page 74
- “perror() — Print error message” on page 1175

tcsendbreak() — Send a break condition to a terminal

Standards

Standards / Extensions	C or C++	Dependencies
POSIX.1 XPG4 XPG4.2 Single UNIX Specification, Version 3	both	

Format

```
#define _POSIX_SOURCE
#include <termios.h>

int tcsendbreak(int fildes, int duration);
```

General description

Sends a break condition to a terminal (indicated by *fildes*) that is using asynchronous serial data transmission. tcsendbreak() sends a continuous stream of zero bits for the specified *duration*. tcsendbreak() is the usual method of sending a BREAK on a line.

If tcsendbreak() is issued against a pseudoterminal, this function has no effect.

If tcsendbreak() is called from a background process group against the caller's controlling terminal, a SIGTTOU signal may be generated depending how the process is handling SIGTTOUs:

Processing for SIGTTOU	System Behavior
Default or signal handler	The SIGTTOU signal is generated, and the function is not performed. tcsendbreak() returns -1 and sets errno to EINTR.
Ignored or blocked	The SIGTTOU signal is not sent, and the function continues normally.

Returned value

If successful, tcsendbreak() returns 0.

If unsuccessful, tcsendbreak() returns -1 and sets errno to one of the following values:

Error Code
Description

EBADF
fildes is not a valid open file descriptor.

EINTR
A signal interrupted tcsendbreak().

EIO The process group of the process issuing the function is an orphaned, background process group, and the process issuing the function is not ignoring or blocking SIGTTOU.

ENOTTY
fildes is not associated with a terminal.

Example

CELEBT08

```
/* CELEBT08

   This example breaks terminal transmission.

 */
#define _POSIX_SOURCE
#include <stdio.h>
#include <termios.h>
#include <unistd.h>

main() {
  if (tcsendbreak(STDIN_FILENO, 100) != 0)
    perror("tcsendbreak() error");
  else
    puts("break sent");
}
```

Related information

- "termios.h" on page 78
- "tcdrain() — Wait until output has been transmitted" on page 1823
- "tcflow() — Suspend or resume data flow on a terminal" on page 1825
- "tcflush() — Flush input or output on a terminal" on page 1828
- "tcgetattr() — Get the attributes for a terminal" on page 1830
- "tcgetpgrp() — Get the foreground process group ID" on page 1835
- "tcsetattr() — Set the attributes for a terminal"
- "tcsetpgrp() — Set the foreground process group ID" on page 1858

tcsetattr() — Set the attributes for a terminal

Standards

Standards / Extensions	C or C++	Dependencies
POSIX.1 XPG4 XPG4.2 Single UNIX Specification, Version 3	both	

Format

```
#define _POSIX_SOURCE
#include <termios.h>

int tcsetattr(int fd, int when, const struct termios *termptr);
```

General description

tcsetattr() only works in an environment where either a controlling terminal exists, or stdin and stderr refer to tty devices. Specifically, it does not work in a TSO environment.

Changes the attributes associated with a terminal. New attributes are specified with a `termios` control structure. Programs should always issue a tcgetattr() first, modify the desired fields, and then issue a tcsetattr(). tcsetattr() should never be issued using a `termios` structure that was not obtained using tcgetattr(). tcsetattr() should use only a `termios` structure that was obtained by tcgetattr().

fd Indicates an open file descriptor associated with a terminal.

when Indicates a symbol, defined in the termios.h header file, specifying when to change the terminal attributes:

Symbol
Meaning

TCSANOW
The change should take place immediately.

TCSADRAIN
The change should take place after all output written to *fd* has been read by the master pseudoterminal. Use this value when changing terminal attributes that affect output.

TCSAFLUSH
The change should take place after all output written to *fd* has been sent; in addition, all input that has been received but not read should be discarded (flushed) before the change is made.

**termptr*
A pointer to a `termios` control structure containing the desired terminal attributes.

A `termios` structure contains the following members:

`tcflag_t c_iflag`
Input modes. `tcflag_t` is defined in the termios.h header file. Each bit in c_iflag indicates an input attribute and is associated with a symbol defined in the termios.h include file. All symbols are bitwise distinct. Thus c_iflag is the bitwise inclusive-OR of several of these symbols. Possible symbols are:

Symbol
Meaning

BRKINT
Indicates that an interrupt should be generated if the user types a BREAK.

ICRNL
Automatically converts input carriage returns to newline (line-feed) characters before they are passed to the application that reads the input.

IGNBRK
Ignores BREAK conditions. If this bit is set to 1, applications are not informed of any BREAK condition on the terminal; the setting of BRKINT has no effect.

If IGNBRK is 0 but BRKINT is 1, BREAK flushes all input and output queues. In addition, if the terminal is the controlling terminal of a foreground process group, the BREAK condition generates a single SIGINT signal for that foreground process group.

If both IGNBRK and BRKINT are 0, a BREAK condition is taken as the single input character NULL, if PARMRK is 0, and as the three input characters \377-NULL-NULL, if PARMRK is 1.

IGNCR

Ignores input carriage returns. If this bit is set to 1, the setting of ICRNL has no effect.

If IGNCR is 0 and ICRNL is 1, input carriage returns are converted to newline characters. For z/OS UNIX "NL" or '\n' is the EBCDIC character NL.

IGNPAR

Ignores input characters (other than BREAK) that have parity errors.

INLCR

Automatically converts input newline (line-feed) characters to carriage returns before they are passed to the application that reads the input.

INPCK

Enables input parity checking. If this bit is set to 0, it allows output parity generation without input parity errors. The enabling of input parity checking is independent of the enabling of parity checking in the control modes field. (See the description of “tcflag_t c_cflag,” which follows.) While the control modes may dictate that the hardware recognizes the parity bit, but the terminal special file does not check whether this bit is set correctly.

ISTRIP

Strips valid input bytes to 7 bits. If this bit is set to 0, the complete byte is processed.

Note: Do not set this bit for pseudoterminals, since it will make the terminal unusable. If you strip the first bit off of EBCDIC characters, you destroy all printable EBCDIC characters.

IUCLC

Map uppercase to lowercase on the received character. In locales other than the POSIX locale, the mapping is unspecified. Thus, this function only applies to the characters in the POSIX-portable character set that have lowercase equivalents, namely the characters A-Z.

Note:

This symbol is kept for historical reasons. It was part of the Legacy Feature in Single UNIX Specification, Version 2, but has been withdrawn and is not supported as part of Single UNIX Specification, Version 3.

If it is necessary to continue using this symbol in an application written for Single UNIX Specification, Version 3, define the feature test macro _UNIX03_WITHDRAWN before including any standard system headers. The macro exposes all interfaces and symbols removed in Single UNIX Specification, Version 3.

IXANY

Enable any character to restart output. If IXOFF and IXANY are set and a previous STOP character has been received, then receipt of any input character will cause the STOP condition to be removed. For pseudoterminals, data in the output queue is passed to the application during master read() processing, and slave pseudoterminal writes are allowed to proceed. The character which caused the output to restart is also processed normally as well (unless it is a STOP character).

IXOFF Enables start/stop input control. If this bit is set to 1, the system attempts to prevent the number of bytes in the input queue from exceeding the MAX_INPUT value. It sends one or more STOP characters to the terminal device when the input queue is in danger of filling up. The character used as the STOP character is dictated by the `c_cc` member of the `termios` structure. It is intended to tell the terminal to stop sending input for a while. The system transmits one or more START characters when it appears that there is space in the input queues for more input. Again, the character used as the START character is dictated by the `c_cc` member. It is intended to tell the terminal that it can resume transmission of input.

Note: Do not use IXOFF while in DBCS mode. If you intersperse STOP and START characters inside DBCS data while using IXOFF, you could corrupt output data,

IXON Enables start/stop output control. If the system receives a STOP character as input, it will suspend output on the associated terminal until a START character is received. An application reading input from the terminal does not see STOP or START characters; they are intercepted by the system, which does all the necessary processing.

If IXON is 0, any STOP or START characters read are passed on as input to an application reading from the terminal.

PARMRK

Marks characters with parity errors. If this bit is set to 1 and IGNPAR is 0, a byte with a framing or parity error is sent to the application as the characters \377 and NULL, followed by the data part of the byte that had the parity error. If ISTRIP is 0, a valid input character of \377 is sent as a pair of characters \377, \377 to avoid ambiguity.

If both PARMRK and IGNPAR are 0, a character with a framing or parity error is sent to the application as NULL.

`tcflag_t c_oflag`

Output modes. Each bit in `c_oflag` indicates an output attribute, and is associated with a symbol defined in the termios.h header file. Thus c_oflag is the bitwise inclusive-OR of a number of these symbols. Possible symbols are:

Symbol

Meaning

OPOST

Modifies lines of text in an implementation-defined way to appear appropriately on the terminal device. If this bit is set to 0, characters that an application puts out are sent without change.

OLCUC

If OPOST and OLCUC are set, then map lowercase to uppercase on the output. In locales other that the POSIX locale, the mapping is unspecified. Thus, this function only applies to the characters in the POSIX-portable character set that have uppercase equivalents, namely the characters a-z.

Note:

This symbol is kept for historical reasons. It was part of the Legacy Feature in Single UNIX Specification, Version 2, but has been withdrawn and is not supported as part of Single UNIX Specification, Version 3.

If it is necessary to continue using this symbol in an application written for Single UNIX Specification, Version 3, define the feature test macro _UNIX03_WITHDRAWN before including any standard system headers. The macro exposes all interfaces and symbols removed in Single UNIX Specification, Version 3.

ONLCR
If OPOST and ONLCR are set, the NL character is transmitted as the CR-NL character pair.

OCRNL
If OPOST and OCRNL are set, the CR character is transmitted as the NL character.

ONOCR
If OPOST and ONOCR are set, no CR character is transmitted if the current column is zero.

ONLRET
If OPOST and ONLRET are set, the NL character does the carriage return function; the column pointer is set to 0. If OPOST is set and ONLRET is not set, then the NL does the line-feed function; the column pointer is unchanged.

OFILL Fill characters are used for delay instead of using a timed delay.

OFDEL
The fill character is DEL. If OFILL is not set, then the fill character is NUL.

NLDLY
Delay associated with newline character.

NL0 No delay.

NL1 0.10 seconds delay. If ONLRET is set, then carriage-return delays are used instead of newline delays. If OFILL is set, then two fill characters are transmitted.

CRDLY
Delay associated with carriage-return character.

CR0 No delay.

CR1 Delay dependent on column position, or if OFILL is set then two fill characters are transmitted.

CR2 0.10 seconds delay, or if OFILL is set then four fill characters are transmitted.

CR3 0.15 seconds delay.

TABDLY
Delay associated with tab character.

TAB0 No horizontal tab processing.

TAB1 Delay dependent on column position, or if OFILL is set then two fill characters are transmitted.

TAB2 0.10 seconds delay, or if OFILL is set then two fill characters are transmitted.

TAB3 Tabs are expanded into spaces.

BSDLY
Delay associated with backspace character.

BS0 No delay.

BS1 0.05 seconds delay, or if OFILL is set then one fill character is transmitted.

VTDLY
Delay associated with vertical-tab processing.

VT0 No delay.

VT1 2 seconds delay.

FFDLY
Delay associated with form-feed processing.

FF0 No delay.

FF1 2 seconds delay.

`tcflag_t c_cflag`
Control modes. Each bit in `c_cflag` indicates a control attribute and is associated with a symbol defined in the termios.h header file. Thus `c_cflag` is the bitwise inclusive-OR of several of these symbols. Possible symbols are:

Symbol
Meaning

CLOCAL
Ignores modem status lines. A call to open() returns immediately without waiting for a modem connection to complete. If this bit is set to 0, modem status lines are monitored and open() waits for the modem connection.

CREAD
Enables reception. If this bit is set to 0, no input characters are received from the terminal.

Using z/OS UNIX pseudoterminal support, this bit is always enabled and set to 1.

CSIZE Is a collection of bits indicating the number of bits per byte (not counting the parity bit, if any). These bits specify byte size for both transmission and reception. Possible settings of CSIZE are given with the following symbols:

```
CS5 - 5 bits per byte
CS6 - 6 bits per byte
CS7 - 7 bits per byte
CS8 - 8 bits per byte
```

Using z/OS UNIX pseudoterminal support, all values are accepted, but CSIZE is changed to `CS8`. Using z/OS UNIX Outboard Communications Server (OCS) support, the specified value is used.

CSTOPB
Sends two stop bits when necessary. If CSTOPB is 0, only one stop bit is used.

Using z/OS UNIX pseudoterminal support, this bit is always 0. Using z/OS UNIX OCS support, the specified value is used.

HUPCL
Lowers the modem control lines for a port when the last process that has the port open closes the port (or the process ends). In other words, this tells the system to hang up when all relevant processes have finished using the port.

For pseudoterminals HUPCL controls what happens when the slave pseudoterminals is closed. If HUPCL is set when the last file descriptor for the slave pseudoterminal is closed, then the slave pseudoterminal cannot be re-opened. The master terminal has to be closed and re-opened before the pair can be used again.

PARENB
Enables parity generation and detection. A parity bit is added to each character on output, and expected from each character on input.

Under z/OS UNIX, if this bit is set to 1 in a request, it is ignored. It is always set to 0. Using z/OS UNIX OCS support, the specified value is used.

PARODD
Indicates odd parity (when parity is enabled). If PARODD is 0, even parity is used (when parity is enabled).

Under z/OS UNIX, if this bit is set to 1 in a request, it is ignored. It is always set to 0. Using z/OS UNIX OCS support, the specified value is used.

If the object for which the control modes are set is not an asynchronous serial connection, some bits may be ignored. For example, on a network connection, it may not be possible to set the baud rate.

`tcflag_t c_lflag`
Local modes. Each bit in `c_lflag` indicates a local attribute, and is associated with a symbol defined in the termios.h include file. Thus `c_lflag` is the bitwise inclusive-OR of a number of these symbols. Possible symbols are:

Symbol
Meaning

ECHO Echoes input characters back to the terminal. If this is bit is 0, input characters are not echoed.

ECHOE
Echoes the ERASE character as an error-correcting backspace. When the user inputs an ERASE character, the terminal erases the last character in the current line from the display (if possible). The

character used as the ERASE character is dictated by the `c_cc` member of the `termios` structure. ECHOE has an effect only if the ICANON bit is 1.

ECHOK
Either causes the terminal to erase the line from the display, or echoes the KILL character followed by an `\n` character. ECHOK has an effect only if the ICANON bit is set to 1.

ECHONL
Echoes the newline (line-feed) character '\n' even if the ECHO bit is off. ECHONL has an effect only if the ICANON bit is set to 1.

ICANON
Enables canonical input processing, also called *line mode*. Input is not delivered to the application until an entire line has been input. The end of a line is indicated by a newline, End Of File (EOF), or EOL character (where the character used as the EOL character is directed by the `c_cc` member of the `termios` structure [described shortly]). Canonical input processing uses the ERASE character to erase a single input character, and the KILL character to erase an entire line. The MAX_CANON value specifies the maximum number of bytes in an input line in canonical mode.

If ICANON is 0, read requests take input directly from the input queue; the system does not wait for the user to enter a complete line. This is called *noncanonical mode*. ERASE and KILL characters are not handled by the system but passed directly to the application. See also the descriptions of MIN and TIME in the `c_cc` member.

IEXTEN
Enables extended implementation-defined functions. These are not defined, and IEXTEN is always set to 0.

If the ERASE, KILL or EOF character is preceded by a backslash character, the special character is placed in the input queue without doing the "special character" processing and the backslash is discarded.

ISIG If ISIG is set to 1, signals are generated if special control characters are entered. SIGINT is generated if INTR is entered; SIGQUIT is generated if QUIT is entered; and SIGTSTP is generated if SUSP is entered and job control is supported. The special control characters are controlled by the `c_cc` member.

If ISIG is 0, the system does not generate signals when these special control characters are entered.

NOFLSH
If this bit is set to 1, the system does not flush the input and output queues if a signal is generated by one of the special characters described in ISIG above. If NOFLSH is set to 0, the queues are flushed if one of the special characters is found.

TOSTOP
If this bit is set to 1, a SIGTTOU signal is sent to the process group of a process that tries to write to a terminal when it is not in the terminal's foreground process group. However, if the process that tries to write to the terminal is blocking or ignoring SIGTTOU signals, the system does not raise the SIGTTOU signal.

If TOSTOP is 0, output from background processes is output to the current output stream, and no signal is raised.

XCASE

Do canonical lower and canonical upper presentation. In locales other than the POSIX locale, the effect is unspecified. XCASE set by itself makes all uppercase letters on input and output be preceded by a "\" character.

Some terminals can generate lowercase characters, but can display only uppercase characters. For these terminals, XCASE would be used by itself. Other terminals cannot generate lowercase characters either. For these terminals, XCASE would be used with IUCLC to generate lowercase characters when characters are typed without the backslash, and uppercase characters when the typed character is preceded by a backslash.

If a terminal can generate only uppercase characters, but can display either upper or lowercase, then XCASE would be used with OLCUC.

Note:

This symbol is kept for historical reasons. It was part of the Legacy Feature in Single UNIX Specification, Version 2, but has been withdrawn and is not supported as part of Single UNIX Specification, Version 3.

If it is necessary to continue using this symbol in an application written for Single UNIX Specification, Version 3, define the feature test macro _UNIX03_WITHDRAWN before including any standard system headers. The macro exposes all interfaces and symbols removed in Single UNIX Specification, Version 3.

cc_t c_cc`[NCCS]`

Control characters. This is an array of characters that may have special meaning for terminal handling. You can access characters in this array using subscripts that are symbols defined in the termios.h header file. For example, the STOP character is given by c_cc[VSTOP]. Possible subscript symbols are:

Symbol
Meaning

VEOF Gives the End Of File character EOF. It is recognized only in canonical (line) mode. When this is found in input, all bytes waiting to be read are immediately passed to the application without waiting for the end of the line. The EOF character itself is discarded. If EOF occurs at the beginning of a line, the `read` function that tries to read that line receives an End Of File (EOF) indication. Note that EOF results in End Of File only if it is at the beginning of a line; if it is preceded by one or more characters, it indicates only End Of Line (EOL).

VEOL Gives the End Of Line character EOL. It is recognized only in canonical (line) mode. This is an alternate character for marking the end of a line (in addition to the newline `\n`).

VERASE

Gives the ERASE character. It is recognized only in canonical (line) mode. It deletes the last character in the current line. It cannot delete beyond the beginning of the line.

VINTR
Gives the interrupt character INTR. It is recognized only if ISIG is set to 1 in `c_lflag`. If the character is received, the system sends a SIGINT signal to all the processes in the foreground process group that has this device as its controlling terminal.

VKILL
Gives the KILL character. It is recognized only in canonical (line) mode. It deletes the entire contents of the current line.

VMIN Gives the MIN value for noncanonical mode processing.

This is the minimum number of bytes that a call to `read` should return in noncanonical mode; it is not used in canonical mode.

If both MIN and TIME are greater than 0, `read` returns when MIN characters are available or when the timer associated with TIME runs out (whichever comes first). The timer starts running as soon as a single character has been entered; if there is already a character in the queue when `read` is called, the timer starts running immediately.

If MIN is greater than zero and TIME is zero, `read` waits for MIN characters to be entered, no matter how long that takes.

If MIN is zero and TIME is greater than zero, `read` returns when the timer runs out or when a single character is received (whichever comes first). `read` returns either one character (if one is received) or zero (if the timer runs out). The timer starts running as soon as `read` is called. (Contrast this with the case where MIN and TIME are both greater than zero, and the timer starts only when a character is received.)

If both MIN and TIME are zero, `read` returns immediately from every call. It returns the number of bytes that are immediately available, up to the maximum specified in the call to `read`.

VQUIT
Gives the quit character QUIT. It is recognized only if ISIG is set to 1 in `c_lflag`. If the character is received, the system sends a SIGQUIT signal to all the processes in the foreground process group that has this device as its controlling terminal.

VSUSP
Gives the suspend character SUSP. It is recognized only if ISIG is set to 1 in `c_lflag`. If the character is received, the system sends a SIGTSTP signal to all the processes in the foreground process group that has this device as its controlling terminal.

VTIME
Gives the TIME value, used in noncanonical mode in connection with MIN. It expresses a time in terms of tenths of a second.

VSTOP
Gives the STOP character. You can use this to suspend output temporarily when IXON is set to 1 in `c_iflag`. Users can enter the STOP character to prevent output from running off the top of a display screen.

VSTART
Gives the START character. You can use this to resume suspended output when IXON is set to 1 in `c_iflag`.

When tcsetattr() is called from a background session against a controlling terminal, SIGTTOU processing is as follows:

Processing for SIGTTOU	Expected Behavior
Default or signal handler	The SIGTTOU signal is generated, and the function is not performed. tcsetattr() returns -1 and sets errno to EINTR.
Ignored or blocked	The SIGTTOU signal is not sent, and the function continues normally.

Note: The tcsetattr() function has a dependency on the level of the Enhanced ASCII Extensions. See "Enhanced ASCII support" on page 2113 for details.

Returned value

If successful, tcsetattr() returns 0.

If unsuccessful, tcsetattr() returns -1 and sets errno to one of the following values:

Error Code
Description

EBADF
fildes is not a valid open file descriptor.

EINTR
A signal interrupted tcsetattr().

EINVAL
when is not a recognized value, or some entry in the supplied `termios` structure had an incorrect value.

EIO The process group of the process issuing the function is an orphaned, background process group, and the process issuing the function is not ignoring or blocking SIGTTOU.

ENOTTY
fildes is not associated with a terminal.

Example

CELEBT09

```
/* CELEBT09

   The following attributes changes the terminal attributes.

 */
#define _POSIX_SOURCE
#include <termios.h>
#include <unistd.h>
#include <stdio.h>

main() {
  struct termios term1, term2;

  if (tcgetattr(STDIN_FILENO, &term1) != 0)
```

```
    perror("tcgetattr() error");
  else {
    printf("the original end-of-file character is x'%02x'\n",
           term1.c_cc[VEOF]);
    term1.c_cc[VEOF] = 'z';
    if (tcsetattr(STDIN_FILENO, TCSANOW, &term1) != 0)
      perror("tcsetattr() error");
    if (tcgetattr(STDIN_FILENO, &term1) != 0)
      perror("tcgetattr() error");
    else
      printf("the new end-of-file character is x'%02x'\n",
             term1.c_cc[VEOF]);
  }
}
```

Output

```
the original End Of File character is x'37'
the new End Of File character is x'a9'
```

Related information

- "termios.h" on page 78
- "cfgetispeed() — Determine the input baud rate" on page 254
- "cfgetospeed() — Determine the output baud rate" on page 256
- "cfsetispeed() — Set the input baud rate in the termios" on page 258
- "cfsetospeed() — Set the output baud rate in the termios" on page 260
- "open() — Open a file" on page 1152
- "read() — Read from a file or socket" on page 1376
- "tcdrain() — Wait until output has been transmitted" on page 1823
- "tcflow() — Suspend or resume data flow on a terminal" on page 1825
- "tcflush() — Flush input or output on a terminal" on page 1828
- "tcgetattr() — Get the attributes for a terminal" on page 1830
- "tcgetpgrp() — Get the foreground process group ID" on page 1835
- "tcsendbreak() — Send a break condition to a terminal" on page 1842
- "tcsetpgrp() — Set the foreground process group ID" on page 1858

__tcsetcp() — Set terminal code page names

Standards

Standards / Extensions	C or C++	Dependencies
z/OS UNIX	both	

Format

```
#define _OPEN_SYS_PTY_EXTENSIONS
#include <termios.h>

int __tcsetcp(int fildes, size_t termcplen, const struct __termcp *termcpptr);
```

General description

The __tcsetcp() function sets (or changes) the terminal session code page information contained in the termcp structure.

The following arguments are used:

fildes The file descriptor of the terminal for which you want to get the code page names and CPCN capability.

termcplen
The length of the passed termcp structure.

termcpptr
A pointer to a __termcp structure.

Use the __tcsetcp() function to send new code page information to the data conversion point in order to change the data conversion environment for the terminal session. This function is used with terminal devices that support the "forward code page names only" Code Page Change Notification (CPCN) capability. The z/OS UNIX pseudotty (pty) device driver supports this capability.

For terminal sessions that use the z/OS UNIX pty device driver, the data conversion point is the application that uses the master pty device. An example data conversion point is the z/OS UNIX **rlogin** server. Here, **rlogin** uses CPCN functions to determine the ASCII source and/or EBCDIC target code pages to use for the conversion of the terminal data. During its processing of the __tcsetcp() function, the pty device driver applies the __termcp structure once the pty outbound data queue is drained. When this occurs, the pty input data queue is also flushed and a TIOCXPKT_CHCP packet exception event is generated if extended packet mode is enabled (PKTXTND is set in the termios structure) to notify the application using the master pty that the code page information has been changed. The master pty application can then use the __tcgetcp() function to retrieve the new code page information and establish a new data conversion environment.

The __tcsetcp() function is supported by both the master and slave pty device drivers, however, CPCN functions first must be enabled by the application that uses the master pty; enabling CPCN functions is performed by the system during the initial __tcsetcp() invocation against the master pty device. Once the __tcsetcp() function is performed against the master pty then it may be subsequently issued against the slave pty.

Note: The data conversion for a z/OS UNIX terminal session is performed on a session (terminal file) basis. If you change the data conversion characteristics for one file descriptor, the new data conversion will apply to all open file descriptors associated with this terminal file.

Attention: Use this service carefully. By changing the code pages for the data conversion you may cause unpredictable behavior in the terminal session if the actual data used for the session is not encoded to the specified source (ASCII) and target (EBCDIC) code pages.

A __termcp structure contains the following members:

__tccp_flags
Flags. The following symbols are defined as bitwise distinct values. Thus, __tccp_flags is the bitwise inclusive-OR of these symbols:

Symbol
Meaning

_TCCP_BINARY
Use _TCCP_BINARY to notify the data conversion point to stop data conversion. If this flag is set, the source and target code page names (__tccp_fromname and __tccp_toname respectively) are not changed from their current values.

Attention: Use this option carefully. Once the data conversion is disabled the z/OS UNIX Shell cannot be used until the data conversion is re-enabled, using valid code pages for the terminal session.

_TCCP_FASTP
Use _TCCP_FASTP to indicate to the data conversion point (for example, **rlogin**) that the data conversion specified by the source and target code page names can be performed locally to the application. This is valid any time that a table-driven conversion can be performed. For example, the data conversion point (application) could use the z/OS UNIX iconv() service to build the local data conversion tables and perform all data conversion using the local tables instead of using iconv() in subsequent conversions. This provides for better-performing data conversion.

__tccp_fromname
The source code page name; typically this is the ASCII code page name. __tccp_fromname is a NULL-terminated string with a maximum length of _TCCP_CPNAMEMAX, including the NULL (\00) character.

__tccp_fromname is case-sensitive.

__tccp_toname
The target code page name; typically this is the EBCDIC code page name. __tccp_toname is a NULL-terminated string with a maximum length of _TCCP_CPNAMEMAX, including the NULL (\00) character.

__tccp_toname is case-sensitive.

When __tcsetcp() is issued against the slave pty from a process in a background process group, SIGTTOU processing is as follows:

Processing for SIGTTOU	Expected Behavior
Default or signal handler	The **SIGTTOU** signal is generated. The function is not performed. __tcsetcp() returns -1 and sets errno to EINTR.
Ignored or blocked	The **SIGTTOU** signal is not sent. The function continues normally.

Note: The __tcsetcp() function has a dependency on the level of the Enhanced ASCII Extensions. See "Enhanced ASCII support" on page 2113 for details.

Returned value

If successful, __tcsetcp() returns 0.

If unsuccessful, __tcsetcp() returns -1 and sets errno to one of the following values:

Error Code
Description

EBADF
fildes is not a valid open file descriptor.

EINTR
A signal interrupted the call.

EINVAL
The value of *termcplen* was invalid.

EIO The process group of the process issuing the function is an orphaned, background process group, and the process issuing the function is not ignoring or blocking SIGTTOU.

ENODEV
One of the following error conditions exist:
- CPCN functions have not been enabled.

  The __tcsetcp() function must be issued against the master pty before any CPCN function can be issued against the slave pty.
- The terminal device driver does not support the “forward code page names only” CPCN capability.

ENOTTY
The file associated with *fildes* is not a terminal device.

Example

The following example retrieves the CPCN capability and code pages and then changes the ASCII code page to IBM-850.

```
#define _OPEN_SYS_PTY_EXTENSIONS
#include <unistd.h>
#include <stdio.h>
#include <fcntl.h>
#include <termios.h>


void main(void)
{
  struct __termcp mytermcp;
  int rv;
  int cterm_fd;

  if ((cterm_fd = open("/dev/tty",O_RDWR)) == -1)
    printf("No controlling terminal established.\n");
  else {
    if ((rv = __tcgetcp(STDIN_FILENO,sizeof(mytermcp),&mytermcp))== -1)
      perror("__tcgetcp() error");
    else {
      if (rv== _CPCN_NAMES) {
        if (_TCCP_BINARY == (mytermcp.__tccp_flags & _TCCP_BINARY))
          printf("Binary mode is in effect.  No change made.\n");
        else {
          strcpy(mytermcp.__tccp_fromname,"IBM-850");
          if (__tcsetcp(STDOUT_FILENO,sizeof(mytermcp),&mytermcp)!=0)
            perror("__tcsetcp() error");
          else
            printf("ASCII code page changed to IBM-850.\n");
        } /*not binary mode */
      } /* _CPCN_NAMES */
    } /* __tcgetcp success */
    close(cterm_fd);
  } /* controlling terminal established */
} /* main */
```

Output

```
ASCII code page changed to IBM-850.
```

Related information

- “termios.h” on page 78
- “__tcgetcp() — Get terminal code page names” on page 1832

- "__tcsettables() — Set terminal code page names and conversion tables" on page 1860

tcsetpgrp() — Set the foreground process group ID

Standards

Standards / Extensions	C or C++	Dependencies
POSIX.1 XPG4 XPG4.2 Single UNIX Specification, Version 3	both	

Format

```
#define _POSIX_SOURCE
#include <unistd.h>

int tcsetpgrp(int fildes, pid_t newid);
```

General description

Sets the process group ID (PGID) of the foreground process group associated with the terminal referred to by *fildes*. This terminal must be the controlling terminal of the process calling tcsetpgrp() and must be currently associated with the session of the calling process. *newid* must match a PGID of a process in the same session as the calling process.

After the PGID associated with the terminal is set, reads by the process group formerly associated with the terminal fail or cause the process group to stop from a SIGTTIN signal. Writes may also cause the process to stop (from a SIGTTOU signal), or they may succeed, depending on how tcsetattr() sets TOSTOP and the signal options for SIGTTOU.

fildes can be any of the descriptors representing the controlling terminal (such as standard input, standard output, and standard error), and the function affects future access from any file descriptor in use for the terminal. Consider using redirection when specifying the file descriptor.

If tcsetpgrp() is called from a background process group against the caller's controlling terminal, a SIGTTOU signal may be generated depending how the process is handling SIGTTOUs:

Processing for SIGTTOU	System Behavior
Default or signal handler	The SIGTTOU signal is generated, and the function is not performed. tcsetpgrp() returns -1 and sets errno to EINTR.
Ignored or blocked	The SIGTTOU signal is not sent, and the function continues normally.

Returned value

If successful, tcsetpgrp() returns 0.

If unsuccessful, tcsetpgrp() returns -1 and sets errno to one of the following values:

Error Code
Description

EBADF
fildes is not a valid open file descriptor.

EINTR
A signal interrupted the tcsetpgrp() function.

EINVAL
The *newid* value is not supported by this implementation.

ENOTTY
The process calling tcsetpgrp() does not have a controlling terminal, or *fildes* is not associated with the controlling terminal, or the controlling terminal is no longer associated with the session of the calling process.

EPERM
The *newid* value is supported by the implementation but does not match the process group ID of any process in the same session as the process calling tcsetpgrp().

Example

CELEBT10

```
/* CELEBT10

   This example changes the PGID.

 */
#define _POSIX_SOURCE
#include <termios.h>
#include <unistd.h>
#include <sys/wait.h>
#include <stdio.h>
#include <signal.h>

main() {
  pid_t pid;
  int status;

  if (fork() == 0)
    if ((pid = tcgetpgrp(STDOUT_FILENO)) < 0)
      perror("tcgetpgrp() error");
    else {
      printf("original foreground process group id of stdout was %d\n",
             (int) pid);
      if (setpgid(getpid(), 0) != 0)
        perror("setpgid() error");
      else {
        printf("now setting to %d\n", (int) getpid());
        if (tcsetpgrp(STDOUT_FILENO, getpid()) != 0)
          perror("tcsetpgrp() error");
        else if ((pid = tcgetpgrp(STDOUT_FILENO)) < 0)
          perror("tcgetpgrp() error");
        else
          printf("new foreground process group id of stdout was %d\n",
                 (int) pid);
      }
    }

  else wait(&status);
}
```

Output

```
original foreground process group id of stdout was 2228230
now setting to 2949128
new foreground process group id of stdout was 2949128
```

Related information

- "unistd.h" on page 82
- "tcdrain() — Wait until output has been transmitted" on page 1823
- "tcflow() — Suspend or resume data flow on a terminal" on page 1825
- "tcflush() — Flush input or output on a terminal" on page 1828
- "tcgetattr() — Get the attributes for a terminal" on page 1830
- "tcgetpgrp() — Get the foreground process group ID" on page 1835
- "tcsendbreak() — Send a break condition to a terminal" on page 1842
- "tcsetattr() — Set the attributes for a terminal" on page 1843

__tcsettables() — Set terminal code page names and conversion tables

Standards

Standards / Extensions	C or C++	Dependencies
z/OS UNIX	both	

Format

```
#define _OPEN_SYS_PTY_EXTENSIONS
#include <termios.h>

int __tcsettables(int fildes, size_t termcplen,
                  const struct __termcp *termcpptr,
                  const char atoe[256],
                  const char etoa[256]);
```

General description

The __tcsettables() function changes the data conversion environment for terminal sessions that support the "forward code page names and tables" Code Page Change Notification (CPCN) capability. The OCS remote-tty (rty) device driver supports this capability.

The following arguments are used:

fildes The file descriptor of the terminal for which you want to set the code page names and data conversion tables.

termcplen
The length of the passed termcp structure.

termcpptr
A pointer to a __termcp structure. A __termcp structure contains the following members:

__tccp_flags
Flags. The following symbols are defined as bitwise distinct values. Thus, __tccp_flags is the bitwise inclusive-OR of these symbols:

Symbol
Meaning

_TCCP_BINARY
Use _TCCP_BINARY to notify the data conversion point to

stop data conversion. If this flag is set the source and target code page names (__tccp_fromname and __tccp_toname respectively) are not changed, and the data conversion tables *atoe* and *etoa* are not used.

Attention: Use this option carefully. Once the data conversion is disabled the z/OS shell cannot be used until the data conversion is re-enabled, using valid code pages for the terminal session.

_TCCP_FASTP

Use _TCCP_FASTP to indicate to the data conversion point that the data conversion specified by the source and target code page names can be performed locally by the application that performs the data conversion. This is valid any time that a table-driven conversion can be performed.

This value is not used by the OCS rty device driver and thus has no effect.

__tccp_fromname

The source code page name; typically this is the ASCII code page name. __tccp_fromname is a NULL-terminated string with a maximum length of _TCCP_CPNAMEMAX, including the NULL (\00) character.

__tccp_fromname is case-sensitive.

__tccp_toname

The target code page name; typically this is the EBCDIC code page name. __tccp_toname is a NULL-terminated string with a maximum length of _TCCP_CPNAMEMAX, including the NULL (\00) character.

__tccp_toname is case-sensitive.

const char *atoe***[256]**

A 256-byte data conversion table for the source-to-target (ASCII-to-EBCDIC) data conversion. The byte offset into this table corresponds to the character code from the source (ASCII) code page. The data value at each offset is the “converted” target (EBCDIC) character code.

const char *etoa***[256]**

A 256-byte data conversion table for the target-to-source (EBCDIC-to-ASCII) data conversion. The byte offset into this table corresponds to the character code from the target (EBCDIC) code page. The data value at each offset is the “converted” source (ASCII) character code.

Note: The data conversion for a z/OS UNIX terminal session is performed on a session (terminal file) basis. If you change the data conversion characteristics for one file descriptor, the new data conversion will apply to all open file descriptors associated with this terminal file.

For terminal sessions that use the OCS rty device driver, the ASCII/EBCDIC data conversion is performed outboard by OCS on the AIX server system. Use the __tcsettables() function to specify new code pages and conversion tables to be used in the data conversion.

During its processing of the __tcsettables() function, the OCS rty device driver applies the new code page names once the outbound data queue is drained. When this occurs, the rty input data queue is also flushed and the new conversion environment takes effect.

OCS processing of the *atoe* and *etoa* arguments is as follows:

- If the code page names specified in the __termcp structure are for supported double-byte data conversion then the *atoe* and *etoa* arguments are not used. The following double-byte translation is supported for OCS sessions:
- If __fromname specifies **ISO8859-1** and __toname specifies **IBM-1047** then OCS uses its own data conversion tables and *atoe* and *etoa* arguments are not used.
- Otherwise the conversion tables in *atoe* and *etoa* are used.

Attention: Use this service carefully. By changing the code pages for the data conversion you may cause unpredictable behavior in the terminal session if the actual data used for the session is not encoded to the specified source (ASCII) and target (EBCDIC) code pages.

When __tcsettables() is issued from a process in a background process group, SIGTTOU is processing in this way:

Processing for SIGTTOU	Expected Behavior
Default or signal handler	The **SIGTTOU** signal is generated. The function is not performed. __tcsettables() returns -1 and sets errno to EINTR.
Ignored or blocked	The **SIGTTOU** signal is not sent. The function continues normally.

Returned value

If successful, __tcsettables() returns 0.

If unsuccessful, __tcsettables() returns -1 and sets errno to one of the following values:

Error Code
Description

EBADF
fildes is not a valid open file descriptor.

EINTR
A signal interrupted the call.

EINVAL
One of the following error conditions exists:

- The value of *termcplen* was invalid.
- An invalid combination of multibyte code page names was specified in the __termcp structure.

  One of the following applies:

  - The source code page specified in __tccp_fromname specified a supported ASCII multibyte code page and the __tccp_toname did not specify a supported EBCDIC multibyte code page.
  - The target code page specified in __tccp_toname specified a supported EBCDIC multibyte code page and the __tccp_fromname did not specify a supported ASCII multibyte code page.

EIO The process group of the process issuing the function is an orphaned, background process group, and the process issuing the function is not ignoring or blocking SIGTTOU.

ENODEV
The terminal device driver does not support the "forward code page names and tables" CPCN capability.

ENOTTY
The file associated with *fildes* is not a terminal device.

Example

The following example retrieves the current code pages used in the data conversion and CPCN capability. The conversion tables using ASCII code page IBM-850 and the current EBCDIC code page are generated and exported to the data conversion point using __tcsettables().

```
#define _OPEN_SYS_PTY_EXTENSIONS

#include <fcntl.h>
#include <unistd.h>
#include <stdio.h>
#include <termios.h>
#include <iconv.h>

main()
{
  struct __termcp mytermcp;       /* local __termcp               */
  unsigned char *intabptr;        /* pointer to input table       */
  unsigned char *outtabptr;       /* pointer to output table      */
  unsigned char intab[256],
                atoe[256],
                etoa[256];        /* conversion tables            */
  iconv_t  cd;                    /* conversion descriptor        */
  size_t inleft;                  /* number of bytes left in input  */
  size_t outleft;                 /* number of bytes left in output */
  int i;                          /* loop variable                */
  int rv;                         /* return value                 */
  int cterm_fd;                   /* file descriptor for controlling
                                     terminal                     */

  if ((cterm_fd = open("/dev/tty",O_RDWR)) == -1)
  {
    printf("No controlling terminal established.  ");
    printf("Code pages were not changed.\n");
    exit(0);
  }
  if ((rv = __tcgetcp(cterm_fd,sizeof(mytermcp),&mytermcp))== -1)
  {
    perror("__tcgetcp() error");
    exit(1);
  }

  if (_TCCP_BINARY == (mytermcp.__tccp_flags & _TCCP_BINARY))
  {
    printf("Binary mode is in effect.  No change made.\n");
    exit(0);
  }

  if (rv == _CPCN_TABLES) {

    /* build ASCII -> EBCDIC conversion table */

    strcpy(mytermcp.__tccp_fromname,"IBM-850");
    if ((cd = iconv_open(mytermcp.__tccp_toname,
                         mytermcp.__tccp_fromname)) ==
        (iconv_t) (-1)) {
        fprintf(stderr,"Cannot open converter from %s to %s\n",
           mytermcp.__tccp_fromname,mytermcp.__tccp_toname);
        exit(1);
    }

    /* build input table with character values of 00 - FF */

    for (i=0; i<256; i++ ) {
       intab[i] = (unsigned char) i;
    } /* endfor */

    inleft = 256;
    outleft = 256;
    intabptr = intab;
    outtabptr = atoe;

    /* build ASCII -> EBCDIC conversion table. */

    rv = iconv(cd,&intabptr, &inleft, &outtabptr, &outleft);
    if (rv == -1) {
       fprintf(stderr,"Error in building ASCII to EBCDIC table\n");
       exit(1);
    }
    iconv_close(cd);

    /* build EBCDIC -> ASCII conversion table */

    if ((cd = iconv_open(mytermcp.__tccp_fromname,
                         mytermcp.__tccp_toname)) ==
        (iconv_t) (-1)) {
        fprintf(stderr,"Cannot open converter from %s to %s\n",
           mytermcp.__tccp_toname,mytermcp.__tccp_fromname);
        exit(1);
    }
    inleft = 256;
    outleft = 256;
    intabptr = intab;
    outtabptr = etoa;
    rv = iconv(cd,&intabptr, &inleft, &outtabptr, &outleft);
    if (rv == -1) {
       fprintf(stderr,"Error in building EBCDIC to ASCII table\n");
       exit(1);
    }
    iconv_close(cd);

    /*
     * Change the data conversion to use IBM-850 as the ASCII source
     */

    if (__tcsettables(cterm_fd, sizeof(mytermcp), &mytermcp,
        atoe,etoa) == -1) {
      perror("__tcsettables() error");
      exit(1);
    } else {
       printf("Data conversion now using ASCII IBM-850\n");
    } /* endif */
  } /* endif */
  close(cterm_fd);
} /* main */
```

Output

```
Data conversion now using ASCII IBM-850.
```

Related information

- "termios.h" on page 78
- "__tcgetcp() — Get terminal code page names" on page 1832
- "__tcsetcp() — Set terminal code page names" on page 1854

tdelete() — Binary tree delete

Standards

Standards / Extensions	C or C++	Dependencies
XPG4 XPG4.2 Single UNIX Specification, Version 3	both	

Format

```
#define _XOPEN_SOURCE
#include <search.h>

void *tdelete(const void *__restrict__ key, void **__restrict__ rootp,
              int (*compar)(const void *, const void *));
```

General description

The tdelete() function deletes a node from a binary search tree. The arguments are the same as for the tsearch() function. The variable pointed to by *rootp* will be changed if the deleted node was the root of the tree. tdelete() returns a pointer to the parent of the deleted node, or a NULL pointer if the node is not found. If the deleted node was the root of the tree, the function returns a pointer to the deleted node, since it had no parent. It frees the storage for this node before returning, so the contents of storage at the returned address are unreliable in this case.

Comparisons are made with a user-supplied routine, the address of which is passed as the *compar* argument. This routine is called with two arguments, the pointers to the elements being compared. The user-supplied routine must return an integer less than, equal to or greater than 0, according to whether the first argument is to be considered less than, equal to or greater than the second argument. The comparison functions need not compare every byte, so arbitrary data may be contained in the elements in addition to the values being compared.

Threading Behavior: see "tsearch() — Binary tree search" on page 1920.

Special behavior for C++: Because C and C++ linkage conventions are incompatible, tdelete() cannot receive a C++ function pointer as the comparator argument. If you attempt to pass a C++ function pointer to tdelete(), the compiler will flag it as an error. You can pass a C or C++ function to tdelete() by declaring it as extern "C".

Returned value

If successful, tdelete() returns a pointer to the parent of the deleted node.

If the node is not found, tdelete() returns a NULL pointer.

If *rootp* is a NULL pointer on entry, tdelete() returns a NULL pointer.

No errors are defined.

Related information

- “search.h” on page 62
- “bsearch() — Search arrays” on page 220
- “hsearch() — Search hash tables” on page 817
- “lsearch() — Linear search and update” on page 1019
- “tfind() — Binary tree find node” on page 1870
- “tsearch() — Binary tree search” on page 1920
- “twalk() — Binary tree walk” on page 1935

telldir() — Current location of directory stream

Standards

Standards / Extensions	C or C++	Dependencies
XPG4 XPG4.2 Single UNIX Specification, Version 3	both	

Format

```
#define _XOPEN_SOURCE
#include <dirent.h>

long telldir(DIR *dirp);
```

General description

The telldir() function obtains the current location associated with the directory stream specified by *dirp*.

If the most recent operation on the directory stream was a seekdir(), then the directory position returned from telldir() is the same as that supplied as a **loc** argument to seekdir().

Returned value

If successful, telldir() returns the current location of the specified directory stream.

If the *dirp* argument supplied is NULL or invalid, telldir() returns -1 and sets errno to one of the following values:

Error Code
  Description

EBADF
  The *dirp* argument was invalid.

Related information

- “dirent.h” on page 22
- “stdio.h” on page 68
- “sys/types.h” on page 75
- “closedir() — Close a directory” on page 291
- “opendir() — Open a directory” on page 1158
- “readdir() — Read an entry from a directory” on page 1382
- “rewinddir() — Reposition a directory stream to the beginning” on page 1446
- “seekdir() — Set position of directory stream” on page 1469

tempnam() — Generate a temporary file name

Standards

Standards / Extensions	C or C++	Dependencies
XPG4 XPG4.2 Single UNIX Specification, Version 3	both	POSIX(ON)

Format

```
#define _XOPEN_SOURCE
#include <stdio.h>

char *tempnam(const char *dir, const char *pfx);
```

General description

The tempnam() function generates a path name that may be used for a temporary file. If the environment variable TMPDIR is set, then the directory it specifies will be used as the directory part of the generated path name if it is accessible. Otherwise, if the *dir* argument is non-NULL and accessible, it will be used in the generated path name. Otherwise, the value of {P_tmpdir} defined in the <stdio.h> header is used as the directory component of the name. If that is inaccessible, then /tmp is used.

The *pfx* argument can be used to specify an initial component of the file name part of the path name. It may be a NULL pointer or point to a string of up to five bytes to be used as the beginning of a file name.

The names generated are unique across processes and threads, and over time, so multiple threads should be able to each repeatedly call tempnam() and consistently obtain unique names.

This function is supported only in a POSIX program.

Returned value

If successful, tempnam() allocates space for the generated name, copies the name into it, and returns a pointer to the name.

If unsuccessful, tempnam() returns a NULL pointer and sets errno to one of the following values:

Error Code
  Description

ENAMETOOLONG
  The generated name exceeded the maximum allowable path name length.

ENOMEM
  Insufficient storage is available.

Related information

- “stdio.h” on page 68
- “fopen() — Open a file” on page 565
- “free() — Free a block of storage” on page 618
- “open() — Open a file” on page 1152

- "tmpfile() — Create temporary file" on page 1888
- "tmpnam() — Produce temporary file name" on page 1889
- "unlink() — Remove a directory entry" on page 1962

terminate() — Terminate after failures in C++ error handling

Standards

Standards / Extensions	C or C++	Dependencies
ANSI/ISO C++	C++ only	

Format

```
#include <exception>

void terminate(void);
```

General description

The terminate() function is called when the C++ error handling mechanism fails. If terminate() is called directly by the program, the terminate_handler is the one most recently set by a call to set_terminate(). If terminate() is called for any of several other reasons during evaluation of a throw expression, the terminate_handler is the one in effect immediately after evaluating the throw expression. If set_terminate() has not yet been called, then terminate() calls abort().

In a multithreaded environment, if a thread issues a throw, the stack is unwound until a matching catcher is found, up to and including the thread start routine. (The thread start routine is the function passed to pthread_create().) If the exception is not caught, then the terminate() function is called, which in turn defaults to calling abort(), which in turn causes a SIGABRT signal to be generated to the thread issuing the throw. If the SIGABRT signal is not caught, the process is terminated. You can replace the default terminate() behavior for all threads in the process by using the set_terminate() function. One possible use of set_terminate() is to call a function which issues a pthread_exit(). If this is done, a throw of a condition by a thread that is uncaught results in thread termination but not process termination.

Returned value

terminate() returns no values.

Refer to *z/OS XL C/C++ Language Reference* for more information about C++ exception handling including the terminate() function.

Related information

- "exception" on page 26
- "abort() — Stop a program" on page 101
- "set_terminate() — Register a function for terminate()" on page 1590
- "unexpected() — Handle exception not listed in exception specification" on page 1957

t_error() — Produce error message

Standards

Standards / Extensions	C or C++	Dependencies
XPG4.2	both	

Format

```
#define _XOPEN_SOURCE_EXTENDED 1
#include <xti.h>

int t_error(char *errmsg);
```

General description

Produces a language-dependent message on the standard error output which describes the last error encountered during a call to a transport function. The argument string *errmsg* is a user-supplied error message that gives context to the error.

The error message is written as follows: first (if *errmsg* is not a NULL pointer and the character pointed to by *errmsg* is not the NULL character) the string pointed to by *errmsg* followed by a colon and a space; then a standard error message string for the current error defined in *t_errno*. If *t_errno* has a value different from TSYSERR, the standard error message string is followed by a newline character. If, however, *t_errno* is equal to TSYSERR, the *t_errno* string is followed by the standard error message string for the current error defined in errno followed by a newline.

If the calling program is running in any one of the SAA, S370, C or POSIX locales, the error message string describing the value in *t_errno* is identical to the comments following the *t_errno* codes defined in xti.h. It is noteworthy that message numbers are not produced in this situation. The contents of the error message strings describing the value in errno are the same as those returned by the strerror(3C) function with an argument of errno.

The error number, *t_errno*, is only set when an error occurs and it is not cleared on successful calls.

Valid states: All - except for T_UNINIT

Returned value

No errors are defined for t_error().

Example

If a t_connect() function fails on transport endpoint *fd2* because a bad address was given, the following call might follow the failure:

```
t_error("t_connect failed on fd2");
```

The diagnostic message to be printed would look like:

```
t_connect failed on fd2: incorrect addr format
```

where *incorrect addr format* identifies the specific error that occurred, and *t_connect failed on fd2* tells the user which function failed on which transport endpoint.

Related information

- "xti.h" on page 87

tfind() — Binary tree find node

Standards

Standards / Extensions	C or C++	Dependencies
XPG4 XPG4.2 Single UNIX Specification, Version 3	both	

Format

```
#define _XOPEN_SOURCE
#include <search.h>

void *tfind(const void *key, void *const *rootp,
            int (*compar)(const void *, const void *));
```

General description

The tfind() function, like tsearch(), will search for a node in the tree, returning a pointer to it if found. However, if it is not found, the tfind() function will return a NULL pointer. The arguments for the tfind() function are the same as for the tsearch() function.

Comparisons are made with a user-supplied routine, the address of which is passed as the *compar* argument. This routine is called with two arguments, the pointers to the elements being compared. The user-supplied routine must return an integer less than, equal to or greater than 0, according to whether the first argument is to be considered less than, equal to or greater than the second argument. The comparison functions need not compare every byte, so arbitrary data may be contained in the elements in addition to the values being compared.

Threading Behavior: see "tsearch() — Binary tree search" on page 1920.

Special behavior for C++: Because C and C++ linkage conventions are incompatible, tfind() cannot receive a C++ function pointer as the comparator argument. If you attempt to pass a C++ function pointer to tfind(), the compiler will flag it as an error. You can pass a C or C++ function to tfind() by declaring it as extern "C".

Returned value

If the node is found, tfind() returns a pointer to it.

If unsuccessful, tfind() returns a NULL pointer.

If *rootp* is a NULL pointer on entry, tfind() returns a NULL pointer.

No errors are defined.

Related information

- "search.h" on page 62
- "bsearch() — Search arrays" on page 220
- "hsearch() — Search hash tables" on page 817

- “lsearch() — Linear search and update” on page 1019
- “tdelete() — Binary tree delete” on page 1865
- “tsearch() — Binary tree search” on page 1920
- “twalk() — Binary tree walk” on page 1935

t_free() — Free a library structure

Standards

Standards / Extensions	C or C++	Dependencies
XPG4.2	both	

Format

```
#define _XOPEN_SOURCE_EXTENDED 1
#include <xti.h>

int t_free(char *ptr, int struct_type);
```

General description

Frees memory previously allocated by t_alloc(). This function frees memory for the specified structure, and also frees memory for buffers referenced by the structure.

The argument *ptr* points to one of the seven structure types described for t_alloc() , and *struct_type* identifies the type of that structure which must be one of the following:

```
T_BIND       struct    t_bind
T_CALL       struct    t_call
T_OPTMGMT    struct    t_optmgmt
T_DIS        struct    t_discon
T_UNITDATA   struct    t_unitdata
T_UDERROR    struct    t_uderr
T_INFO       struct    t_info
```

where each of these structures is used as an argument to one or more transport functions.

t_free() checks the *addr*, *opt* and *udata* fields of the given structure (as appropriate) and frees the buffers pointed to by the *buf* field of the netbuf structure. If *buf* is a NULL pointer, t_free() does not attempt to free memory. After all buffers are freed, t_free() frees the memory associated with the structure pointed to by *ptr*.

Undefined results occur if *ptr* or any of the *buf* pointers points to a block of memory that was not previously allocated by t_alloc().

Valid states: All - except for T_UNINIT

Returned value

If successful, t_free() returns 0.

If unsuccessful, t_free() returns -1 and sets errno to one of the following values:

Error Code
Description

TNOSTRUCTYPE
Unsupported *struct_type* requested.

TPROTO
This error indicates that a communication problem has been detected between XTI and the transport provider for which there is no other suitable XTI (t_errno).

TSYSERR
A system error has occurred during execution of this function.

Related information

- "xti.h" on page 87
- "t_alloc() — Allocate a library structure" on page 1814

tgamma(), tgammaf(), tgammal() — Calculate gamma function

Standards

Standards / Extensions	C or C++	Dependencies
C99 Single UNIX Specification, Version 3 C++ TR1 C99	both	z/OS V1R5

Format

```
#define _ISOC99_SOURCE
#include <math.h>

double tgamma(double x);
float tgammaf(float x);
long double tgammal(long double x);
```

C++ TR1 C99

```
#define _TR1_C99
#include <math.h>

float tgamma(float x);
long double tgamma(long double x);
```

General description

The tgamma functions compute the gamma function of *x*. A domain error occurs if x is a negative integer or when x is zero and the result cannot be represented. A range error occurs if the magnitude of *x* is too large or too small.

Note: The following table shows the viable formats for these functions. See "IEEE binary floating-point" on page 94 for more information about IEEE Binary Floating-Point.

Function	Hex	IEEE
tgamma	X	X
tgammaf	X	X
tgammal	X	X

Special behavior for IEEE: A pole error occurs if the input of tgammaf is 0 and sets errno to ERANGE. A domain error occurs if the input of tgammaf is a negative integer and sets errno to EDOM.

Returned value

The tgamma functions return G(x).

tgammad32(), tgammad64(), tgammad128() - Calculate gamma function

Standards

Standards / Extensions	C or C++	Dependencies
C/C++ DFP	both	z/OS V1.10

Format

```
#define __STDC_WANT_DEC_FP__
#include <math.h>

_Decimal32  tgammad32(_Decimal32 x);
_Decimal64  tgammad64(_Decimal64 x);
_Decimal128 tgammad128(_Decimal128 x);
_Decimal32  tgamma(_Decimal32 x);     /* C++ only */
_Decimal64  tgamma(_Decimal64 x);     /* C++ only */
_Decimal128 tgamma(_Decimal128 x);    /* C++ only */
```

General description

The tgamma() functions compute the gamma function of *x*.

These functions work in IEEE decimal floating-point format. See “IEEE decimal floating-point” on page 95 for more information.

Note: To use IEEE decimal floating-point, the hardware must have the Decimal Floating-Point Facility installed.

Returned value

The tgamma functions return G(*x*).

A domain error occurs if *x* is a negative integer or when *x* is zero and the result cannot be represented. A range error occurs if the magnitude of *x* is too large or too small.

Example

CELEBT24

```
/* CELEBT24

   This example illustrates the tgammad128() function.

*/

#define  __STDC_WANT_DEC_FP__
#include <math.h>
#include <stdio.h>

int main(void)
```

```
{
   _Decimal128 x, y;

   x = 5.6DL;
   y = tgammad128(x);

   printf("tgammad128(%DDf) = %DDf\n", x, y);
}
```

Related information

- "math.h" on page 44

t_getinfo() — Get protocol-specific service information

Standards

Standards / Extensions	C or C++	Dependencies
XPG4.2	both	

Format

```
#define _XOPEN_SOURCE_EXTENDED 1
#include <xti.h>

int t_getinfo(int fd, struct t_info *info);
```

General description

Returns the current characteristics of the underlying transport protocol and/or transport connection associated with file descriptor *fd*. The *info* pointer is used to return the same information returned by t_open() , although not necessarily precisely the same values. This function enables a transport user to access this information during any phase of communication. This argument points to a t_info structure which contains the following members:

```
long addr;     /* max size of the transport protocol address      */
long options;  /* max number of bytes of protocol-specific options */
long tsdu;     /* max size of a transport service data unit (TSDU) */
long etsdu;    /* max size of an expedited transport service      */
               /* data unit (ETSDU)                               */
long connect;  /* max amount of data allowed on connection        */
               /* establishment functions                         */
long discon;   /* max amount of data allowed on t_snddis()        */
               /* and t_rcvdis() functions                        */
long servtype; /* sdis() functions                                */
long servtype; /* service type supported by the transport provider */
long flags;    /* other info about the transport provider         */
```

The fields take on the following values:

addr The size of a struct sockaddr_in is returned.

options
The value 304, which is the maximum number of bytes of options which can possibly be specified or requested, is returned.

tsdu Zero is returned, indicating that the TCP transport provider does not support the concept of TSDUs.

etsdu A value of -1 is returned, indicating that there is no limit on the size of an ETSDU.

connect
A value of -2 is returned, indicating that the TCP transport provider does not allow data to be sent with connection establishment functions.

discon A value of -2 is returned, indicating that the transport provider does not allow data to be sent with the abortive release functions.

servtype
T_COTS is always returned, since this is the only service type supported.

flags The T_SENDZERO bit is always set in this field, indicating that the TCP transport provider supports the sending of zero-length TSDUs.

If a transport user is concerned with protocol independence, the above sizes may be accessed to determine how large the buffers must be to hold each piece of information. Alternatively, the t_alloc() function may be used to allocate these buffers. An error results if a transport user exceeds the allowed data size on any function. The value of each field may change as a result of protocol option negotiation during connection establishment (the t_optmgmt() call has no affect on the values returned by t_getinfo()). These values will only change from the values presented to t_open() after the endpoint enters the T_DATAXFER state.

Valid states: All - except for T_UNINIT

Returned value

If successful, t_getinfo() returns 0.

If unsuccessful, t_getinfo() returns -1 and sets errno to one of the following values:

Error Code
Description

TBADF
The specified file descriptor does not refer to a transport endpoint.

TPROTO
This error indicates that a communication problem has been detected between XTI and the transport provider for which there is no other suitable XTI (*t_errno*).

TSYSERR
A system error has occurred during execution of this function.

Related information

- "xti.h" on page 87
- "t_alloc() — Allocate a library structure" on page 1814
- "t_open() — Establish a transport endpoint" on page 1899

t_getprotaddr() — Get the protocol addresses

Standards

Standards / Extensions	C or C++	Dependencies
XPG4.2	both	

Format

```
#define _XOPEN_SOURCE_EXTENDED 1
#include <xti.h>

int t_getprotaddr(int fd, struct t_bind *boundaddr,
                  struct t_bind *peeraddr);
```

General description

Returns local and remote protocol addresses currently associated with the transport endpoint specified by *fd*. In *boundaddr* and *peeraddr* the user specifies *maxlen*, which is the maximum size of the address buffer, and *buf* which points to the buffer where the address is to be placed. On return, the *buf* field of *boundaddr* points to the address, if any, currently bound to *fd*, and the *len* field specifies the length of the address. If the transport endpoint is in the T_UNBND state, zero is returned in the *len* field of *boundaddr*. The *buf* field of *peeraddr* points to the address, if any, currently connected to *fd*, and the *len* field specifies the length of the address. If the transport endpoint is not in the T_DATAXFER state, zero is returned in the *len* field of *peeraddr*.

Valid states: All - except for T_UNINIT

Returned value

If successful, t_getprotaddr() returns 0.

If unsuccessful, t_getprotaddr() returns -1 and sets errno to one of the following values:

Error Code
Description

TBADF
The specified file descriptor does not refer to a transport endpoint.

TBUFOVFLW
The number of bytes allocated for an incoming argument (maxlen) is greater than 0 but not sufficient to store the value of that argument.

TPROTO
This error indicates that a communication problem has been detected between XTI and the transport provider for which there is no other suitable XTI (t_errno).

TSYSERR
A system error has occurred during execution of this function.

Related information

- "xti.h" on page 87
- "t_bind() — Bind an address to a transport endpoint" on page 1821

t_getstate() — Get the current state

Standards

Standards / Extensions	C or C++	Dependencies
XPG4.2	both	

Format

```
#define _XOPEN_SOURCE_EXTENDED 1
#include <xti.h>

int t_getstate(int fd);
```

General description

Returns the current state of the provider associated with the transport endpoint specified by *fd*.

Valid states: All - except for T_UNINIT

Returned value

If successful, t_getstate() returns the state. The current state is one of the following:

T_DATAXFER
Data transfer.

T_IDLE
Idle.

T_INCON
Incoming connection pending.

T_OUTCON
Outgoing connection pending.

T_UNBND
Unbound.

If the provider is undergoing a state transition when t_getstate() is called, the function will fail.

If unsuccessful, t_getstate() returns -1 and sets errno to one of the following values:

Error Code
Description

TBADF
The specified file descriptor does not refer to a transport endpoint.

TPROTO
This error indicates that a communication problem has been detected between XTI and the transport provider for which there is no other suitable XTI (t_errno).

TSTATECHNG
The transport provider is undergoing a transient state change.

TSYSERR
A system error has occurred during execution of this function.

Related information

- “xti.h” on page 87
- “t_open() — Establish a transport endpoint” on page 1899

time(),time64() — Determine current UTC time

Standards

Standards / Extensions	C or C++	Dependencies
ISO C POSIX.1 XPG4 XPG4.2 C99 Single UNIX Specification, Version 3 Language Environment	both	

Format

```
#include <time.h>

time_t time(time_t *timeptr);
```

```
#define _LARGE_TIME_API
#include <time.h>

time64_t time64(time64_t *timer);
```

General description

Determines the current UTC time.

Returned value

The time() function returns the value of time in seconds since the Epoch.

Returns the current UTC time. The returned value is also stored in the location given by *timeptr*. If *timeptr* is NULL, the returned value is not stored. If the calendar time is not available, the value (time_t)-1 is returned.

time() returns the current value of the time-of-day (TOD) clock value obtained with the STCK instruction, rounded off to the nearest second, and normalized to the POSIX Epoch, January 1, 1970. The TOD clock value does not account for leap seconds. If you need more accuracy, use the STCK instruction or the TIME macro which does account for leap seconds using whatever value the system operator has entered for number of leap seconds in the CVT field. For more information about the STCK instruction, refer to *z/Architecture Principles of Operation*.

A returned value of 0 indicates the epoch, which was at the Coordinated Universal Time (UTC) of 00:00:00 on January 1, 1970.

The function time64() will behave exactly like time() except it will support calendar times beyond 03:14:07 UTC on January 19, 2038.

Example

CELEBT11

```
/* CELEBT11

   This example gets the time and assigns it to ltime, then uses
   the &ctime. function to convert the number of seconds to the
   current date and time.
   Finally, it prints a message giving the current time.
```

```
 */
#include <time.h>
#include <stdio.h>

int main(void)
{
   time_t ltime;

   time(&ltime);
   printf("The time is %s\n", ctime(&ltime));
}
```

Output

```
The time is Fri Jun 16 11:01:41 2006
```

Related information

- "time.h" on page 79
- "asctime(), asctime64() — Convert time to character string" on page 182
- "asctime_r(), asctime64_r() — Convert date and time to a character string" on page 184
- "clock() — Determine processor time" on page 285
- "ctime(), ctime64() — Convert time to character string" on page 359
- "ctime_r(), ctime64_r() — Convert time value to date and time character string" on page 362
- "gmtime(), gmtime64() — Convert time to broken-down UTC time" on page 810
- "gmtime_r(), gmtime64_r() — Convert a time value to broken-down UTC time" on page 812
- "localdtconv() — Date and time formatting convention inquiry" on page 981
- "localtime(), localtime64() — Convert time and correct for local time" on page 984
- "localtime_r(), localtime64_r() — Convert time value to broken-down local time" on page 986
- "mktime(), mktime64() — Convert local time" on page 1078
- "strftime() — Convert to formatted time" on page 1743
- "tzset() — Set the time zone" on page 1936

times() — Get process and child process times

Standards

Standards / Extensions	C or C++	Dependencies
POSIX.1 XPG4 XPG4.2 Single UNIX Specification, Version 3	both	

Format

```
#define _POSIX_SOURCE
#include <sys/times.h>

clock_t times(struct tms *buffer);
```

General description

Gets processor times of interest to a process.

`struct tms` **buffer*
: Points to a memory location where times() can store a structure of information describing processor time used by the current process and other related processes.

times() returns information in a `tms` structure, which has the following elements:

`clock_t tms_utime`

Amount of processor time used by instructions in the calling process.

Under z/OS UNIX, this does not include processor time spent running in the kernel. It does include any processor time accumulated for the address space before it became a z/OS UNIX process.

`clock_t tms_stime`

Amount of processor time used by the system.

Under z/OS UNIX, this value represents kernel busy time running on behalf of the calling process. It does not include processor time performing other MVS system functions on behalf of the process.

`clock_t tms_cutime`

The sum of tms_utime and tms_cutime values for all waited-for child processes which have terminated.

`clock_t tms_cstime`

The sum of tms_stime and tms_cstime values for all terminated child processes of the calling process.

`clock_t` is an integral type determined in the time.h header file. It measures times in terms of *clock ticks*. The number of clock ticks in a second (for your installation) can be found in `sysconf(_SC_CLK_TCK)`.

Times for a terminated child can be determined once wait() or waitpid() have reported the child's termination.

Pthreads can not be separately clocked by the times() function because they do not run in a separate process like forked children do.

Returned value

If successful, times() returns a value giving the elapsed time since the process was last invoked (for example, at system startup). If this time value cannot be determined, times() returns (clock_t)-1.

If unsuccessful, times() sets errno to one of the following values:

Error Code
: **Description**

ERANGE
: An overflow having occurred computing time values.

Example

CELEBT12

```
/* CELEBT12

   This example provides the amount of processor time
   used by instructions and the system for the parent and child
   processes.

 */
#define _POSIX_SOURCE
#include <sys/times.h>
#include <time.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <stdio.h>
#include <unistd.h>

main() {
  int status;
  long i, j;
  struct tms t;
  clock_t dub;

  int tics_per_second;

  tics_per_second = sysconf(_SC_CLK_TCK);

  if (fork() == 0) {
    for (i=0, j=0; i<1000000; i++)
       j += i;
    exit(0);
  }

  if (wait(&status) == -1)
    perror("wait() error");
  else if (!WIFEXITED(status))
    puts("Child did not exit successfully");
  else if ((dub = times(&t)) == -1)
    perror("times() error");
  else {
    printf("process was dubbed %f seconds ago.\n\n",
           ((double) dub)/tics_per_second);
    printf("            utime           stime\n");
    printf("parent:    %f        %f\n",
           ((double) t.tms_utime)/tics_per_second,
           ((double) t.tms_stime)/tics_per_second);
    printf("child:     %f        %f\n",
           ((double) t.tms_cutime)/tics_per_second,
           ((double) t.tms_cstime)/tics_per_second);
  }
}
```

Output

```
process was dubbed 1.600000 seconds ago.

            utime           stime
parent:    0.000000        0.020000
child:     0.320000        0.000000
```

Related information

- “sys/times.h” on page 75
- “time.h” on page 79
- “exec functions” on page 436
- “fork() — Create a new process” on page 571
- “time(),time64() — Determine current UTC time” on page 1878
- “wait() — Wait for a child process to end” on page 1992
- “waitpid() — Wait for a specific child process to end” on page 1997

tinit() — Attach and initialize MTF subtasks

Standards

Standards / Extensions	C or C++	Dependencies
Language Environment	C only	

Format

```
#include <mtf.h>

int tinit(const char *parallel_loadmod_name, int num_subtasks);
```

General description

Restriction: This function is not supported in AMODE 64.

Initializes the multitasking facility (MTF) environment under MVS.

To avoid infringing on the user's name space, this nonstandard function has two names. One name is prefixed with two underscore characters, and one name is not. The name without the prefix underscore characters is exposed only when you use LANGLVL(EXTENDED).

To use this function, you must either invoke the function using its external entry point name (that is, the name that begins with two underscore characters), or compile with LANGLVL(EXTENDED). When you use LANGLVL(EXTENDED) any relevant information in the header is also exposed.

tinit() is invoked from a main task to dynamically attach and initialize the number of subtasks specified by *num_subtasks*, where *num_subtasks* ranges from 1 to `MAXTASK` (which is defined in the header file mtf.h). After the subtasks have been attached and initialized by tinit(), each of the subtasks will be given a *task_id* and can then compute independent pieces of the program, in parallel with the main task, under the control of the tsched() and tsyncro() library functions.

The parallel load module (*parallel_loadmod_name*) must contain all parallel functions and reside in a partitioned data set named in the `STEPLIB DD` statement of the JCL that runs the program.

The tinit() function may be called by a main task only once before invoking tterm(). Invocations of tinit() after the first are one are terminated with a returned value indicating that MTF is already active.

After tterm() has been called to terminate and remove the MTF environment, or after an abend, tinit() can be called again to create a new MTF environment. The new initialization is independent of the old one and may provide a different number of tasks and/or a different parallel load module.

If tinit() is called from a parallel function, tinit() will be terminated with a returned value indicating that MTF calls cannot be issued from a parallel function.

If tinit() is called by a program running under IMS, CICS, or DB2, the request will not be processed and the returned value will indicate that MTF calls are not supported under these systems.

Returned value

If the subtasks have been attached successfully and the MTF environment created, tinit() returns MTF_OK.

If unsuccessful, tinit() sets errno to one of the following values:

Error Code
Description

EACTIVE
MTF has already been initialized and is active.

EAUTOALC
Automatic allocation of standard stream DD has failed.

EMODFIND
Parallel load module was not found.

EMODFMT
Parallel load module has an invalid format.

EMODREAD
Parallel load module was not successfully read.

ENAME2LNG
Parallel load module name is longer than 8 characters.

ENOMEM
There was insufficient storage for MTF-internal areas.

ESUBCALL
The MTF call was issued from a subtask.

ETASKABND
One or more subtasks have terminated abnormally.

ETASKFAIL
The attempt to attach task(s) failed.

ETASKNUM
Number of tasks specified is invalid (<1 or >MAXTASK).

EWRONGOS
MTF is not supported under IMS, CICS or DB2*.

Note: These values are macros and can be found in the mtf.h header file.

Related information

- "mtf.h" on page 49
- "tsched() — Schedule MTF subtask" on page 1918
- "tsyncro() — Wait for MTF subtask termination" on page 1928
- "tterm() — Terminate MTF subtasks" on page 1929

t_listen() — Listen for a connect indication

Standards

Standards / Extensions	C or C++	Dependencies
XPG4.2	both	

Format

```
#define _XOPEN_SOURCE_EXTENDED 1
#include <xti.h>

int t_listen(int fd, struct t_call *call);
```

General description

Listens for a connect request from a calling transport user. The argument *fd* identifies the local transport endpoint where connect indications arrive, and on return, *call* contains information describing the connect indication. The parameter *call* points to a t_call structure which contains the following members:

```
struct netbuf   addr;
struct netbuf   opt;
struct netbuf   udata;
int             sequence;
```

In *call*, *addr* returns the protocol address of the calling transport user. This address is in a format usable in future calls to t_connect(). However, t_connect() may fail for other reasons; For example TADDRBUSY. *opt* returns options associated with the connect request. *udata* is meaningless because transmission of user data is not supported across a connect request. *sequence* is a number that uniquely identifies the returned connect indication. The value of sequence enables the user to listen for multiple connect indications before responding to any of them.

Since this function returns values for the *addr*, *opt* and *udata* fields of call, the *maxlen* field of each must be set before issuing the t_listen() to indicate the maximum size of the buffer for each.

By default, t_listen() executes in synchronous mode and waits for a connect indication to arrive before returning to the user. However, if O_NONBLOCK is set using t_open() or fcntl(), t_listen() executes asynchronously, reducing to a poll for existing connect indications. If none are available, it returns -1 and sets *t_errno* to TNODATA.

Valid states: T_IDLE, T_INCON

Returned value

If successful, t_listen() returns 0. The TCP transport provider does not differentiate between a connect indication and the connection itself. A successful return of t_listen() indicates an existing connection.

If unsuccessful, t_listen() returns -1 and sets errno to one of the following values:

Error Code
    Description

TBADF
    The specified file descriptor does not refer to a transport endpoint.

TBADQLEN
    The argument *qlen* of the endpoint referenced by *fd* is zero.

TBUFOVFLW
    The number of bytes allocated for an incoming argument (*maxlen*) is greater than 0 but not sufficient to store the value of that argument. The provider's state, as seen by the user, changes to T_INCON, and the connect

indication information to be returned in call is discarded. The value of sequence returned can be used to do a t_snddis() .

TLOOK
An asynchronous event has occurred on this transport endpoint and requires immediate attention.

TNODATA
O_NONBLOCK was set, but no connect indications had been queued.

TNOTSUPPORT
This function is not supported by the underlying transport provider.

TOUTSTATE
The function was issued in the wrong sequence on the transport endpoint referenced by *fd*.

TPROTO
This error indicates that a communication problem has been detected between XTI and the transport provider for which there is no other suitable XTI (*t_errno*).

TQFULL
The maximum number of outstanding indications has been reached for the endpoint referenced by *fd*.

TSYSERR
A system error has occurred during execution of this function.

Related information

- “xti.h” on page 87
- “fcntl() — Control open file descriptors” on page 474
- “t_accept() — Accept a connect request” on page 1810
- “t_alloc() — Allocate a library structure” on page 1814
- “t_bind() — Bind an address to a transport endpoint” on page 1821
- “t_connect() — Establish a connection with another transport user” on page 1838
- “t_open() — Establish a transport endpoint” on page 1899
- “t_optmgmt() — Manage options for a transport endpoint” on page 1901
- “t_rcvconnect() — Receive the confirmation from a connect request” on page 1910

t_look() — Look at the current event on a transport endpoint

Standards

Standards / Extensions	C or C++	Dependencies
XPG4.2	both	

Format

```
#define _XOPEN_SOURCE_EXTENDED 1
#include <xti.h>

int t_look(int fd);
```

General description

Returns the current event on the transport endpoint specified by *fd*. This function enables a transport provider to notify a transport user of an asynchronous event when the user is calling functions in synchronous mode. Certain events require immediate notification of the user and are indicated by a specific error, TLOOK, on the current or next function to be executed. This function also enables a transport user to poll a transport endpoint periodically for asynchronous events.

Additional functionality for handling events is provided through select and poll.

Valid states: All - except for T_UNINIT

The following list describes the asynchronous events which cause an XTI call to return with a TLOOK error:

t_accept()
T_DISCONNECT, T_LISTEN

t_connect()
T_DISCONNECT, T_LISTEN

This occurs only when a t_connect is done on an endpoint which has been bound with a *qlen* > 0 and for which a connect indication is pending.

t_listen()
T_DISCONNECT

This event indicates a disconnect on an outstanding connect indication.

t_rcv() T_DISCONNECT

This occurs only when all pending data has been read.

t_rcvconnect()
T_DISCONNECT

t_rcvudata()
T_UDERR

t_snd()
T_DISCONNECT

t_sndudata()
T_UDERR

t_unbind()
T_LISTEN, T_DATA

T_DATA may only occur for the connectionless mode.

t_snddis()
T_DISCONNECT

Once a TLOOK error has been received on a transport endpoint using an XTI function, subsequent calls to that and other XTI functions, to which the same TLOOK error applies, will continue to return TLOOK until the event is consumed. An event causing the TLOOK error can be determined by calling t_look() and then can be consumed by calling the corresponding consuming XTI function as defined in Table 59.

Table 59. Events and t_look()

Event	Cleared on t_look()?	Consuming XTI functions
T_LISTEN	No	t_listen()

Table 59. Events and t_look() (continued)

Event	Cleared on t_look()?	Consuming XTI functions
T_CONNECT	No	t_{rcv}connect() In the case of the t_connect() function the T_CONNECT event is both generated and consumed by the execution of the function and is therefore not visible to the application.
T_DATA	No	t_rcv()
T_EXDATA	No	t_rcv()
T_DISCONNECT	No	t_rcvdis()
T_GODATA	Yes	t_snd()
T_GOEXDATA	Yes	t_snd()

Returned value

If successful, t_look() returns a value that indicates which of the allowable events has occurred, or returns 0 if no event exists. One of the following events is returned:

T_CONNECT
Connect confirmation received.

T_DATA
Normal data received.

T_DISCONNECT
Disconnect received.

T_EXDATA
Expedited data received.

T_GODATA
Flow control restrictions on normal data flow that led to a TFLOW error have been lifted. Normal data may be sent again.

T_GOEXDATA
Flow control restrictions on expedited data flow that led to a TFLOW error have been lifted. Expedited data may be sent again.

T_LISTEN
Connection indication received.

If unsuccessful, t_look() returns -1 and sets errno to one of the following values:

Error Code
Description

TBADF
The specified file descriptor does not refer to a transport endpoint.

TPROTO
This error indicates that a communication problem has been detected between XTI and the transport provider for which there is no other suitable XTI (*t_errno*).

TSYSERR
A system error has occurred during execution of this function.

Related information

- "xti.h" on page 87
- "t_open() — Establish a transport endpoint" on page 1899
- "t_snd() — Send data or expedited data over a connection" on page 1921
- "t_sndudata() — Send a data unit" on page 1925

tmpfile() — Create temporary file

Standards

Standards / Extensions	C or C++	Dependencies
ISO C POSIX.1 XPG4 XPG4.2 C99 Single UNIX Specification, Version 3	both	

Format

```
#include <stdio.h>

FILE *tmpfile(void);
```

General description

Creates a temporary binary file. It opens the temporary file in wb+ mode. The file is automatically removed when it is closed or when the program is terminated.

Note: When the tmpfile() function is issued from multiple tasks within one address space, the temporary file names may not be unique. The execution of the tmpfile() function concurrently within one address space will result in errors. For example, an open will fail because the file is already open.

Returned value

If successful, tmpfile() returns a pointer to the stream associated with the file created.

If tmpfile() cannot open the file, it returns a NULL pointer. On normal termination (exit()), these temporary files are removed. On abnormal termination, an effort is made to remove these files.

Returned value for POSIX C

When the calling application is a z/OS UNIX program, the temporary file is created in the hierarchical file system. The file is created in the directory referred to by the TMPDIR environment variable, or '/tmp' if TMPDIR is not defined.

Special behavior for XPG4: The following are the possible values of errno:

Error Code
  Description

EINTR
  A signal was caught during tmpfile().

EMFILE
: **OPEN_MAX** file descriptors are currently open in the calling process.

{FOPEN_MAX} streams are currently open in the calling process.

ENFILE
: The maximum allowable number of files is currently open in the system.

ENOMEM
: Insufficient storage space is available.

ENOSPC
: The directory or file system which would contain the new file cannot be expanded.

Example

CELEBT13

```
/* CELEBT13

   This example creates a temporary file and if successful,
   writes tmpstring to it.
   At program termination, the file is removed.

 */
#include <stdio.h>

int main(void) {
   FILE *stream;
   char tmpstring[ ] = "This string will be written";

{
   if((stream = tmpfile( )) == NULL)
      printf("Cannot make a temporary file\n");
   else
      fprintf(stream, "%s", tmpstring);
}
}
```

Related information

- "stdio.h" on page 68
- "fopen() — Open a file" on page 565

tmpnam() — Produce temporary file name

Standards

Standards / Extensions	C or C++	Dependencies
ISO C POSIX.1 XPG4 XPG4.2 C99 Single UNIX Specification, Version 3	both	

Format

```
#include <stdio.h>

char *tmpnam(char *string);
```

General description

Produces a valid file name that is not the same as the name of any existing file. It stores this name in *string*. If *string* is a NULL pointer, tmpnam() leaves the result in an internal static buffer. Any subsequent calls may modify this object. If *string* is not a NULL pointer, it must point to an array of at least `L_tmpnam` bytes. The value of `L_tmpnam` is defined in the stdio.h header file.

Returned value

If string is a NULL pointer, tmpnam() returns the pointer to the internal static object in which the generated unique name is placed. Otherwise, if *string* is not a NULL pointer, it returns the value of *string*. The tmpnam() function produces a different name each time it is called within a module up to at least `TMP_MAX` names. Files created using names returned by tmpnam() are not automatically discarded at the end of the program.

Returned value for POSIX C

The tmpnam() function behaves differently when a program is running with POSIX(ON).

When the __POSIX_TMPNAM environment variable is unset or is not set to *NO*, the file name returned is a unique file name in the UNIX file system. The directory component of the file name will be the value of the *TMPDIR* environment variable, or '/tmp' if TMPDIR is not defined.

When the __POSIX_TMPNAM environment variable is set to *NO*, the file name returned is as if the calling application were running POSIX(OFF).

Example

CELEBT14

```
/* CELEBT14

   This example calls &tmpnam. to produce a valid file name.

 */
#include <stdio.h>

int main(void)
{
   char *name1;
   if ((name1 = tmpnam(NULL)) !=NULL)
      printf("%s can be used as a file name.\n", name1);
   else printf("Cannot create a unique file name\n");
}
```

Related information

- “stdio.h” on page 68
- “fopen() — Open a file” on page 565

toascii() — Translate integer to a 7-bit ASCII character

Standards

Standards / Extensions	C or C++	Dependencies
XPG4 XPG4.2 Single UNIX Specification, Version 3	both	POSIX(ON)

Format

_XOPEN_SOURCE

```
#define _XOPEN_SOURCE
#include <ctype.h>

int toascii(int c);
```

_ALL_SOURCE

```
#define _ALL_SOURCE
#include <ctype.h>

int toascii(int c);
```

General description

Special behavior for _XOPEN_SOURCE: The toascii() function converts its argument to a 7-bit US-ASCII character code.

The toascii() function is not intended to be used to convert EBCDIC characters to ASCII, attempts to use it in this manner will not function as expected.

Special behavior for _ALL_SOURCE: The toascii() function assumes *c* modulo 256 is a single-byte EBCDIC encoding for a Latin 1 character, <input-character>, in the current locale. Then, toascii() determines to what character, <output-character>, toascii() would map <input-character> in an ASCII locale (for example, on an *IX system) and returns the EBCDIC encoding for <output-character> in the current locale.

For example, if the program invoking toascii() was compiled with _ALL_SOURCE defined and if the value *c* input to toascii() modulo 256 is the EBCDIC encoding for <international-currency-symbol> in the current locale, toascii() returns the EBCDIC encoding for <dollar> in the current locale because toascii() maps <international-currency-symbol> to <dollar> on ASCII platforms.

Returned value

EBCDIC and ASCII encodings and <input-character> to <output-character> mapping performed by toascii() in an IBM-1047 locale are as follows:

```
/*********************************************************************/
/*                                                                   */
/*  IBM-1047 toascii table  (sorted by ebcdic)                       */
/*                                                                   */
/*    For ISO-8859 character encoding toascii(ch) returns ch for     */
/*    values of ch less than 128 and ch-128 for values between       */
/*    128 and 255, inclusive. Table below shows corresponding        */
/*    toascii(ch) equivalence for IBM-1047 character encoding        */
/*    of the Latin character set.                                    */
/*                                                                   */
/*                         IBM 1047               ISO 8859-1         */
/*    Character            Encoding               Encoding           */
/* (Symbolic Name)         (Hexadecimal)          (Hexadecimal)      */
/*                                                                   */
/*   ch     toascii(ch)    ch  toascii(ch)        ch  toascii(ch)    */
/*                                                                   */
/*  <NUL>      <NUL>       00      00             00      00         */
/*  <SOH>      <SOH>       01      01             01      01         */
/*  <STX>      <STX>       02      02             02      02         */
/*  <ETX>      <ETX>       03      03             03      03         */
/*  <SEL>      <IFS/IS4>   04      1C             9C      1C         */
/*  <HT>       <HT>        05      05             09      09         */
/*  <RNL>      <ACK>       06      2E             86      06         */
/*  <DEL>      <DEL>       07      07             7F      7F         */
/*  <GE>       <ETB>       08      26             97      17         */
/*  <SPS>      <CR>        09      0D             8D      0D         */
/*  <RPT>      <SO>        0A      0E             8E      0E         */
/*  <VT>       <VT>        0B      OB             0B      0B         */
/*  <FF>       <FF>        0C      0C             0C      0C         */
/*  <CR>       <CR>        0D      0D             0D      OD         */
/*  <SO>       <SO>        0E      0E             0E      OE         */
/*  <SI>       <SI>        0F      0F             0F      0F         */
/*  <DLE>      <DLE>       10      10             10      10         */
/*  <DC1>      <DC1>       11      11             11      11         */
/*  <DC2>      <DC2>       12      12             12      12         */
/*  <DC3>      <DC3>       13      13             13      13         */
/*  <RES/ENP>  <IGS/IS3>   14      1D             9D      1D         */
/*  <NL>       <NL>        15      15             0A      0A         */
/*  <BS>       <BS>        16      16             08      08         */
/*  <POC>      <BEL>       17      2F             87      07         */
/*  <CAN>      <CAN>       18      18             18      18         */
/*  <EM>       <EM>        19      19             19      19         */
/*  <UBS>      <DC2>       1A      12             92      12         */
/*  <CU1>      <SI>        1B      0F             8F      0F         */
/*  <IFS/IS4>  <IFS/IS4>   1C      1C             1C      1C         */
/*  <IGS/IS3>  <IGS/IS3>   1D      1D             1D      1D         */
/*  <IRS/IS2>  <IRS/IS2>   1E      1E             1E      1E         */
/*  <IUS/IS1>  <IUS/IS1>   1F      1F             1F      1F         */
/*  <DS>       <NUL>       20      00             80      00         */
/*  <SOS>      <SOH>       21      01             81      01         */
/*  <FS>       <STX>       22      02             82      02         */
/*  <WUS>      <ETX>       23      03             83      03         */
/*  <BYP/INP>  <EOT>       24      37             84      04         */
/*  <LF>       <ENQ>       25      2D             85      05         */
/*  <ETB>      <ETB>       26      26             17      17         */
/*  <ESC>      <ESC>       27      27             1B      1B         */
/*  <SA>       <BS>        28      16             88      08         */
/*  <SFE>      <HT>        29      05             89      09         */
/*  <SM/SW>    <NL>        2A      15             8A      0A         */
/*  <CSP>      <VT>        2B      0B             8B      0B         */
/*  <MFA>      <FF>        2C      0C             8C      0C         */
/*  <ENQ>      <ENQ>       2D      2D             05      05         */
/*  <ACK>      <ACK>       2E      2E             06      06         */
/*  <BEL>      <BEL>       2F      2F             07      07         */
/*  (reserved) <DLE>       30      10             90      10         */
/*  (reserved) <DC1>       31      11             91      11         */
/*  <SYN>      <SYN>       32      32             16      16         */
/*  <IR>       <DC3>       33      13             93      13         */
/*  <PP>       <DC4>       34      3C             94      14         */
/*  <TRN>      <NAK>       35      3D             95      15         */
/*  <NBS>      <SYN>       36      32             96      16         */
/*  <EOT>      <EOT>       37      37             04      04         */
/*  <SBS>      <CAN>       38      18             98      18         */
/*  <IT>       <EM>        39      19             99      19         */
/*  <RFF>      <SUB>       3A      3F             9A      1A         */
/*  <CU3>      <ESC>       3B      27             9B      1B         */
/*  <DC4>      <DC4>       3C      3C             14      14         */
/*  <NAK>      <NAK>       3D      3D             15      15         */
/*  (reserved) <IRS/IS2>   3E      1E             9E      1E         */
/*  <SUB>      <SUB>       3F      3F             1A      1A         */
/*                                                                   */
/*  <space>    <space>     40      40             20      20         */
/*  <nobrk-sp> <space>     41      40             A0      20         */
/*  <a-circum> <b>         42      82             E2      62         */
/*  <a-diaere> <d>         43      84             E4      64         */
/*  <a-grave>  <grave>     44      79             E0      60         */
/*  <a-acute>  <a>         45      81             E1      61         */
/*  <a-tilde>  <c>         46      83             E3      63         */
/*  <a-ring>   <e>         47      85             E5      65         */
/*  <c-cedilla><g>         48      87             E7      67         */
/*  <n-tilde>  <q>         49      98             F1      71         */
/*  <cent-sign><quote>     4A      7F             A2      22         */
/*  <period>   <period>    4B      4B             2E      2E         */
/*  <lt>       <lt>        4C      4C             3C      3C         */
/*  <l-paren>  <l-paren>   4D      4D             28      28         */
/*  <plus>     <plus>      4E      4E             2B      2B         */
/*  <ver-line> <ver-line>  4F      4F             7C      7C         */
/*  <ampersand><ampersand> 50      50             26      26         */
/*  <e-acute>  <i>         51      89             E9      69         */
/*  <e-circum> <j>         52      91             EA      6A         */
/*  <e-diaere> <k>         53      92             EB      6B         */
/*  <e-grave>  <h>         54      88             E8      68         */
/*  <i-acute>  <m>         55      94             ED      6D         */
/*  <i-circum> <n>         56      95             EE      6E         */
/*  <i-diaere> <o>         57      96             EF      6F         */
/*  <i-grave>  <l>         58      93             EC      6C         */
/*  <s-sharp>  <underscr>  59      6D             DF      5F         */
/*  <exclama>  <exclama>   5A      5A             21      21         */
/*  <dollar>   <dollar>    5B      5B             24      24         */
/*  <asterisk> <asterisk>  5C      5C             2A      2A         */
/*  <r-paren>  <r-paren>   5D      5D             29      29         */
/*  <semicolon><semicolon> 5E      5E             3B      3B         */
/*  <circum>   <circum>    5F      5F             5E      5E         */
/*  <hyphen>   <hyphen>    60      60             2D      2D         */
/*  <slash>    <slash>     61      61             2F      2F         */
/*  <A-circum> <B>         62      C2             C2      42         */
/*  <A-diaere> <D>         63      C4             C4      44         */
/*  <A-grave>  <at>        64      7C             C0      40         */
/*  <A-acute>  <A>         65      C1             C1      41         */
/*  <A-tilde>  <C>         66      C3             C3      43         */
/*  <A-ring>   <E>         67      C5             C5      45         */
/*  <C-cedilla><G>         68      C7             C7      47         */
/*  <N-tilde>  <Q>         69      D8             D1      51         */
/*  <brok-bar> <ampersand> 6A      50             A6      26         */
/*  <comma>    <comma>     6B      6B             2C      2C         */
/*  <percent>  <percent>   6C      6C             25      25         */
/*  <underscr> <underscr>  6D      6D             5F      5F         */
/*  <gt>       <gt>        6E      6E             3E      3E         */
/*  <question> <question>  6F      6F             3F      3F         */
/*  <o-stroke> <x>         70      A7             F8      78         */
/*  <E-acute>  <I>         71      C9             C9      49         */
/*  <E-circum> <J>         72      D1             CA      4A         */
/*  <E-diaere> <K>         73      D2             CB      4B         */
/*  <E-grave>  <H>         74      C8             C8      48         */
/*  <I-acute>  <M>         75      D4             CD      4D         */
/*  <I-circum> <N>         76      D5             CE      4E         */
/*  <I-diaere> <O>         77      D6             CF      4F         */
/*  <I-grave>  <L>         78      D3             CC      4C         */
/*  <grave>    <grave>     79      79             60      60         */
/*  <colon>    <colon>     7A      7A             3A      3A         */
/*  <num-sign> <num-sign>  7B      7B             23      23         */
/*  <at>       <at>        7C      7C             40      40         */
/*  <apostro>  <apostro>   7D      7D             27      27         */
/*  <eq>       <eq>        7E      7E             3D      3D         */
/*  <quote>    <quote>     7F      7F             22      22         */
/*  <O-stroke> <X>         80      E7             D8      58         */
/*  <a>        <a>         81      81             61      61         */
/*  <b>        <b>         82      82             62      62         */
/*  <c>        <c>         83      83             63      63         */
/*  <d>        <d>         84      84             64      64         */
/*  <e>        <e>         85      85             65      65         */
/*  <f>        <f>         86      86             66      66         */
/*  <g>        <g>         87      87             67      67         */
/*  <h>        <h>         88      88             68      68         */
/*  <i>        <i>         89      89             69      69         */
/*  <l-guille> <plus>      8A      4E             AB      2B         */
/*  <r-guille> <semicolon> 8B      5E             BB      3B         */
/*  <eth>      <p>         8C      97             F0      70         */
/*  <y-acute>  <r-brace>   8D      D0             FD      7D         */
/*  <thorn>    <tilde>     8E      A1             FE      7E         */
/*  <plusminus><one>       8F      F1             B1      31         */
/*  <degree>   <zero>      90      F0             B0      30         */
/*  <j>        <j>         91      91             6A      6A         */
/*  <k>        <k>         92      92             6B      6B         */
/*  <l>        <l>         93      93             6C      6C         */
/*  <m>        <m>         94      94             6D      6D         */
/*  <n>        <n>         95      95             6E      6E         */
/*  <o>        <o>         96      96             6F      6F         */
/*  <p>        <p>         97      97             70      70         */
/*  <q>        <q>         98      98             71      71         */
/*  <r>        <r>         99      99             72      72         */
/*  <fem-ind>  <asterisk>  9A      5C             AA      2A         */
/*  <mas-ind>  <colon>     9B      7A             BA      3A         */
/*  <ae>       <f>         9C      86             E6      66         */
/*  <cedilla>  <eight>     9D      F8             B8      38         */
/*  <AE>       <F>         9E      C6             C6      46         */
/*  <cur-sign> <dollar>    9F      5B             A4      24         */
/*  <mu>       <five>      A0      F5             B5      35         */
/*  <tilde>    <tilde>     A1      A1             7E      7E         */
/*  <s>        <s>         A2      A2             73      73         */
/*  <t>        <t>         A3      A3             74      74         */
/*  <u>        <u>         A4      A4             75      75         */
/*  <v>        <v>         A5      A5             76      76         */
/*  <w>        <w>         A6      A6             77      77         */
/*  <x>        <x>         A7      A7             78      78         */
/*  <y>        <y>         A8      A8             79      79         */
/*  <z>        <z>         A9      A9             7A      7A         */
/*  <inv-excl> <exclama>   AA      5A             A1      21         */
/*  <inv-ques> <question>  AB      6F             BF      3F         */
/*  <Eth>      <P>         AC      D7             D0      50         */
/*  <l-brk>    <l-brk>     AD      AD             5B      5B         */
/*  <Thorn>    <circum>    AE      5F             DE      5E         */
/*  <register> <period>    AF      4B             AE      2E         */
/*  <not-sign> <comma>     B0      6B             AC      2C         */
/*  <pound>    <num-sign>  B1      7B             A3      23         */
/*  <yen>      <percent>   B2      6C             A5      25         */
/*  <mid-dot>  <seven>     B3      F7             B7      37         */
/*  <copyright><r-paren>   B4      5D             A9      29         */
/*  <section>  <apostro>   B5      7D             A7      27         */
/*  <paragraph><six>       B6      F6             B6      36         */
/*  <1/4>      <lt>        B7      4C             BC      3C         */
/*  <1/2>      <eq>        B8      7E             BD      3D         */
/*  <3/4>      <gt>        B9      6E             BE      3E         */
/*  <Y acute>  <r-brk>     BA      BD             DD      5D         */
/*  <diaeresis><l-paren>   BB      4D             A8      28         */
/*  <macron>   <slash>     BC      61             AF      2F         */
/*  <r-brk>    <r-brk>     BD      BD             5D      5D         */
/*  <acute>    <four>      BE      F4             B4      34         */
/*  <multiply> <W>         BF      E6             D7      57         */
/*  <l-brace>  <l-brace>   C0      C0             7B      7B         */
/*  <A>        <A>         C1      C1             41      41          */
/*  <B>        <B>         C2      C2             42      42          */
/*  <C>        <C>         C3      C3             43      43          */
/*  <D>        <D>         C4      C4             44      44          */
/*  <E>        <E>         C5      C5             45      45          */
/*  <F>        <F>         C6      C6             46      46          */
/*  <G>        <G>         C7      C7             47      47          */
/*  <H>        <H>         C8      C8             48      48          */
/*  <I>        <I>         C9      C9             49      49          */
/*  <soft-hyp> <hyphen>    CA      60             AD      2D          */
/*  <o-circum> <t>         CB      A3             F4      74          */
/*  <o-diaere> <v>         CC      A5             F6      76          */
/*  <o-grave>  <r>         CD      99             F2      72          */
/*  <o-acute>  <s>         CE      A2             F3      73          */
/*  <o-tilde>  <u>         CF      A4             F5      75          */
/*  <r-brace>  <r-brace>   D0      D0             7D      7D          */
/*  <J>        <J>         D1      D1             4A      4A          */
/*  <K>        <K>         D2      D2             4B      4B          */
/*  <L>        <L>         D3      D3             4C      4C          */
/*  <M>        <M>         D4      D4             4D      4D          */
/*  <N>        <N>         D5      D5             4E      4E          */
/*  <O>        <O>         D6      D6             4F      4F          */
/*  <P>        <P>         D7      D7             50      50          */
/*  <Q>        <Q>         D8      D8             51      51          */
/*  <R>        <R>         D9      D9             52      52          */
/*  <super-1>  <nine>      DA      F9             B9      39          */
/*  <u-circum> <l-brace>   DB      C0             FB      7B          */
/*  <u-diaere> <ver_line>  DC      4F             FC      7C          */
/*  <u-grave>  <y>         DD      A8             F9      79          */
/*  <u-acute>  <z>         DE      A9             FA      7A          */
/*  <y-diaere> <DEL>       DF      07             FF      7F          */
/*  <backslash><backslash> E0      E0             5C      5C          */
/*  <division> <w>         E1      A6             F7      77          */
/*  <S>        <S>         E2      E2             53      53          */
/*  <T>        <T>         E3      E3             54      54          */
/*  <U>        <U>         E4      E4             55      55          */
/*  <V>        <V>         E5      E5             56      56          */
/*  <W>        <W>         E6      E6             57      57          */
/*  <X>        <X>         E7      E7             58      58          */
/*  <Y>        <Y>         E8      E8             59      59          */
/*  <Z>        <Z>         E9      E9             5A      5A          */
/*  <super-2>  <two>       EA      F2             B2      32          */
/*  <O-circum> <T>         EB      E3             D4      54          */
/*  <O-diaere> <V>         EC      E5             D6      56          */
/*  <O-grave>  <R>         ED      D9             D2      52          */
/*  <O-acute>  <S>         EE      E2             D3      53          */
/*  <O-tilde>  <U>         EF      E4             D5      55          */
/*  <zero>     <zero>      F0      F0             30      30          */
/*  <one>      <one>       F1      F1             31      31          */
/*  <two>      <two>       F2      F2             32      32          */
/*  <three>    <three>     F3      F3             33      33          */
/*  <four>     <four>      F4      F4             34      34          */
/*  <five>     <five>      F5      F5             35      35          */
/*  <six>      <six>       F6      F6             36      36          */
/*  <seven>    <seven>     F7      F7             37      37          */
/*  <eight>    <eight>     F8      F8             38      38          */
/*  <nine>     <nine>      F9      F9             39      39          */
/*  <super-3>  <three>     FA      F3             B3      33          */
/*  <U-circum> <l-brk>     FB      AD             DB      5B          */
/*  <U-diaere> <backslash> FC      E0             DC      5C          */
/*  <U-grave>  <Y>         FD      E8             D9      59          */
/*  <U-acute>  <Z>         FE      E9             DA      5A          */
/*  <EO>       <IUS/IS1>   FF      1F             9F      1F          */
/*                                                                    */
/********************************************************************/
```

Special behavior for _XOPEN_SOURCE: The toascii() function returns the value (c & 0x7f).

Special behavior for _ALL_SOURCE: If the current locale is not a single-byte locale (that is, mb_cur_max > 1), toascii() sets errno to **ENOSYS** and returns -1. Otherwise, toascii() assumes *c* modulo 256 is the encoding of a Latin 1 character, <input_character>, in the current locale and returns the EBCDIC encoding of the same or another Latin 1 character, <output-character>, in the current locale; where <output-character> corresponds to the character to which toascii() would map <input-character> on an ASCII platform.

Related information

- “ctype.h” on page 21
- “isascii() — Test for 7-bit US-ASCII character” on page 898

__toCcsid() — Convert codeset name to coded character set ID

Standards

Standards / Extensions	C or C++	Dependencies
z/OS UNIX	both	z/OS V1R2

Format

```
#include <_Ccsid.h>

__ccsid_t __toCcsid(char *codesetName);
```

General description

The __toCcsid() function returns the coded character set ID corresponding to the provided *codesetName* argument.

Returned value

If successful, __toCcsid() returns the corresponding CCSID for *codesetName*, if one exists. The returned *__ccsid_t* type is defined in <_Ccsid.h> as an unsigned short.

If unsuccessful, __toCcsid() returns 0 and sets errno to one of the following values:

Error Code
Description

EINVAL
The length of the *codesetName* argument is greater than _CSNAME_LEN_MAX as defined in header <_Ccsid.h>.

Related information

- “_Ccsid.h” on page 17
- “__CcsidType() — Return coded character set ID type” on page 246
- “__CSNameType() — Return codeset name type” on page 350
- “__toCSName() — Convert coded character set ID to codeset name” on page 1897

__toCSName() — Convert coded character set ID to codeset name

Standards

Standards / Extensions	C or C++	Dependencies
z/OS UNIX	both	z/OS V1R2

Format

```
#include <_Ccsid.h>

int __toCSName(__ccsid_t Ccsid, char *codesetName);
```

General description

The __toCSName() function returns the codeset name, corresponding to coded character codeset ID, *Ccsid*, in *codesetName*.

Returned value

If successful, __toCSName() returns 0 and stores the codeset name in *codesetName*, when a corresponding codeset name exists for *Ccsid*.

If unsuccessful, __toCSName() returns -1.

If __toCSName() returns -1 for a reason other than no corresponding codeset name exists, it sets errno to one of the following values:

Error Code
Description

EINVAL
The corresponding *codesetName* length is greater than _CSNAME_LEN_MAX as defined in header <_Ccsid.h>.

Related information

- "_Ccsid.h" on page 17
- "__CcsidType() — Return coded character set ID type" on page 246
- "__CSNameType() — Return codeset name type" on page 350
- "__toCcsid() — Convert codeset name to coded character set ID" on page 1896

tolower(), toupper() — Convert character case

Standards

Standards / Extensions	C or C++	Dependencies
ISO C POSIX.1 XPG4 XPG4.2 C99 Single UNIX Specification, Version 3	both	

Format

```
#include <ctype.h>

int tolower(int c);   /* Convert c to lowercase if appropriate */
int toupper(int c);   /* Convert c to uppercase if appropriate */
```

General description

Converts *c* to a lowercase letter, if possible. Conversely, the toupper() function converts *c* to an uppercase letter, if possible.

The DBCS is not supported. The use of characters from the DBCS results in unspecified behavior.

Returned value

If successful, tolower() and toupper() return the corresponding character, as defined in the LC_CTYPE category of the current locale, if such a character exists.

If unsuccessful, tolower() and toupper() return the unchanged value *c*.

Example

CELEBT15

```
/* CELEBT15

   This example demonstrates the result of using
   &toupper. and &tolower. on a lower-case a.

 */
#include <stdio.h>
#include <ctype.h>

int main(void)
{
   int ch;

   ch = 0x81;
   printf("toupper=%#04x\n", toupper(ch));
   printf("tolower=%#04x\n", tolower(ch));
}
```

Related information

- “ctype.h” on page 21
- “isalnum() to isxdigit() — Test integer value” on page 895
- “towlower(), towupper() — Convert wide character case” on page 1908

_tolower() — Translate uppercase characters to lowercase

Standards

Standards / Extensions	C or C++	Dependencies
XPG4 XPG4.2 Single UNIX Specification, Version 3	both	

Format

```
#define _XOPEN_SOURCE
#include <ctype.h>

int _tolower(int c);
```

General description

The _tolower() macro is equivalent to tolower(c) except that the argument c must be an uppercase letter.

Returned value

_tolower() returns the lowercase letter corresponding to the argument passed.

Related information

- “ctype.h” on page 21
- “tolower(), toupper() — Convert character case” on page 1897

t_open() — Establish a transport endpoint

Standards

Standards / Extensions	C or C++	Dependencies
XPG4.2	both	

Format

```
#define _XOPEN_SOURCE_EXTENDED 1
#include <xti.h>

int t_open(char *name, int oflag, struct t_info *info);
```

General description

t_open() must be called as the first step in the initialization of a transport endpoint. This function establishes a transport endpoint by supplying a transport provider identifier that indicates a particular transport provider (that is, transport protocol) and returning a file descriptor that identifies that endpoint.

Note: There must be at least one available file descriptor less than 65536.

The argument *name* points to a transport provider identifier. The only supported transport provider is "/dev/tcp", indicating a TCP transport provider. No device by that name actually exists in the file system. It is purely used to follow historical convention. The argument *oflag* identifies any open flags (as in open()). It is constructed from O_RDWR optionally bitwise inclusive-OR'ed with O_NONBLOCK. These flags are defined by the header <fcntl.h>. The file descriptor returned by t_open() will be used by all subsequent functions to identify the particular local transport endpoint.

This function also returns various default characteristics of the underlying transport protocol by setting fields in the *info* structure. This argument points to a t_info structure which contains the following members:

```
long addr;      /* max size of the transport protocol address  */
long options;   /* max number of bytes of                      */
                /* protocol-specific options                   */
```

```
long tsdu;      /* max size of a transport service data       */
                /* unit (TSDU)                                 */
long etsdu;     /* max size of an expedited transport         */
                /* service data unit (ETSDU)                   */
long connect;   /* max amount of data allowed on              */
                /* connection establishment functions         */
long discon;    /* max amount of data allowed on              */
                /* t_snddis() and t_rcvdis() functions        */
long servtype;  /* service type supported by the              */
                /* transport provider                         */
long flags;     /* other info about the transport provider    */
```

The fields take on the following values:

addr The size of a struct sockaddr_in is returned.

options
The value 304, which is the maximum number of bytes of options which can possibly be specified or requested, is returned.

tsdu Zero is returned, indicating that the TCP transport provider does not support the concept of TSDUs.

etsdu A value of -1 is returned, indicating that there is no limit on the size of an ETSDU.

connect
A value of -2 is returned, indicating that the TCP transport provider does not allow data to be sent with connection establishment functions.

discon A value of -2 is returned, indicating that the transport provider does not allow data to be sent with the abortive release functions.

servtype
T_COTS is always returned, since this is the only service type supported.

flags The T_SENDZERO bit is always set in this field, indicating that the TCP transport provider supports the sending of zero-length TSDUs.

If info is set to a NULL pointer by the transport user, no protocol information is returned by t_open().

Valid states: T_UNINIT

Returned value

If successful, t_open() returns a valid file descriptor.

If unsuccessful, t_open() returns -1 and sets errno to one of the following values:

Error Code
Description

TBADFLAG
An invalid flag is specified.

TBADNAME
Invalid transport provider name.

TPROTO
This error indicates that a communication problem has been detected between XTI and the transport provider for which there is no other suitable XTI (*t_errno*).

TSYSERR
A system error has occurred during execution of this function.

Related information

- “xti.h” on page 87
- “open() — Open a file” on page 1152

t_optmgmt() — Manage options for a transport endpoint

Standards

Standards / Extensions	C or C++	Dependencies
XPG4.2	both	

Format

```
#define _XOPEN_SOURCE_EXTENDED 1
#include <xti.h>

int t_optmgmt(int fd, struct t_optmgmt *req, struct t_optmgmt *ret);
```

General description

Enables a transport user to retrieve, verify or negotiate protocol options with the transport provider. The argument *fd* identifies a transport endpoint. The *req* and *ret* arguments point to a t_optmgmt structure containing the following members:

```
struct netbuf   opt;
long            flags;
```

The *opt* field identifies protocol options and the *flags* field is used to specify the action to take with those options.

The options are represented by a *netbuf* structure in a manner similar to the address in t_bind(). The *netbuf* structure contains the following members:

```
unsigned int     maxlen      maximum buffer value length
unsigned int     len         actual buffer value length
char *           buf         pointer to buffer
```

The argument *req* is used to request a specific action of the provider and to send options to the provider. The argument *len* specifies the number of bytes in the options, *buf* points to the options buffer, and *maxlen* has no meaning for the *req* argument. The transport provider may return options and flag values to the user through *ret*. For *ret*, *maxlen* specifies the maximum size of the options buffer, and *buf* points to the buffer where the options are to be placed. On return, *len* specifies the number of bytes of options returned. The value in *maxlen* has no meaning for the *req* argument, but must be set in the *ret* argument to specify the maximum number of bytes the options buffer can hold.

Each option in the options buffer is of the form struct *t_opthdr* possibly followed by an option value. The *t_opthdr* structure contains the following members:

AMODE(31):

```
unsigned long    len         sizeof(t_opthdr)+optval len
unsigned long    level       protocol affected
unsigned long    name        option value
unsigned long    status      status value
```

AMODE(64):

```
unsigned int    len         sizeof(t_opthdr)+optval len
unsigned int    level       protocol affected
unsigned int    name        option value
unsigned int    status      status value
```

The *level* field of struct t_opthdr identifies the XTI level or a protocol of the transport provider. The *name* field identifies the option within the level, and *len* contains its total length. The total length is the length of the option header t_opthdr plus the length of the option value. If t_optmgmt() is called with the action T_NEGOTIATE set, the *status* field of the returned options contains information about the success or failure of a negotiation.

Each option in the input or output option buffer must start at a long-word boundary. The macro OPT_NEXTHDR(*pbuf*, *buflen*, *poption*) can be used for that purpose. The parameter *pbuf* denotes a pointer to an option buffer *opt.buf*, and *buflen* is its length. The parameter *poption* points to the current option in the option buffer. OPT_NEXTHDR returns a pointer to the position of the next option, or returns a NULL pointer if the option buffer is exhausted. The macro is helpful for writing and reading. See “xti.h” on page 87 for the exact definition.

If the transport user specifies several options on input, all options must address the same level.

If any option in the options buffer does not indicate the same level as the first option, or the level specified is unsupported, then the t_optmgmt() request will fail with TBADOPT. If the error is detected, some options have possibly been successfully negotiated. The transport user can check the current status by calling t_optmgmt() with the T_CURRENT flag set.

Before using this function, you should read Chapter 6 , “Use of Options in XTI”, in *X/Open CAE Specification, Networking Services, Issue 4*

The *flags* field of *req* must specify one of the following actions:

T_NEGOTIATE

This action enables the transport user to negotiate option values.

The user specifies the options of interest and their values in the buffer specified by *req->opt.buf* and *req->opt.len*. The negotiated option values are returned in the buffer pointed to by *ret->opt.buf*. The *status* field of each returned option is set to indicate the result of the negotiation. The value is T_SUCCESS if the proposed value was negotiated, T_PARTSUCCESS if a degraded value was negotiated, T_FAILURE if the negotiation failed (according to the negotiation rules), T_NOTSUPPORT if the transport provider does not support this option or illegally requests negotiation of a privileged option, and T_READONLY if modification of a read-only option was requested. If the status is T_SUCCESS, T_FAILURE, T_NOTSUPPORT or T_READONLY, the returned option value is the same as the one requested on input.

The overall result of the negotiation is returned in *ret->flags*.

This field contains the worst single result, whereby the rating is done according to the order T_NOTSUPPORT, T_READONLY, T_FAILURE, T_PARTSUCCESS, T_SUCCESS. The value T_NOTSUPPORT is the worst result and T_SUCCESS is the best.

For each level, the option T_ALLOPT (see below) can be requested on input. No value is given with this option; only the *t_opthdr* part is specified. This input requests to negotiate all supported options of this level to their default values. The result is returned option by option in *ret->opt.buf*. (Note that depending on the state of the transport endpoint, not all requests to negotiate the default value may be successful.)

T_CHECK

This action enables the user to verify whether the options specified in *req* are supported by the transport provider.

If an option is specified with no option value (it consists only of a t_opthdr structure), the option is returned with its status field set to T_SUCCESS if it is supported, T_NOTSUPPORT if it is not or needs additional user privileges, and T_READONLY if it is read-only (in the current XTI state). No option value is returned.

If an option is specified with an option value, the *status* field of the returned option has the same value, as if the user had tried to negotiate this value with T_NEGOTIATE. If the status is T_SUCCESS, T_FAILURE, T_NOTSUPPORT or T_READONLY, the returned option value is the same as the one requested on input.

The overall result of the option checks is returned in *ret->flags*. This field contains the worst single result of the option checks, whereby the rating is the same as for T_NEGOTIATE.

Note that no negotiation takes place. All currently effective option values remain unchanged.

T_DEFAULT

This action enables the transport user to retrieve the default option values. The user specifies the options of interest in *req->opt.buf*. The option values are irrelevant and will be ignored. It is sufficient to specify the *t_opthdr* part of an option only. The default values are then returned in *ret->opt.buf*.

The status field returned is T_NOTSUPPORT if the protocol level does not support this option or the transport user illegally requested a privileged option, T_READONLY if the option is read-only, and set to T_SUCCESS in all other cases. The overall result of the request is returned in *ret->flags*. This field contains the worst single result, whereby the rating is the same as for T_NEGOTIATE.

For each level, the option T_ALLOPT (see below) can be requested on input. All supported options of this level with their default values are then returned. In this case, *ret->opt.maxlen* must be given at least the value *info->options* (see t_getinfo(), t_open()) before the call.

T_CURRENT

This action enables the transport user to retrieve the currently effective option values. The user specifies the options of interest in *req->opt.buf*. The option values are irrelevant and will be ignored. It is sufficient to specify the *t_opthdr* part of an option only. The currently effective values are then returned in *ret->opt.buf*.

The status field returned is T_NOTSUPPORT if the protocol level does not support this option, or the transport user illegally requested a privileged option, T_READONLY if the option is read-only, and set to T_SUCCESS in all other cases. The overall result of the request is returned in *ret->flags*. This field contains the worst single result, whereby the rating is the same as for T_NEGOTIATE.

For each level, the option T_ALLOPT (see below) can be requested on input. All supported options of this level with their currently effective values are then returned.

The option T_ALLOPT can only be used with t_optmgmt() and the actions T_NEGOTIATE, T_DEFAULT and T_CURRENT. It can be used with any supported level and addresses all supported options of this level. The option has no value; it consists of a *t_opthdr* only. Since in a t_optmgmt() call only options of one level may be addressed, this option should not be requested together with other options. The function returns as soon as this option has been processed.

Options are independently processed in the order they appear in the input option buffer. If an option is multiply input, it depends on the implementation whether it is multiply output or whether it is returned only once.

The function t_optmgmt() may block under various circumstances and depending on the implementation. The function will block, for instance, if the protocol addressed by the call resides on a separate controller. It may also block due to flow control constraints. For example, if data sent previously across this transport endpoint has not yet been fully processed. If the function is interrupted by a signal, the option negotiations that have been done so far may remain valid. The behavior of the function is not changed if O_NONBLOCK is set.

Valid states: All - except for T_UNINIT

XTI-level options: XTI-level options are not specific for a particular transport provider. An XTI implementation supports none, all or any subset of the options defined below. An implementation may restrict the use of any of these options by offering them only in the privileged or read-only mode, or if *fd* relates to specific transport providers.

The subsequent options are not association-related They may be negotiated in all XTI states except T_UNINIT. See Chapter 6 , “Use of Options in XTI”, in *X/Open CAE Specification, Networking Services, Issue 4* for more information.

The protocol level is XTI_GENERIC. For this level, the following options are defined:

XTI_DEBUG

This option enables debugging. The valid values of this option are:

- None (option header only) - indicating that debug is to be turned off.
- -1 - indicating that debug output is to go to stderr.
- A file descriptor - indicating the destination file for debug output.

The debug output contains varying information depending on the XTI services invoked. It is meant to be used by customer support personnel.

XTI_LINGER

This option is used to linger the execution of a t_close() or close() if send data is still queued in the send buffer. The option value specifies the linger period. If a close() or t_close() is issued and the send buffer is not empty, the system attempts to send the pending data within the linger period before closing the endpoint. Data still pending after the linger period has elapsed is discarded.

Depending on the implementation, t_close() or close() either block for at maximum the linger period, or immediately return, whereupon the system holds the connection in existence for at most the linger period.

The *option* value consists of a structure t_linger declared as:

```
struct t_linger {
       long l_onoff;   /* switch option on/off     */
       long l_linger;  /* linger period in seconds */
}
```

Legal values for the field *l_onoff* are:

T_NO switch option off

T_YES
activate option

The value *l_onoff* is an absolute requirement.

The field *l_linger* determines the linger period in seconds. The transport user can request the default value by setting the field to T_UNSPEC. The default timeout value depends on the underlying transport provider (it is often T_INFINITE). Legal values for this field are T_UNSPEC, T_INFINITE and all nonnegative numbers.

The *l_linger* value is not an absolute requirement. The implementation may place upper and lower limits to this value. Requests that fall short of the lower limit are negotiated to the lower limit.

Note that this option does not linger the execution of t_snddis().

XTI_RCVBUF
This option is used to adjust the internal buffer size allocated for the receive buffer. The buffer size may be increased for high-volume connections, or decreased to limit the possible backlog of incoming data.

This request is not an absolute requirement. The implementation may place upper and lower limits on the option value. Requests that fall short of the lower limit are negotiated to the lower limit.

Legal values are all positive numbers.

XTI_SNDBUF
This option is used to adjust the internal buffer size allocated for the send buffer.

This request is not an absolute requirement. The implementation may place upper and lower limits on the option value. Requests that fall short of the lower limit are negotiated to the lower limit.

Legal values are all positive numbers.

TCP-level options: The protocol level is INET_TCP. The following TCP-level options are supported. They are not association-related.

TCP_KEEPALIVE
If this option is set, a keep-alive timer is activated to monitor idle connections that might no longer exist. If a connection has been idle since the last keep-alive timeout, a keep-alive packet is sent to check if the connection is still alive or broken.

The option value consists of a structure *t_kpalive* declared as:

```
struct t_kpalive {
     long    kp_onoff;       /* switch option on/off */
     long    kp_timeout;     /* keep-alive timeout in minutes */
 }
```

Legal values fot the field *kp_onoff* are:

T_NO switch keep-alive timer off

T_YES
activate keep-alive timer

The field *kp_timeout* determines the frequency of keep-alive packets being sent, in minutes. The transport user can request the default value by setting the field to T_UNSPEC. The default is 120 minutes. Legal values for this field are T_UNSPEC and all positive numbers.

The timeout value is not an absolute requirement. However, no limits are currently specified by the TCP transport provider.

IP-level options: The protocol level is INET_IP. The following IP-level options are supported. The are not association-related.

IP_REUSEADDR
Generally, users are not allowed to bind more than one transport endpoint to addresses with identical port numbers, If IP_REUSEADDR is set to T_YES this restriction is relaxed in the sense that it is now permissible to bind a transport endpoint to an address with a port number and an under-specified internet address and further endpoints to addresses with the same port number and (mutually exclusive) fully specified internet addresses.

Returned value

If successful, t_optmgmt() returns 0.

If unsuccessful, t_optmgmt() returns -1 and sets errno to one of the following values:

Error Code
Description

TACCES
The user does not have permission to negotiate the specified options.

TBADF
The specified file descriptor does not refer to a transport endpoint.

TBADFLAG
An invalid flag was specified.

TBADOPT
The specified options were in an incorrect format or contained illegal information.

TBUFOVFLW
The number of bytes allowed for an incoming argument (*maxlen*) is greater than 0 but not sufficient to store the value of that argument. The information to be returned in *ret* will be discarded.

TNOTSUPPORT
This action is not supported by the transport provider.

TOUTSTATE
The function was issued in the wrong sequence.

TPROTO
This error indicates that a communication problem has been detected between XTI and the transport provider for which there is no other suitable XTI (*t_errno*).

TSYSERR
A system error has occurred during execution of this function.

Related information

- "xti.h" on page 87
- "t_accept() — Accept a connect request" on page 1810
- "t_alloc() — Allocate a library structure" on page 1814
- "t_connect() — Establish a connection with another transport user" on page 1838
- "t_getinfo() — Get protocol-specific service information" on page 1874
- "t_listen() — Listen for a connect indication" on page 1883
- "t_open() — Establish a transport endpoint" on page 1899
- "t_rcvconnect() — Receive the confirmation from a connect request" on page 1910
- *X/Open CAE Specification, Networking Services, Issue 4*, Chapter 6 "Use of Options in XTI".

_toupper() — Translate lowercase characters to uppercase

Standards

Standards / Extensions	C or C++	Dependencies
XPG4 XPG4.2 Single UNIX Specification, Version 3	both	

Format

```
#define _XOPEN_SOURCE
#include <ctype.h>

int _toupper(int c);
```

General description

The _toupper() macro is equivalent to toupper(c) except that the argument c must be a lowercase letter.

Returned value

_toupper() returns the uppercase letter corresponding to the argument passed.

Related information

- "ctype.h" on page 21
- "isalnum() to isxdigit() — Test integer value" on page 895
- "tolower(), toupper() — Convert character case" on page 1897

towlower(), towupper() — Convert wide character case

Standards

Standards / Extensions	C or C++	Dependencies
POSIX.1 XPG4 XPG4.2 Single UNIX Specification, Version 3	both	

Format

```
#include <wctype.h>

wint_t towlower(wint_t wc);
wint_t towupper(wint_t wc);
```

General description

Converts *wc* to the corresponding lowercase letter. The towupper() function converts *wc* to the corresponding uppercase letter.

Returned value

If *wc* is a wide character for which iswupper() (or iswlower()) is true and there is a corresponding wide character for which iswlower() (or iswupper()) is true, towlower() (or towupper()) returns the corresponding wide character; otherwise, *wc* is returned unchanged.

Related information

- "wctype.h" on page 86
- "iswalnum() to iswxdigit() — Test wide integer value" on page 920
- "tolower(), toupper() — Convert character case" on page 1897

towctrans() — Transliterate wide character transliteration

The information for this function is included in "wctrans(), towctrans() — Transliterate wide character" on page 2055.

t_rcv() — Receive data or expedited data sent over a connection

Standards

Standards / Extensions	C or C++	Dependencies
XPG4.2	both	

Format

```
#define _XOPEN_SOURCE_EXTENDED 1
#include <xti.h>

int t_rcv(int fd, char *buf, unsigned int nbytes, int *flags);
```

General description

Receives either normal or expedited data. The argument *fd* identifies the local transport endpoint through which data will arrive. *buf* points to a receive buffer where user data is placed. *nbytes* specifies the size of the receive buffer. The argument *flags* may be set on return from t_rcv() and specifies optional flags as described below.

By default, t_rcv() operates in synchronous mode and will wait for data to arrive if none is currently available. However, if O_NONBLOCK is set (using t_open() or fcntl()), t_rcv() will execute in asynchronous mode and will fail if no data is available. (See TNODATA below.)

On return from the call, if T_MORE is set in *flags*, this indicates that there is more data, and the current expedited transport service data unit (ETSDU) must be received in multiple t_rcv() calls. In the asynchronous mode, the T_MORE flag may be set on return from the t_rcv() call even when the number of bytes received is less than the size of the receive buffer specified. Each t_rcv() with the T_MORE flag set indicates that another t_rcv() must follow to get more data for the current ETSDU. The end of the ETSDU is identified by the return of a t_rcv() call with the T_MORE flag not set. The T_MORE flag is not meaningful for normal data when using the TCP transport provider and should be ignored. If *nbytes* is greater than zero on the call to t_rcv() , t_rcv() will return 0 only if the end of a TSDU is being returned to the user.

On return, the data returned is expedited data if T_EXPEDITED is set in *flags*. If the number of bytes of expedited data exceeds *nbytes*, t_rcv() will set T_EXPEDITED and T_MORE on return from the initial call. Subsequent calls to retrieve the remaining ETSDU will have T_EXPEDITED set on return. The end of the ETSDU is identified by the return of a t_rcv() call with the T_MORE flag not set.

In synchronous mode, the only way for the user to be notified of the arrival of normal or expedited data is to issue this function or check for the T_DATA or T_EXDATA events using the t_look() function. Additionally, the process can arrange to be notified by the select/poll interface.

Valid states: T_DATAXFER

Returned value

If successful, t_rcv() returns the number of bytes received.

If unsuccessful, t_rcv() returns -1 and sets errno to one of the following values:

Error Code
Description

TBADF
The specified file descriptor does not refer to a transport endpoint.

TLOOK
An asynchronous event has occurred on this transport endpoint and requires immediate attention.

TNODATA
O_NONBLOCK was set, but no data is currently available from the transport provider.

TNOTSUPPORT
This function is not supported by the underlying transport provider.

TOUTSTATE
The function was issued in the wrong sequence on the transport endpoint referenced by *fd*.

TPROTO
This error indicates that a communication problem has been detected between XTI and the transport provider for which there is no other suitable XTI (*t_errno*).

TSYSERR
A system error has occurred during execution of this function.

Related information

- "xti.h" on page 87
- "fcntl() — Control open file descriptors" on page 474
- "t_getinfo() — Get protocol-specific service information" on page 1874
- "t_look() — Look at the current event on a transport endpoint" on page 1885
- "t_open() — Establish a transport endpoint" on page 1899
- "t_snd() — Send data or expedited data over a connection" on page 1921

t_rcvconnect() — Receive the confirmation from a connect request

Standards

Standards / Extensions	C or C++	Dependencies
XPG4.2	both	

Format

```
#define _XOPEN_SOURCE_EXTENDED 1
#include <xti.h>

int t_rcvconnect(int fd, struct t_call *call);
```

General description

Enables a calling transport user to determine the status of a previously sent connect request. t_rcvconnect() is used in conjunction with t_connect() to establish a connection in asynchronous mode. The connection will be established on successful completion of this function.

The argument *fd* identifies the local transport endpoint where communication will be established, and call contains information associated with the newly established connection. The argument call points to a *t_call* structure which contains the following members:

```
        struct netbuf   addr;
        struct netbuf   opt;
        struct netbuf   udata;
        int             sequence;
```

In *call*, *addr* returns the protocol address associated with the responding transport endpoint. *opt* presents any options associated with the connection. *udata* is meaningless since the TCP transport provider does not support transmission of user data during connection establishment. *sequence* has no meaning for this function.

The *maxlen* field of each argument must be set before issuing this function to indicate the maximum size of the buffer for each. However, *call* may be a NULL pointer, in which case no information is given to the user on return from t_rcvconnect() . By default, t_rcvconnect() executes in synchronous mode and waits for the connection to be established before returning. On return, the *addr* field contains the address of the remote endpoint, and *opt* reflects the result of negotiation of the options the user specified on input.

If O_NONBLOCK is set (using t_open() or fcntl()), t_rcvconnect() executes in asynchronous mode, and reduces to a poll for existing connect confirmations. If none are available, t_rcvconnect() fails and returns immediately without waiting for the connection to be established. (See `TNODATA` below.) In this case, t_rcvconnect() must be called again to complete the connection establishment phase and retrieve the information returned in call.

Valid states: T_OUTCON

Returned value

If successful, t_rcvconnect() returns 0.

If unsuccessful, t_rcvconnect() returns -1 and sets errno to one of the following values:

Error Code
Description

TBADF
The specified file descriptor does not refer to a transport endpoint.

TBUFOVFLW
The number of bytes allocated for an incoming argument (*maxlen*) is greater than 0 but not sufficient to store the value of that argument, and the connect information to be returned in *call* will be discarded. The provider's state, as seen by the user, will be changed to T_DATAXFER.

TLOOK
An asynchronous event has occurred on this transport connection and requires immediate attention.

TNODATA
O_NONBLOCK was set, but a connect confirmation has not yet arrived.

TNOTSUPPORT
This function is not supported by the underlying transport provider.

TOUTSTATE
The function was issued in the wrong sequence on the transport endpoint referenced by *fd*.

TPROTO
This error indicates that a communication problem has been detected between XTI and the transport provider for which there is no other suitable XTI (*t_errno*).

TSYSERR
A system error has occurred during execution of this function.

Related information

- “xti.h” on page 87
- “t_accept() — Accept a connect request” on page 1810
- “t_alloc() — Allocate a library structure” on page 1814

- "t_bind() — Bind an address to a transport endpoint" on page 1821
- "t_connect() — Establish a connection with another transport user" on page 1838
- "t_listen() — Listen for a connect indication" on page 1883
- "t_open() — Establish a transport endpoint" on page 1899
- "t_optmgmt() — Manage options for a transport endpoint" on page 1901

t_rcvdis() — Retrieve information from disconnect

Standards

Standards / Extensions	C or C++	Dependencies
XPG4.2	both	

Format

```
#define _XOPEN_SOURCE_EXTENDED 1
#include <xti.h>

int t_rcvdis(int fd, struct t_discon *discon);
```

General description

Used to identify the cause of a disconnect. The argument *fd* identifies the local transport endpoint where the connection existed, and discon points to a t_discon structure containing the following members:

```
struct netbuf   udata;
int             reason;
int             sequence;
```

The field *reason* specifies the reason for the disconnect through a protocol-dependent reason code, udata is always empty since the TCP transport provider does not support sending of user data with a disconnect, and *sequence* may identify an outstanding connect indication with which the disconnect is associated. The field *sequence* is only meaningful when t_rcvdis() is issued by a passive transport user who has executed one or more t_listen() functions and is processing the resulting connect indications. If a disconnect indication occurs, *sequence* can be used to identify which of the outstanding connect indications is associated with the disconnect.

If a user does not care if there is incoming data and does not need to know the value of *reason* or *sequence*, *discon* may be a NULL pointer. However, if a user has retrieved more than one outstanding connect indication (using t_listen()) and *discon* is a NULL pointer, the user will be unable to identify with which connect indication the disconnect is associated.

Valid states: T_DATAXFER,T_OUTCON,T_INCON(ocnt > 0)

Returned value

If successful, t_rcvdis() returns 0.

If unsuccessful, t_rcvdis() returns -1 and sets errno to one of the following values:

Error Code
Description

TBADF
The specified file descriptor does not refer to a transport endpoint.

TBUFOVFLW
The number of bytes allocated for incoming data (maxlen) is greater than 0 but not sufficient to store the data. If *fd* is a passive endpoint with ocnt > 1, it remains in state T_INCON; otherwise, the endpoint state is set to T_IDLE.

TNODIS
No disconnect indication currently exists on the specified transport endpoint.

TNOTSUPPORT
This function is not supported by the underlying transport provider.

TOUTSTATE
The function was issued in the wrong sequence on the transport endpoint referenced by *fd*.

TPROTO
This error indicates that a communication problem has been detected between XTI and the transport provider for which there is no other suitable XTI (*t_errno*).

TSYSERR
A system error has occurred during execution of this function.

Related information

- "xti.h" on page 87
- "t_alloc() — Allocate a library structure" on page 1814
- "t_connect() — Establish a connection with another transport user" on page 1838
- "t_listen() — Listen for a connect indication" on page 1883
- "t_open() — Establish a transport endpoint" on page 1899
- "t_snddis() — Send user-initiated disconnect request" on page 1923

t_rcvrel() — Acknowledge receipt of an orderly release indication

Standards

Standards / Extensions	C or C++	Dependencies
XPG4.2	both	

Format

```
#define _XOPEN_SOURCE_EXTENDED 1
#include <xti.h>

int t_rcvrel(int fd);
```

General description

Since orderly release is not supported, this function always fails.

Returned value

t_rcvrel() always returns -1 and sets *t_errno* to TNOTSUPPORT.

Related information

- "xti.h" on page 87
- "t_getinfo() — Get protocol-specific service information" on page 1874
- "t_open() — Establish a transport endpoint" on page 1899
- "t_sndrel() — Initiate an orderly release" on page 1924

t_rcvudata() — Receive a data unit

Standards

Standards / Extensions	C or C++	Dependencies
XPG4.2	both	

Format

```
#define _XOPEN_SOURCE_EXTENDED 1
#include <xti.h>

int t_rcvudata(int fd, struct t_unitdata *unitdata, int *flags);
```

General description

T_CLTS service is not supported in this implementation, so this function always fails.

Returned value

t_rcvudata() always returns -1 and sets t_errno to TNOTSUPPORT.

Related information

- "xti.h" on page 87
- "t_open() — Establish a transport endpoint" on page 1899
- "t_sndudata() — Send a data unit" on page 1925
- "t_rcvuderr() — Receive a unit data error indication"

t_rcvuderr() — Receive a unit data error indication

Standards

Standards / Extensions	C or C++	Dependencies
XPG4.2	both	

Format

```
#define _XOPEN_SOURCE_EXTENDED 1
#include <xti.h>

int t_rcvuderr(int fd, struct t_uderr *uderr);
```

General description

T_CLTS service is not supported in this implementation, so this function always fails.

Returned value

t_rcvuderr() always returns -1 and sets t_errno to TNOTSUPPORT.

Related information

- “xti.h” on page 87
- “t_open() — Establish a transport endpoint” on page 1899
- “t_rcvudata() — Receive a data unit” on page 1914
- “t_sndudata() — Send a data unit” on page 1925

trunc(), truncf(), truncl() — Truncate an integer value

Standards

Standards / Extensions	C or C++	Dependencies
C99 Single UNIX Specification, Version 3 C++ TR1 C99	both	z/OS V1R5

Format

```
#define _ISOC99_SOURCE
#include <math.h>

double trunc(double x);
float truncf(float x);
long double Truncl(long double x);
```

C++ TR1 C99

```
#define _TR1_C99
#include <math.h>

float trunc(float x);
long double trunc(long double x);
```

General description

The trunc functions round *x* to the integer value, in floating-point format, nearest to but no larger in magnitude than *x*.

Note: The following table shows the viable formats for these functions. See “IEEE binary floating-point” on page 94 for more information about IEEE Binary Floating-Point.

Function	Hex	IEEE
trunc	X	X
truncf	X	X
truncl	X	X

Returned value

The trunc functions return the truncated integer value of *x*.

Related information

- “math.h” on page 44

truncd32(), truncd64(), truncd128() — Truncate an integer value

Standards

Standards / Extensions	C or C++	Dependencies
C/C++ DFP	both	z/OS V1.8

Format

```
#define __STDC_WANT_DEC_FP__
#include <math.h>

_Decimal32  truncd32(_Decimal32 x);
_Decimal64  truncd64(_Decimal64 x);
_Decimal128 truncd128(_Decimal128 x);
_Decimal32  trunc(_Decimal32 x);      /* C++ only */
_Decimal64  trunc(_Decimal64 x);      /* C++ only */
_Decimal128 trunc(_Decimal128 x);     /* C++ only */
```

General description

The trunc functions round *x* to the integer value, in decimal floating-point format, nearest to but no larger in magnitude than *x*.

Notes:

1. To use IEEE decimal floating-point, the hardware must have the Decimal Floating-Point Facility installed.
2. These functions work in IEEE decimal floating-point format. See "IEEE Decimal Floating-Point" for more information.

Returned value

The trunc functions return the truncated integer value of *x*.

Example

```
/* CELEBT21

   This example illustrates the truncd128() function.

*/

#define  __STDC_WANT_DEC_FP__
#include <math.h>
#include <stdio.h>

int main(void)
{
   _Decimal128 x = 123456789.40DL, y;

   y = truncd128(x);

   printf("The result of truncd128(%DDf) is %DDf\n", x, y);
}
```

Related information

- “math.h” on page 44
- “trunc(), truncf(), truncl() — Truncate an integer value” on page 1915

truncate() — Truncate a file to a specified length

Standards

Standards / Extensions	C or C++	Dependencies
XPG4.2 Single UNIX Specification, Version 3	both	

Format

```
#define _XOPEN_SOURCE_EXTENDED 1
#include <unistd.h>

int truncate(const char *path, off_t length);
```

General description

Truncates the file indicated by the *path* to the indicated *length*. The calling process must have write permission for the file. If the file size exceeds *length*, any extra data is discarded. If the file size is smaller than length, bytes between the old and new lengths are read as zeros. A change to the size of the file has no impact on the file offset.

If truncate() would cause the file size to exceed the soft file size limit for the process, truncate() will fail and a SIGXFSZ signal will be generated for the process.

If successful, truncate() marks the `st_ctime` and `st_mtime` fields of the file.

If unsuccessful, the file is unchanged.

Large file support for z/OS UNIX files: Large z/OS UNIX files are supported automatically for AMODE 64 C/C++ applications. AMODE 31 C/C++ applications must be compiled with the option LANGLVL(LONGLONG) and define the _LARGE_FILES feature test macro before any headers are included to enable this function to operate on z/OS UNIX files that are larger than 2 GB in size. File size and offset fields are enlarged to 63 bits in width. Therefore, any other function operating on the file is required to define the _LARGE_FILES feature test macro as well.

Returned value

If successful, truncate() returns 0.

If unsuccessful, truncate() returns -1 and sets errno to one of the following values:

Error Code
Description

EFBIG
The length argument was greater than the maximum file size.

EINTR
A signal was caught during execution

EINVAL
path does not refer to a regular file, or the length specified is incorrect.

EIO An I/O error occurred while reading from or writing to a file system.

EISDIR
The file specified is a directory. The system cannot perform the requested function on a directory.

EROFS
The file resides on a read-only file system.

Related information

- "ftruncate() — Truncate a file" on page 663
- "open() — Open a file" on page 1152

tsched() — Schedule MTF subtask

Standards

Standards / Extensions	C or C++	Dependencies
Language Environment	C only	

Format

```
#include <mtf.h>

int tsched(int task_id, const char *func_name, ...);
```

General description

Restriction: This function is not supported in AMODE 64.

The tsched() built-in library function is used to schedule parallel functions under MVS.

The tsched() library function schedules a parallel function (*func_name*) to be executed in a subtask that has been attached and initialized by tinit(). The first argument *task_id* can be used to specify the task where the parallel function is to be executed. The *task_id* can be set to `MTF_ANY` to indicate that the parallel function can be run in any task or, *task_id* can be set to a specific task number between 1 and the number of subtasks specified on tinit().

The tsched() function allocates approximately 2K of space as a work area. This work area is not freed. However, a call to the tsyncro() function will make work areas previously allocated by tsched() available for reuse by tsched(). Because tsched() does not free the work area storage, it is possible the tsched() will cause an application Short on Storage condition to occur if enough calls are made to the tsched() function.

You can call tsched() from your main C task as often as necessary to schedule parallel functions for execution. If all of the subtasks are busy running previously scheduled functions, each call in excess of the number of subtasks will cause *func_name* to be run after previously scheduled functions in the first qualifying subtask that becomes available.

All scheduled functions must be computationally independent. A function cannot use variables that are modified by another scheduled function or the scheduling function. To determine if all other scheduled functions have completed, use the tsyncro() library function (see "tsyncro() — Wait for MTF subtask termination" on page 1928).

The name of the parallel function, *func_name*, must not be longer than 8 characters. If it is, the name will be truncated to the first 8 characters with no warning.

Usually tsched() returns to the calling main task program before the scheduled parallel function has completed execution. Therefore, you must call tsyncro() to ensure that your parallel functions have completed execution.

If tinit() has not been successfully called before tsched(), tsched() indicates that MTF is inactive.

If tinit() is called by a program running under IMS, CICS, or DB2, the request will not be processed and the returned value will indicate that MTF calls are not supported under these systems.

Note: This function is *not* supported under the z/OS UNIX with the POSIX(ON) runtime option.

Returned value

If successful, scheduling the parallel function for execution in a subtask, tsched() returns MTF_OK.

If unsuccessful, tsched() returns one of the following values:

Error Code
Description

EBADLNKG
tsched() has been invoked using an invalid linkage. The header file mtf.h may have been missing from the source at compile.

EENTRY
The parallel function was not found in the parallel module.

EINACTIVE
MTF is inactive.

ENOMEM
There was insufficient storage for MTF-internal areas.

ETASKABND
One or more subtasks have terminated abnormally.

ETASKID
The task_id specified is not valid.

Note: These values are macros. They can be found in the mtf.h header file.

Related information

- "mtf.h" on page 49
- "tinit() — Attach and initialize MTF subtasks" on page 1882
- "tsyncro() — Wait for MTF subtask termination" on page 1928
- "tterm() — Terminate MTF subtasks" on page 1929

tsearch() — Binary tree search

Standards

Standards / Extensions	C or C++	Dependencies
XPG4 XPG4.2 Single UNIX Specification, Version 3	both	

Format

```
#define _XOPEN_SOURCE
#include <search.h>

void *tsearch(const void *key, void **rootp,
             int (*compar)(const void *, const void *));
```

General description

The tsearch() function is used to build and access a binary search tree. The *key* argument is a pointer to an element to be accessed or stored. If there is a node in the tree whose element is equal to the value pointed to by key, a pointer to this found node is returned. Otherwise, the value pointed to by *key* is inserted (that is, a new node is created and the value of *key* is copied to this node), and a pointer to this node returned. Only pointers are copied, so the calling routine must store the data. The *rootp* argument points to a variable that points to the root node of the tree. A NULL pointer value for the variable pointed to by *rootp* denotes an empty tree; in this case, the variable will be set to point to the node which will be at the root of the new tree.

Comparisons are made with a user-supplied routine, the address of which is passed as the *compar* argument. This routine is called with two arguments, the pointers to the elements being compared. The user-supplied routine must return an integer less than, equal to or greater than 0, according to whether the first argument is to be considered less than, equal to or greater than the second argument. The comparison functions need not compare every byte, so arbitrary data may be contained in the elements in addition to the values being compared.

Threading Behavior: Since the tree is anchored by the user's *rootp* pointer, the tree storage is visible to the user and could be shared among threads. The user would be responsible for serializing access to a shared tree. There are no variables related to these functions which are internal to the library and/or give rise to multithreading considerations.

Special behavior for C++: Because C and C++ linkage conventions are incompatible, tsearch() cannot receive a C++ function pointer as the comparator argument. If you attempt to pass a C++ function pointer to tsearch(), the compiler will flag it as an error. You can pass a C or C++ function to tsearch() by declaring it as extern “C”.

Returned value

If the node is found, tsearch() returns a pointer to it, otherwise it returns a pointer to the inserted item.

If there is not enough space available to create a new node, or if *rootp* is a NULL pointer on entry, tsearch() returns a NULL pointer.

No errors are defined.

Related information

- “search.h” on page 62
- “bsearch() — Search arrays” on page 220
- “hsearch() — Search hash tables” on page 817
- “lsearch() — Linear search and update” on page 1019
- “tdelete() — Binary tree delete” on page 1865
- “tfind() — Binary tree find node” on page 1870
- “twalk() — Binary tree walk” on page 1935

t_snd() — Send data or expedited data over a connection

Standards

Standards / Extensions	C or C++	Dependencies
XPG4.2	both	

Format

```
#define _XOPEN_SOURCE_EXTENDED 1
#include <xti.h>

int t_snd(int fd, char *buf, unsigned int nbytes, int flags);
```

General description

Sends either normal or expedited data. The argument *fd* identifies the local transport endpoint over which data should be sent, *buf* points to the user data, *nbytes* specifies the number of bytes of user data to be sent, and *flags* specifies any optional flags described below:

T_EXPEDITED
: If set in *flags*, the data will be sent as expedited data and will be subject to the interpretations of the transport provider.

T_MORE
: Since the TCP transport provider does not support the concept of a TSDU, the T_MORE flag is not meaningful and will be ignored if set.

By default, t_snd() operates in synchronous mode and may wait if flow control restrictions prevent the data from being accepted by the local transport provider at the time the call is made. However, if O_NONBLOCK is set (using t_open() or fcntl()), t_snd() will execute in asynchronous mode, and will fail immediately if there are flow control restrictions. The process can arrange to be informed when the flow control restrictions are cleared using either t_look() or select/poll.

If successful, t_snd() returns the number of bytes accepted by the transport provider. Normally this will equal the number of bytes specified in *nbytes*. However, if O_NONBLOCK is set, it is possible that only part of the data will actually be accepted by the transport provider. In this case, t_snd() will return a value that is less than the value of *nbytes*.

The size of each TSDU or ETSDU must not exceed the limits of the transport provider as specified by the current values in the TSDU or ETSDU fields in the *info* argument returned by t_getinfo() . The error `TLOOK` may be returned to inform the process that an event (for example, a disconnect) has occurred.

Valid states: T_DATAXFER

Returned value

If successful, t_snd() returns the number of bytes accepted by the transport provider.

Note that in asynchronous mode, if the number of bytes accepted by the transport provider is less than the number of bytes requested, this may indicate that the transport provider is blocked due to flow control.

If unsuccessful, t_snd() returns -1 and sets errno to one of the following values:

Error Code
Description

TBADDATA
Illegal amount of data:

- A single send was attempted specifying a TSDU (ETSDU) or fragment TSDU (ETSDU) greater than that specified by the current values of the TSDU or ETSDU fields in the info argument.
- Multiple sends were attempted resulting in a TSDU (ETSDU) larger than that specified by the current value of the TSDU or ETSDU fields in the *info* argument

TBADF
The specified file descriptor does not refer to a transport endpoint.

TBADFLAG
An invalid flag was specified.

TFLOW
O_NONBLOCK was set, but the flow control mechanism prevented the transport provider from accepting any data at this time.

TLOOK
An asynchronous event has occurred on this transport endpoint.

TNOTSUPPORT
This function is not supported by the underlying transport provider.

TOUTSTATE
The function was issued in the wrong sequence on the transport endpoint referenced by *fd*.

TPROTO
This error indicates that a communication problem has been detected between XTI and the transport provider for which there is no other suitable XTI (t_errno).

TSYSERR
A system error has occurred during execution of this function.

It is important to remember that the transport provider treats all users of a transport endpoint as a single user. Therefore if several processes issue concurrent t_snd() calls then the different data may be intermixed. Multiple sends which exceed the maximum TSDU or ETSDU size may not be discovered by XTI. In this case an implementation-dependent error will result (generated by the transport provider) perhaps on a subsequent XTI call. This error may take the form of a connection abort, a `TSYSERR`, a `TBADDATA` or a `TPROTO` error. If multiple sends which exceed the maximum TSDU or ETSDU size are detected by XTI, t_snd() fails with TBADDATA.

Related information

- "xti.h" on page 87
- "t_getinfo() — Get protocol-specific service information" on page 1874
- "t_open() — Establish a transport endpoint" on page 1899
- "t_rcv() — Receive data or expedited data sent over a connection" on page 1908

t_snddis() — Send user-initiated disconnect request

Standards

Standards / Extensions	C or C++	Dependencies
XPG4.2	both	

Format

```
#define _XOPEN_SOURCE_EXTENDED 1
#include <xti.h>

int t_snddis(int fd, struct t_call *call);
```

General description

Initiates an abortive release on an already established connection, or rejects a connect request. The argument *fd* identifies the local transport endpoint of the connection, and *call* specifies information associated with the abortive release. The argument *call* points to a *t_call* structure which contains the following members:

```
struct netbuf   addr;
struct netbuf   opt;
struct netbuf   udata;
int             sequence;
```

The values in *call* have different semantics, depending on the context of the call to t_snddis() . When rejecting a connect request, *call* must be non-NULL and contain a valid value of *sequence* to uniquely identify the rejected connect indication to the transport provider. The *sequence* field is only meaningful if the transport connection is in the T_INCON state. The *addr* and *opt* fields of *call* are ignored. In all other cases, *call* should be a NULL pointer, since its only use would be to specify user data to be passed on the disconnect request, which is not supported by the TCP transport provider.

t_snddis() is an abortive disconnect. Therefore a t_snddis() issued on a connection endpoint may cause data previously sent using t_snd(), or data not yet received, to be lost (even if an error is returned).

Because of implementation restrictions, a t_snddis() called on one descriptor referring to an endpoint will not affect descriptors in other processes referring to the same endpoint. If descriptors in multiple processes refer to the same endpoint, the endpoint will not actually be disconnected by a t_snddis in one process. Multiple processes cooperating on an endpoint are responsible for providing their own explicit synchronization to support coordinated disconnects.

Valid states: T_DATAXFER,T_OUTCON,T_INCON(ocnt > 0)

Returned value

If successful, t_snddis() returns 0.

If unsuccessful, t_snddis() returns -1 and sets errno to one of the following values:

Error Code
Description

TBADDATA
The amount of user data specified was not within the bounds allowed by the transport provider.

TBADF
The specified file descriptor does not refer to a transport endpoint.

TBADSEQ
An invalid sequence number was specified, or a NULL *call* pointer was specified, when rejecting a connect request.

TLOOK
An asynchronous event, which requires attention, has occurred.

TNOTSUPPORT
This function is not supported by the underlying transport provider.

TOUTSTATE
The function was issued in the wrong sequence on the transport endpoint referenced by *fd*.

TPROTO
This error indicates that a communication problem has been detected between XTI and the transport provider for which there is no other suitable XTI (t_errno).

TSYSERR
A system error has occurred during execution of this function.

Related information

- “xti.h” on page 87
- “t_connect() — Establish a connection with another transport user” on page 1838
- “t_getinfo() — Get protocol-specific service information” on page 1874
- “t_listen() — Listen for a connect indication” on page 1883
- “t_open() — Establish a transport endpoint” on page 1899

t_sndrel() — Initiate an orderly release

Standards

Standards / Extensions	C or C++	Dependencies
XPG4.2	both	

Format

```
#define _XOPEN_SOURCE_EXTENDED 1
#include <xti.h>

int t_sndrel(int fd);
```

General description

Since orderly release is not supported, t_sndrel() always fails.

Returned value

t_sndrel() always returns -1 and sets *t_errno* to TNOTSUPPORT.

Related information

- "xti.h" on page 87
- "t_getinfo() — Get protocol-specific service information" on page 1874
- "t_open() — Establish a transport endpoint" on page 1899
- "t_rcvrel() — Acknowledge receipt of an orderly release indication" on page 1913

t_sndudata() — Send a data unit

Standards

Standards / Extensions	C or C++	Dependencies
XPG4.2	both	

Format

```
#define _XOPEN_SOURCE_EXTENDED 1
#include <xti.h>

int t_sndudata(int fd, struct t_unitdata *unitdata);
```

General description

T_CLTS service is not supported in this implementation, so this function always fails.

Returned value

t_sndudata() always returns -1 and sets t_errno to TNOTSUPPORT.

Related information

- "xti.h" on page 87
- "t_open() — Establish a transport endpoint" on page 1899
- "t_rcvudata() — Receive a data unit" on page 1914
- "t_rcvuderr() — Receive a unit data error indication" on page 1914

t_strerror() — Produce an error message string

Standards

Standards / Extensions	C or C++	Dependencies
XPG4.2	both	

Format

```
#define _XOPEN_SOURCE_EXTENDED 1
#include <xti.h>

char *t_strerror(int errnum);
```

General description

Maps the error number in *errnum* that corresponds to an XTI error to a language-dependent error message string and returns a pointer to the string. The string pointed to should not be modified by the program, but may be overwritten by a subsequent call to the t_strerror function. The string is not terminated by a newline character. If the calling program is operating in any one of the C, POSIX, SAA or S370 locales, then the error message string describing the value in *t_errno* is identical to the comments following the *t_errno* codes defined in <xti.h>. Note that no message number is prefixed to the message text in this situation. If an error code is unknown, and the language is English, t_strerror() returns the string:

```
"<error>: error unknown"
```

where <error> is the error number supplied as input. In other languages, an equivalent text is provided.

Valid states: All - except for T_UNINIT

Returned value

t_strerror() returns a pointer to the generated message string.

Related information

- “xti.h” on page 87
- “t_error() — Produce error message” on page 1869

t_sync() — Synchronize transport library

Standards

Standards / Extensions	C or C++	Dependencies
XPG4.2	both	

Format

```
#define _XOPEN_SOURCE_EXTENDED 1
#include <xti.h>

int t_sync(int fd);
```

General description

Synchronizes the data structures managed by the transport library with information from the underlying transport provider, for the transport endpoint specified by *fd*, In doing so, it can convert an uninitialized file descriptor (obtained using open(), dup(), or as a result of a fork() and exec()) to an initialized transport endpoint, assuming that the file descriptor referenced a transport endpoint, by updating and allocating the necessary library data structures. This function also allows two cooperating processes to synchronize their interaction with a transport provider.

For example, if a process forks a new process and issues an exec(), the new process must issue a t_sync() to build the private library data structure associated with a transport endpoint and to synchronize the data structure with the relevant provider information.

It is important to remember that the transport provider treats all users of a transport endpoint as a single user. If multiple processes are using the same endpoint, they should coordinate their activities so as not to violate the state of the transport endpoint. The function t_sync() returns the current state of the transport endpoint to the user, thereby enabling the user to verify the state before taking further action. This coordination is only valid among cooperating processes. It is possible that a process or an incoming event could change the endpoint's state after a t_sync() is issued.

If the transport endpoint is undergoing a state transition when t_sync() is called, the function will fail.

Valid states: All - except for T_UNINIT

Returned value

If successful, t_sync() returns the state of the transport endpoint. The state returned is one of the following:

T_DATAXFER
Data transfer.

T_IDLE
Idle.

T_INCON
Incoming connection pending.

T_OUTCON
Outgoing connection pending.

T_UNBND
Unbound.

If unsuccessful, t_sync() returns -1 and sets errno to one of the following values:

Error Code
Description

TBADF
The specified file descriptor does not refer to a transport endpoint. This error may be returned when the fd has been previously closed or an erroneous number may have been passed to the call.

TPROTO
This error indicates that a communication problem has been detected between XTI and the transport provider for which there is no other suitable XTI (t_errno).

TSTATECHNG
The transport endpoint is undergoing a state change.

TSYSERR
A system error has occurred during execution of this function.

Related information

- “xti.h” on page 87
- “dup() — Duplicate an open file descriptor” on page 404
- “exec functions” on page 436
- “fork() — Create a new process” on page 571
- “open() — Open a file” on page 1152

tsyncro() — Wait for MTF subtask termination

Standards

Standards / Extensions	C or C++	Dependencies
Language Environment	C only	

Format

```
#include <mtf.h>

int tsyncro(int MTF_ANY|MTF_ALL|nn);
```

General description

Waits for termination of parallel functions under MVS.

To avoid infringing on the user's name space, this nonstandard function has two names. One name is prefixed with two underscore characters, and one name is not. The name without the prefix underscore characters is exposed only when you use LANGLVL(EXTENDED).

To use this function, you must either invoke the function using its external entry point name (that is, the name that begins with two underscore characters), or compile with LANGLVL(EXTENDED). When you use LANGLVL(EXTENDED) any relevant information in the header is also exposed.

The tsyncro() library function causes the main task to wait for the first subtask, a particular subtask, or all subtasks to finish executing all of the parallel functions that have been scheduled for them. You can monitor the completion of any, all, or a specific subtask by specifying:

MTF_ANY
To wait for the completion of any subtask.

MTF_ALL
To wait for the completion of all subtasks.

nn To wait for the completion of the subtask having a *task_id* of *nn*. See “tinit() — Attach and initialize MTF subtasks” on page 1882 for a description of *task_id*.

You can invoke tsyncro() from your main task program as often as necessary.

If tinit() is called by a program running under IMS, CICS, or DB2, the request will not be processed and the returned value will indicate that MTF calls are not supported under these systems.

Note: This function is *not* supported under the z/OS UNIX with the POSIX(ON) runtime option.

Returned value

If successful, tsyncro() will always return a value suitable for use as a target task on a subsequent tsched(). In particular, the return codes on success depend on the nature of the tsyncro() call, and the state of the subtasks at the time of the tsyncro() call as follows:

Task_id passed to tsyncro()	Return code (if successful)
MTF_ANY (and at least one subtask not previously returned from a tsyncro())	*nn* = *task_id* of first subtask to become free
MTF_ANY (and all subtasks have previously been returned from a tsyncro())	MTF_ANY
MTF_ALL	MTF_ANY
nn = *task_id*	*nn*

If tinit() has not been successfully called before the tsyncro() call, tsyncro() indicates that MTF is inactive and returns one of the following values:

Error Code
Description

EINACTIVE
MTF is inactive.

ETASKABND
One or more subtasks have terminated abnormally.

ETASKID
The *task_id* argument specified is out of range.

Note: These values are macros. They can be found in the mtf.h header file.

Related information

- “mtf.h” on page 49
- “tinit() — Attach and initialize MTF subtasks” on page 1882
- “tsched() — Schedule MTF subtask” on page 1918
- “tterm() — Terminate MTF subtasks”

tterm() — Terminate MTF subtasks

Standards

Standards / Extensions	C or C++	Dependencies
Language Environment	C only	

Format

```
#include <mtf.h>

int tterm(void);
```

General description

Restriction: This function is not supported in AMODE 64.

Terminates the MTF environment under MVS. The function is invoked by a main task to await the completion of all parallel functions that were scheduled by tsched() and to detach all subtasks and remove the MTF environment created by tinit(), see “tinit() — Attach and initialize MTF subtasks” on page 1882.

To avoid infringing on the user's name space, this nonstandard function has two names. One name is prefixed with two underscore characters, and one name is not. The name without the prefix underscore characters is exposed only when you use LANGLVL(EXTENDED).

To use this function, you must either invoke the function using its external entry point name (that is, the name that begins with two underscore characters), or compile with LANGLVL(EXTENDED). When you use LANGLVL(EXTENDED) any relevant information in the header is also exposed.

If tinit() has not been successfully called before a tterm() call, tterm() indicates that MTF is already inactive.

If a tterm() call is not issued before main program termination, a system abend with completion code `A03` will occur. The program's termination will terminate the main task while subtasks are still active even if all scheduled parallel functions have completed execution.

If tinit() is called by a program running under IMS, CICS, or DB2, the request will not be processed and the returned value will indicate that MTF calls are not supported under these systems.

Note: This function is *not* supported under the z/OS UNIXwith the POSIX(ON) runtime option.

Returned value

If successful, detaching the subtasks and removing the MTF environment, tterm() returns MTF_OK.

If unsuccessful, tsched() returns one of the following values:

Error Code
  Description

EINACTIVE
  MTF is inactive.

ETASKABND
  One or more subtasks have terminated abnormally.

Note: These values are macros. They can be found in the mtf.h header file (“mtf.h” on page 49).

Related information

- “mtf.h” on page 49
- “tinit() — Attach and initialize MTF subtasks” on page 1882
- “tsched() — Schedule MTF subtask” on page 1918
- “tsyncro() — Wait for MTF subtask termination” on page 1928

ttyname() — Get the name of a terminal

Standards

Standards / Extensions	C or C++	Dependencies
POSIX.1 XPG4 XPG4.2 Single UNIX Specification, Version 3	both	

Format

```
#define _POSIX_SOURCE
#include <unistd.h>

char *ttyname(int fildes);
```

General description

Returns a string containing the path name of the terminal associated with the given file descriptor, *fildes*. Subsequent calls to ttyname() may overwrite this string, because the pointer returned may point to static data.

Returned value

If successful, ttyname() returns a string containing a path name.

If unsuccessful because *fildes* is not a terminal, or the path name cannot be determined, ttyname() returns a NULL pointer.

Special behavior for XPG4: The ttyname() function sets errno to one of the following values:

Error Code
Description

EBADF
The *fildes* argument is not a valid open file descriptor.

ENOTTY
The *fildes* argument is not associated with a terminal.

Example

CELEBT16

```
/* CELEBT16

   This example provides the pathname of the terminal
   associated with stdin.

 */
#define _POSIX_SOURCE
#include <unistd.h>
#include <stdio.h>

main() {
  char *ret, tty[40];

  if ((ret = ttyname(STDIN_FILENO)) == NULL)
    perror("ttyname() error");
  else {
    strcpy(tty, ret);
    printf("The ttyname associated with my stdin is %s\n", tty);
  }
}
```

Output

```
The ttyname associated with my stdin is /dev/ttyp0000
```

Related information

- “unistd.h” on page 82
- “ctermid() — Generate path name for controlling terminal” on page 357
- “isatty() — Test if descriptor represents a terminal” on page 903
- “ttyname_r() — Find path name of a terminal”

ttyname_r() — Find path name of a terminal

Standards

Standards / Extensions	C or C++	Dependencies
Single UNIX Specification, Version 2 Single UNIX Specification, Version 3	both	OS/390 V2R8

Format

```
#define _XOPEN_SOURCE 500
#include <unistd.h>

int ttyname_r(int fildes, char *name, size_t namesize);
```

General description

The ttyname_r() function stores the NULL-terminated path name of the terminal associated with the file descriptor, *fildes*, in the character array referenced by *name*. The array is *namesize* characters long and should have space for the name and the terminating NULL character. The maximum length of the terminal name is **TTY_NAME_MAX**.

Returned value

If successful, ttyname_r() returns 0.

If unsuccessful, ttyname_r() sets errno to one of the following values:

EBADF
The *fildes* argument is not a valid file descriptor.

ENOTTY
The *fildes* argument does not refer to a tty.

ERANGE
The value of *namesize* is smaller than the length of the string to be returned including the terminating NULL character.

Related information

- “unistd.h” on page 82
- “isatty() — Test if descriptor represents a terminal” on page 903
- “ctermid() — Generate path name for controlling terminal” on page 357

- “ttyname() — Get the name of a terminal” on page 1931

ttyslot() — Find the slot in the utmpx file of the current user

Standards

Standards / Extensions	C or C++	Dependencies
XPG4.2	both	

Format

```
#define _XOPEN_SOURCE_EXTENDED 1
#include <stdlib.h>

int ttyslot(void);
```

General description

The ttyslot() function returns the index of the current user's entry in the utmpx database. The current user's entry is an entry for which the ut_line member matches the name of a terminal device associated with any of the process's file descriptors 0, 1 or 2. The ttyname() function is used to obtain the terminal device. The "/dev/" part returned by ttyname() is not used when searching the utmpx database member ut_line.

Note:

This function is kept for historical reasons. It was part of the Legacy Feature in Single UNIX Specification, Version 2, but has been withdrawn and is not supported as part of Single UNIX Specification, Version 3.

If it is necessary to continue using this function in an application written for Single UNIX Specification, Version 3, define the feature test macro _UNIX03_WITHDRAWN before including any standard system headers. The macro exposes all interfaces and symbols removed in Single UNIX Specification, Version 3.

Returned value

If successful, ttyslot() returns the index of the current user's entry in the utmpx database.

If unsuccessful, ttyslot() returns -1 if an error was encountered while searching the database or if none of the file descriptors 0, 1, or 2 is associated with a terminal device.

No errors are defined.

Related information

- “stdlib.h” on page 70
- “endutxent() — Close the utmpx database” on page 425
- “ttyname() — Get the name of a terminal” on page 1931

t_unbind() — Disable a transport endpoint

Standards

Standards / Extensions	C or C++	Dependencies
XPG4.2	both	

Format

```
#define _XOPEN_SOURCE_EXTENDED 1
#include <xti.h>

int t_unbind(int fd);
```

General description

Disables the transport endpoint specified by *fd* which was previously bound by t_bind() . On completion of this call, no further data or events destined for this transport endpoint will be accepted by the transport provider. An endpoint which is disabled by using t_unbind() can be enabled by a subsequent call to t_bind() .

Due to implementation-imposed restrictions, t_unbind does not affect descriptors in processes other than the caller which were derived from *fd* by normal descriptor inheritance. Processes cooperating on an endpoint in this way must explicitly provide their own synchronization for endpoint takedown.

Valid states: T_IDLE

Returned value

If successful, t_unbind() returns 0.

If unsuccessful, t_unbind() returns -1 and sets errno to one of the following values:

Error Code
: **Description**

TBADF
: The specified file descriptor does not refer to a transport endpoint.

TLOOK
: An asynchronous event has occurred on this transport endpoint.

TOUTSTATE
: The function was issued in the wrong sequence.

TPROTO
: This error indicates that a communication problem has been detected between XTI and the transport provider for which there is no other suitable XTI (t_errno).

TSYSERR
: A system error has occurred during execution of this function.

Related information

- "xti.h" on page 87
- "t_bind() — Bind an address to a transport endpoint" on page 1821

twalk() — Binary tree walk

Standards

Standards / Extensions	C or C++	Dependencies
XPG4 XPG4.2 Single UNIX Specification, Version 3	both	

Format

```
#define _XOPEN_SOURCE
#include <search.h>

void *twalk(const void *root, void (*action)(const void *, VISIT, int));
```

General description

The twalk() function traverses a binary search tree. The *root* argument is a pointer to the root node of the tree to be traversed. (Any node in a tree may be used as the root for a walk below that node.) The argument *action* is the name of a routine to be invoked at each node. This routine is, in turn, called with three arguments. The first argument is the address of the node being visited. The structure pointed to by this argument is unspecified and must not be modified by the application, but is guaranteed that a pointer-to-node can be converted to pointer-to-pointer-to-element to access the element stored in the node. The second argument is a value from an enumeration data type:

```
typedef enum {preorder, postorder, endorder, leaf } VISIT;
```

(defined in the <search.h> header), depending on whether this is the first, second or third time that the node is visited (during a depth-first, left-to-right traversal of the tree), or whether the node is a leaf. The third argument is the level of the node in the tree, with the root being level zero.

Threading Behavior: see “tsearch() — Binary tree search” on page 1920.

Special behavior for C++: Because C and C++ linkage conventions are incompatible, twalk() cannot receive a C++ function pointer as the argument. If you attempt to pass a C++ function pointer to twalk(), the compiler will flag it as an error. You can pass a C or C++ function to twalk() by declaring it as extern “C”.

Returned value

twalk() returns no values.

No errors are defined.

Related information

- “search.h” on page 62
- “bsearch() — Search arrays” on page 220
- “hsearch() — Search hash tables” on page 817
- “lsearch() — Linear search and update” on page 1019
- “tdelete() — Binary tree delete” on page 1865
- “tfind() — Binary tree find node” on page 1870
- “tsearch() — Binary tree search” on page 1920

tzset() — Set the time zone

Standards

Standards / Extensions	C or C++	Dependencies
POSIX.1 XPG4 XPG4.2 z/OS UNIX Single UNIX Specification, Version 3	both	

Format

```
#define _POSIX_SOURCE
#include <time.h>

void tzset(void);
```

General description

The tzset() function uses the value of the environment variable TZ to set time conversion information used by ctime(), localtime(), mktime(), and strftime(). If TZ is absent from the environment, or it is incorrect, time-conversion information is obtained from the LC_TOD locale category.

tzset() also sets the external variable *tzname*:

```
extern char *tzname[2] = {"std", "dst"};
```

Here, *std* and *dst* are Standard and Daylight Savings time zone names, specified by TZ or the LC_TOD local category, respectively.

tzset() is called by ctime(), localtime(), mktime(), setlocale(), and strftime(). It can also be called explicitly by an application program.

The format of TZ values recognized by tzset() is as follows:

```
stdoffset[dst[offset][,rule]]
```

std **and** *dst*
: Indicate no fewer than three, but not more than TZNAME_MAX, bytes that are the designation for the standard (*std*) and daylight savings (*dst*) time zones. If more than TZNAME_MAX bytes are specified for *std* or *dst*, tzset() truncates to TZNAME_MAX bytes. Only *std* is required; if *dst* is missing, daylight savings time does not apply in this locale. Uppercase and lowercase letters are explicitly allowed. Any character except a leading colon (**:**) or digits, the comma (**,**), the minus (-), the plus (+), and the NULL character are permitted to appear in these fields. The meaning of these letters and characters is unspecified.

offset
: Indicates the value that must be added to the local time to arrive at Coordinated Universal Time (UTC). *offset* has the form: `hh[:mm[:ss]]` The minutes (mm) and seconds (ss) are optional. The hour (hh) is required and may be a single digit. *offset* following *std* is required. If no *offset* follows *dst*, daylight savings time is assumed to be 1 hour ahead of standard time. One or more digits may be used; the value is always interpreted as a decimal number. The hour must be between 0 and 24; minutes and seconds, if present, between 0 and 59. The difference between standard time *offset* and daylight savings time *offset* must be greater than or equal to 0, but the

difference may not be greater than 24 hours. Use of values outside of these ranges causes tzset() to use the LC_TOD category rather than the TZ environment variable for time conversion information. An *offset* preceded by a minus (-) indicates a time zone east of the Prime Meridian. A plus (+) preceding *offset* is optional and indicates the time zone west of the Prime Meridian.

rule Indicates when to change to and back from daylight savings time. The rule has the form: `date[/time],date[/time]`

The first date describes when the change from standard to daylight savings time occurs and the second date describes when the change back happens. Each time field describes when, in current local time, the change to the other time is made.

The format of date must be one of the following:

Jn The Julian day *n* (1≤*n*≤365). Leap days are not counted. That is, in all years—including leap years—February 28 is day 59 and March 1 is day 60. It is impossible to explicitly refer to the occasional February 29.

n The zero-based Julian day (0≤*n*≤365). Leap days are counted, and it is possible to refer to February 29.

Mm.n.d
The *d*th day (0≤*d*≤6) of week *n* of month *m* of the year (1≤*n*≤5, and 1≤*m*≤12, where week 5 means " the last *d* day in month *m*," which may occur in either the fourth or the fifth week). Week 1 is the first week in which the *d*th day occurs. Day zero is Sunday.

The time has the same format as offset except that no leading sign, minus (-) or plus (+), is allowed. The default, if time is not given, is 02:00:00.

If *dst* is specified and *rule* is not specified by TZ or in LC_TOD category, the default for the daylight savings time start date is M4.1.0 and for the daylight savings time end date is M10.5.0.

Special behavior for XPG4: The tzset() function sets the external variable `timezone` to the difference, in seconds, between Coordinated Universal Time (UTC) and local standard time. tzset() sets the external variable `daylight` to 0 if summer time conversions should never be applied for the time zone in use; otherwise to nonzero.

Since the external variables `timezone` and `daylight` are global to the process, they cannot be reliably used in a multithreaded application or an application running from a DLL. The runtime library provides two special functions, __tzone() and __dlght(), which return the address of thread-specific versions of these external variables.

Special behavior for POSIX C: The tzset() function only parses the TZ environment variable if it is called from the initial processing thread (IPT) by a threaded application.

Note: This function is sensitive to time zone information which is provided by:

- The TZ environmental variable when POSIX(ON) and TZ is correctly defined, or by the _TZ environmental variable when POSIX(OFF) and _TZ is correctly defined.
- The LC_TOD category of the current locale if POSIX(OFF) or TZ is not defined.

- When neither TZ nor _TZ is defined, the current locale is interrogated for time zone information. If neither TZ nor _TZ is defined and LC_TOD time zone information is not present in the current locale, a default value is applied to local time. POSIX programs simply default to Coordinated Universal Time (UTC), while non-POSIX programs establish an offset from UTC based on the setting of the system clock. For more information about customizing a time zone to work with local time, see "Customizing a time zone" in *z/OS XL C/C++ Programming Guide*.

The time zone external variables `tzname`, `timezone`, and `daylight` declarations remain feature test protected in time.h.

Returned value

tzset() returns no values.

There are no documented errno values.

Example

CELEBT17

```
/* CELEBT17

   This example set time conversion information for
   Eastern Standard and Eastern Daylight Savings Time in the
   United States.

 */
#define _POSIX_SOURCE
#include <env.h>
#include <time.h>

int main(void)
{

   setenv("TZ", "EST5EDT", 1);
   tzset();
}
```

Related information

- For more information about external variables and associated restrictions, see "External variables" on page 96.
- "time.h" on page 79
- "asctime(), asctime64() — Convert time to character string" on page 182
- "asctime_r(), asctime64_r() — Convert date and time to a character string" on page 184
- "ctime(), ctime64() — Convert time to character string" on page 359
- "ctime_r(), ctime64_r() — Convert time value to date and time character string" on page 362
- "gmtime(), gmtime64() — Convert time to broken-down UTC time" on page 810
- "gmtime_r(), gmtime64_r() — Convert a time value to broken-down UTC time" on page 812
- "localdtconv() — Date and time formatting convention inquiry" on page 981
- "localtime(), localtime64() — Convert time and correct for local time" on page 984
- "localtime_r(), localtime64_r() — Convert time value to broken-down local time" on page 986
- "mktime(), mktime64() — Convert local time" on page 1078

- “strftime() — Convert to formatted time” on page 1743
- “time(),time64() — Determine current UTC time” on page 1878

ualarm() — Set the interval timer

Standards

Standards / Extensions	C or C++	Dependencies
XPG4.2 Single Unix Standard, Version 3	both	POSIX(ON)

Format

```
#define _XOPEN_SOURCE_EXTENDED 1
#include <unistd.h>

useconds_t ualarm(useconds_t uscs, useconds_t intrval)
```

General description

The ualarm() function causes the SIGALRM signal to be generated for the calling process after the number of real-time microseconds specified by the *uscs* argument has elapsed. When the *intrval* argument is nonzero, repeated timeout notification occurs with a period in microseconds specified by the *intrval* argument. If the notification signal, SIGALRM, is not caught or ignored, the calling process is terminated.

The ualarm() function is a simplified interface to setitimer() and uses the ITIMER_REAL interval timer.

Note: The ualarm() and usleep() functions have been moved to obsolescence in Single UNIX Specification, Version 3 and may be withdrawn in a future version. The setitimer(), timer_create(), timer_delete(), timer_getoverrun(), timer_gettime(), or timer_settime() functions are preferred for portability.

Returned value

ualarm() returns the number of microseconds remaining from the previous ualarm(), alarm(), or setitimer(ITIMER_REAL) call.

If no timeouts are pending, ualarm() returns 0.

No errnos are defined for the ualarm() function.

Related information

- “unistd.h” on page 82
- “alarm() — Set an alarm” on page 156
- “setitimer() — Set value of an interval timer” on page 1544
- “sigaction() — Examine or change a signal action” on page 1611
- “sleep() — Suspend execution of a thread” on page 1678
- “usleep() — Suspend execution for an interval” on page 1966

__ucreate() — Create a heap using user-provided storage

Standards

Standards / Extensions	C or C++	Dependencies
Language Environment	both	

Format

```
#include <uheap.h>

__uheapid_t __ucreate(void *block,
                      size_t size,
                      __uheap_cellpool_attrib_table_t *cellpool_attrib_table,
                      void *rsvd1,
                      void *rsvd2,
                      void *rsvd3,
                      void *rsvd4);
```

General description

The __ucreate() function creates a heap out of storage that is provided by the caller. The heap is divided up into cell pools based on the information provided in the cellpool_attrib_table. Up to 12 cell pools can be created within the heap. Note that this is a fixed-size heap; when storage within a given cell pool is exhausted, no additional storage will be allocated. __ucreate() returns a uheapid that is used to identify the heap on subsequent user-created heap function calls, such as __umalloc(), __ufree(), and __uheapreport() calls.

Parameter
Description

block A pointer to the storage which is to be used for the heap.

size The size of the block of storage. Note that Language Environment reserves approximately 328 bytes of this storage for use in allocating heap management control blocks. Additional storage is reserved if storage report usage statistics are being collected for the heap. The amount of this storage is related to the largest cell size and the granularity of the statistics, and is calculated as: storage amount = ((largestcellsize+granularity-1)/ granularity)*4.

cellpool_attrib_table
A pointer to a structure describing the attributes of the cell pools to be created by __ucreate().

The first field of the structure, number_of_pools, indicates the number of cell pools to be created. Up to 12 cell pools can be created in the heap.

The second field of the structure, granularity, indicates the granularity to which storage usage statistics is to be collected. This value must be zero, or a power of 2 greater than or equal to 8. If the value is zero, then statistics are not collected.

Following these words are pairs of words describing the attributes of each cell pool in the heap:

The first field in the pair, size, is the size of the cell in the cell pool. The cell size must be a multiple of 8 and greater than or equal to 8, up to a maximum of 64K (65536). Note that Language Environment adds an additional 8 bytes to the size of the cell for use in managing the cells. The second field in the pair, count, is the number of cells of this size to be allocated. Note the minimum is four.

rsvd1-rsvd4
Reserved for future use.

Returned value

If successful, __ucreate() returns a uheapid.

If unsuccessful, __ucreate() returns -1 and sets errno to EINVAL.

Related information

- "uheap.h" on page 82
- "__ufree() — Return storage to a user-created heap"
- "__uheapreport() — Produce a storage report for a user-created heap" on page 1942
- "__umalloc() — Allocate storage from a user-created heap" on page 1946

__ufree() — Return storage to a user-created heap

Standards

Standards / Extensions	C or C++	Dependencies
Language Environment	both	

Format

```
#include <uheap.h>

void __ufree(__uheapid_t heapid, void *ptr);
```

General description

The __ufree() function returns storage to the heap identified by the heapid. If the returned storage does not belong to the given heap, the result is unpredictable.

Parameter
Description

heapid The identifier of the user-created heap to which the storage is to be returned.

ptr A pointer to the storage to be returned to the heap.

Returned value

__ufree() returns no values.

Related information

- "uheap.h" on page 82
- "__ucreate() — Create a heap using user-provided storage" on page 1940
- "__uheapreport() — Produce a storage report for a user-created heap" on page 1942
- "__umalloc() — Allocate storage from a user-created heap" on page 1946

__uheapreport() — Produce a storage report for a user-created heap

Standards

Standards / Extensions	C or C++	Dependencies
Language Environment	both	

Format

```
#include <uheap.h>

void __uheapreport(__uheapid_t heapid);
```

General description

The __uheapreport() function generates a report of the storage used within the user-created heap identified by *heapid*. The report is directed to the ddname specified in the MSGFILE runtime option except for AMODE 64 applications, in which case the report is directed to the stderr stream. The report format is similar to the heap pools portion of the storage report generated for the RPTSTG runtime option.

Statistics for the user-created heap will only be collected if the granularity field of the cellpool_attrib_table passed to __ucreate() is nonzero and a valid value.

Parameter
Description

heapid The identifier of the user-created heap for which a report is to be produced.

Returned value

__uheapreport() returns no values.

Related information

- "uheap.h" on page 82
- "__ucreate() — Create a heap using user-provided storage" on page 1940
- "__ufree() — Return storage to a user-created heap" on page 1941
- "__umalloc() — Allocate storage from a user-created heap" on page 1946

ulimit() — Get or set process file size limits

Standards

Standards / Extensions	C or C++	Dependencies
XPG4 XPG4.2 Single UNIX Specification, Version 3	both	

Format

```
#define _XOPEN_SOURCE
#include <ulimit.h>

long int ulimit(int cmd, ...);
```

General description

The ulimit() function provides control over the process file size limits. The *cmd* argument controls whether the file size limits are obtained or set. The *cmd* argument can be one of the following, defined in <ulimit.h>:

UL_GETFSIZE
Return the soft (current) file size limit of the process. The limit returned is in units of 512-byte blocks. The return value is the integer portion of the soft file size limit divided by 512. Refer to the setrlimit() function, RLIMIT_FSIZE resource description for more detail.

UL_SETFSIZE
Set the hard (maximum) and soft (current) file size limits for output operations of the process. The value of the second argument is used, and is treated as a long int. Refer to the setrlimit() function, RLIMIT_FSIZE resource description for more detail and restrictions on lowering and raising file size limits. The hard and soft file size limits are set to the specified value multiplied by 512. The new file size limit (in 512 byte increments) is returned.

Returned value

If successful, ulimit() returns the value of the requested limit.

If unsuccessful, ulimit() returns -1 and sets errno to one of the following values:

Error Code
Description

EINVAL
The *cmd* argument is not valid.

EPERM
To increase the file size limit, superuser authority is required.

Related information

- “ulimit.h” on page 82
- “getrlimit() — Get current or maximum resource consumption” on page 765
- “setrlimit() — Control maximum resource consumption” on page 1573

ulltoa() — Convert unsigned long long into a string

Standards

Standards / Extensions	C or C++	Dependencies
z/OS UNIX	both	z/OS V1R5

Format

```
#define _OPEN_SYS_ITOA_EXT
#include <stdlib.h>

char * ulltoa(uint64_t ll, char * buffer, int radix);
```

Compile Requirement

Use of this function requires the long long data type. See *z/OS XL C/C++ Language Reference* for information on how to make long long available.

General description

The ulltoa() function coverts the uint64_t ll into a character string. The string is placed in the buffer passed, which must be large enough to hold the output. The radix values can be OCTAL, DECIMAL, or HEX. When the radix is DECIMAL, ulltoa() produces the same result as the following statement:

```
(void) sprintf(buffer, "%llu", ll);
```

with buffer the returned character string. When the radix is OCTAL, ulltoa() formats uint64_t ll into an octal constant. When the radix is HEX, ulltoa() formats uint64_t ll into a hexadecimal constant. The hexadecimal value will include lower case abcdef, as necessary.

Returned value

String pointer (same as buffer) will be returned. When passed an invalid radix argument, function will return NULL and set errno to EINVAL.

Portability considerations

This is a non-standard function. Even though the prototype given is commonly used by compilers on other platforms, there is no guarantee that this function will behave the same on all platforms, in all cases. You can use this function to help port applications from other platforms, but you should avoid using it when writing new applications, in order to ensure maximum portability.

Example

CELEBU12

```
/* CELEBU12

   This example reads uint64_t and formats it to a decimal,
   octal, hexadecimal constants converted to a character string.

*/

#define _OPEN_SYS_ITOA_EXT
#include <stdio.h>
#include <stdlib.h>

int main ()
{
   uint64_t i;
   char buffer [sizeof(uint64_t)*8+1];
   printf ("Enter a number: ");
   if (scanf ("%llu",&i) == 1) {
      ulltoa (i,buffer,DECIMAL);
      printf ("decimal: %s\n",buffer);
      ulltoa (i,buffer,HEX);
      printf ("hexadecimal: %s\n",buffer);
      ulltoa (i,buffer,OCTAL);
      printf ("octal: %s\n",buffer);
   }
   return 0;
}
```

Output

If the input is 1234, then the output should be:

```
decimal: 1234
hexadecimal: 4d2
octal: 2322
```

Related information

- “stdlib.h” on page 70
- “itoa() — Convert int into a string” on page 925
- “lltoa() — Convert long long into a string” on page 979
- “ltoa() — Convert long into a string” on page 1027
- “ultoa() — Convert unsigned long into a string”
- “utoa() — Convert unsigned int into a string” on page 1972

ultoa() — Convert unsigned long into a string

Standards

Standards / Extensions	C or C++	Dependencies
z/OS UNIX	both	z/OS V1R5

Format

```
#define _OPEN_SYS_ITOA_EXT
#include <stdlib.h>

char * ultoa(unsigned long l, char * buffer, int radix);
```

General description

The ultoa() function coverts the unsigned long l into a character string. The string is placed in the buffer passed, which must be large enough to hold the output. The radix values can be OCTAL, DECIMAL, or HEX. When the radix is DECIMAL, ultoa() produces the same result as the following statement:

```
(void) sprintf(buffer, "%lu", l);
```

with buffer the returned character string. When the radix is OCTAL, ultoa() formats unsigned long l into an octal constant. When the radix is HEX, ultoa() formats unsigned long l into a hexadecimal constant. The hexadecimal value will include lower case abcdef, as necessary.

Returned value

String pointer (same as buffer) will be returned. When passed an invalid radix argument, function will return NULL and set errno to EINVAL.

Portability considerations

This is a non-standard function. Even though the prototype given is commonly used by compilers on other platforms, there is no guarantee that this function will behave the same on all platforms, in all cases. You can use this function to help port applications from other platforms, but you should avoid using it when writing new applications, in order to ensure maximum portability.

Example

CELEBU11

```
/* CELEBU11

   This example reads an unsigned long and formats it to a decimal,
   octal, hexadecimal constants converted to a character string.

*/

#define _OPEN_SYS_ITOA_EXT
#include <stdio.h>
#include <stdlib.h>

int main ()
{
   unsigned long i;
   char buffer [sizeof(unsigned long)*8+1];
   printf ("Enter a number: ");
   if (scanf ("%lu",&i) == 1) {
      ultoa (i,buffer,DECIMAL);
      printf ("decimal: %s\n",buffer);
      ultoa (i,buffer,HEX);
      printf ("hexadecimal: %s\n",buffer);
      ultoa (i,buffer,OCTAL);
      printf ("octal: %s\n",buffer);
   }
   return 0;
}
```

Output

If the input is 1234, then the output should be:

```
decimal: 1234
hexadecimal: 4d2
octal: 2322
```

Related information

- “stdlib.h” on page 70
- “itoa() — Convert int into a string” on page 925
- “lltoa() — Convert long long into a string” on page 979
- “ltoa() — Convert long into a string” on page 1027
- “ulltoa() — Convert unsigned long long into a string” on page 1943
- “utoa() — Convert unsigned int into a string” on page 1972

__umalloc() — Allocate storage from a user-created heap

Standards

Standards / Extensions	C or C++	Dependencies
Language Environment	both	

Format

```
#include <uheap.h>

void *__umalloc(__uheapid_t heapid, size_t size);
```

General description

The __umalloc() function allocates storage from the heap identified by the heapid. __umalloc() will search for an available cell within the cell pool that contains cells at least as large and closest in size to the requested size.

Parameter
Description

heapid The identifier of the user-created heap from which the storage is to be allocated.

size The amount of storage to be allocated.

Returned value

If successful, __umalloc() returns a pointer to the reserved cell.

If a cell of the required size is not available, if size was larger than the largest available cell size, or if size was specified as 0, __umalloc() returns NULL.

If there is not enough storage or if the requested size was too large, __umalloc() returns NULL and sets errno to one of the following values:

Error Code
Description

E2BIG Requested amount of storage is larger than the largest available cell size

ENOMEM
Insufficient memory is available

Related information

- "uheap.h" on page 82
- "__ucreate() — Create a heap using user-provided storage" on page 1940
- "__ufree() — Return storage to a user-created heap" on page 1941
- "__uheapreport() — Produce a storage report for a user-created heap" on page 1942

umask() — Set and retrieve file creation mask

Standards

Standards / Extensions	C or C++	Dependencies
POSIX.1 XPG4 XPG4.2 Single UNIX Specification, Version 3	both	

Format

```
#define _POSIX_SOURCE
#include <sys/stat.h>

mode_t umask(mode_t newmask);
```

General description

Changes the file creation mask of the process. *newmask* specifies new file-permission bits for the file creation mask of the process.

This mask restricts the setting of (or turns off) file-permission bits specified in the *mode* argument that is used with all `open()`, `creat()`, `mkdir()`, and `mkfifo()`

functions issued by the current process. File-permission bits set to 1 in the file creation mask are set to 0 in the file-permission bits of files that are created by the process.

For example, if a call to `open()` specifies a *mode* argument with file-permission bits, the file creation mask of the process affects the *mode* argument; bits that are 1 in the mask are set to 0 in the *mode* argument, and therefore in the mode of the created file.

Only the file-permission bits of the new mask are used. The meaning of other bits is implementation-defined. For more information on these symbols, refer to "chmod() — Change the mode of a file or directory" on page 272.

The _EDC_UMASK_DFLT environment variable controls how the C runtime library sets the default umask if the calling program was not started by one of the `exec` or `spawn` functions.

- When _EDC_UMASK_DFLT is not set, if z/OS UNIX BPXPRMxx statement UMASK is set to a valid value, the default umask value is the same as that value; otherwise, the C runtime library establishes a default umask value of 022.
- When _EDC_UMASK_DFLT is set to one of the following values:

  NO (case insensitive)
  The C runtime library does not change the default umask value, which depends on how BPXPRMxx statement UMASK is set.

  A valid octal value
  The library uses this octal value as the default umask value. The BPXPRMxx statement UMASK is overridden by this value.

  Any other value
  If the BPXPRMxx statement UMASK is set to a valid value, the default umask value is the same as that value; otherwise, the C runtime library establishes a default umask value of 022.

Returned value

`umask()` is always successful and returns the previous value of the file creation mask.

There are no documented `errno` values.

Example

CELEBU01

```
/* CELEBU01

   This example will work only from C/MVS, not C++/MVS.

 */
#define _POSIX_SOURCE
#include <fcntl.h>
#include <sys/stat.h>
#include <sys/types.h>
#include <unistd.h>
#undef _POSIX_SOURCE
#include <stdio.h>

main() {
  int fd;
  mode_t oldmask;

  printf("Your old umask is %i\n",oldmask=umask(S_IRWXG));
  if ((fd = creat("umask.file", S_IRWXU|S_IRWXG)) < 0)
    perror("creat() error");
  else {
    system("ls -l umask.file");
    close(fd);
    unlink("umask.file");
  }
  umask(oldmask);
}
```

Output

```
-rwx------   1 WELLIE   SYS1          0 Apr 19 14:50 umask.file
```

Related information

- “sys/stat.h” on page 75
- “chmod() — Change the mode of a file or directory” on page 272
- “creat() — Create a new file or rewrite an existing one” on page 340
- “mkdir() — Make a directory” on page 1068
- “mkfifo() — Make a FIFO special file” on page 1071
- “open() — Open a file” on page 1152

umount() — Remove a virtual file system

Standards

Standards / Extensions	C or C++	Dependencies
z/OS UNIX	both	

Format

```
#include <sys/stat.h>

int umount(const char *filesystem, mtm_t mtm);
```

General description

Removes a file system from the file hierarchy, or changes the mount mode of a mounted file system between read-only and read/write. The *filesystem* argument is a NULL-terminated string containing the file-system name. This is the same name that was specified when the file system was mounted.

In order to umount a file system, the caller must be an authorized program, or must be running for a user with appropriate privileges.

The *mtm* argument can be one of the following:

MTM_UMOUNT
A normal unmount request. If the files in the named file system are not in use, the unmount is done. Otherwise, the request is rejected.

MTM_DRAIN
An unmount drain request. The requester is willing to wait for all uses of this file system to be ended normally before the file system is unmounted.

MTM_IMMED
An immediate unmount request. The file system is unmounted

immediately, forcing any users of files in the specified file system to fail. All data changes that were made up to the time of the request are saved. If there is a problem saving the data, the unmount request fails.

MTM_FORCE
A forced unmount request. The file system is unmounted immediately, forcing any users of any files in the specified file system to fail. All data changes that were made up to the time of the request are saved. If there is a problem saving the data, the request continues, and the data may be lost. To prevent lost data, issue an immediate umount() request before issuing a forced umount() request.

MTM_RESET
A reset unmount request. This allows a previous unmount drain request to be stopped.

MTM_REMOUNT
A remount request. This changes the mount mode of a file system from read-only to read/write or from read/write to read-only. If neither MTM_RDONLY nor MTM_RDWR is specified, the mode is set to the opposite of its current state. If a mode is specified, it must be the opposite of the current state.

MTM_SAMEMODE
A remount request to unmount and mount back without changing the mount mode. If either MTM_RDONLY or MTM_RDWR is also specified, it must be the current state. This can be used to attempt to regain use of a file system that has had I/O errors.

Returned value

If successful, umount() returns 0.

If unsuccessful, umount() returns -1 and sets errno to one of the following values:

Error Code
Description

EBUSY
The file system is busy, for one of these reasons:

- A umount() (MTM_UMOUNT) was requested, and the file system still has open files or other file systems mounted under it.
- A file system is currently mounted on the requested file system.
- A RESET was requested, and the previous umount() request was either immediate or forced, instead of a drain request.
- There is a umount() request already in progress for the specified file system.
- A umount() drain request is being reset.

EINTR
umount() was interrupted by a signal.

EINVAL
A parameter was incorrectly specified. Verify the spelling of the file-system name and the setting of *mtm*.

EIO An I/O error occurred.

EPERM
Superuser authority is required to issue an unmount.

Example

CELEBU02

```
/* CELEBU02

   This example removes a file, using umount().

 */
#define _OPEN_SYS 1
#include <sys/stat.h>
#include <stdio.h>
#include <unistd.h>

main() {
  char HFS[]="POSIX.NEW.HFS";
  char filesystype[9]="HFS     ";
  setvbuf(stdout, NULL, _IOLBF, 0);
  puts("before umount()");
  system("df -Pk");
  if (umount(HFS, MTM_UMOUNT) != 0)
    perror("umount() error");
  else {
    puts("After umount()");
    system("df -Pk");
  }
}
```

Output

```
before umount()
Filesystem   1024-blocks        Used  Available  Capacity     Mounted on
POSIX.NEW.HFS        200          20        180       10%     /new_fs
POSIX.ROOT.FS       9600        8180       1420       85%     /

After umount()
Filesystem   1024-blocks        Used  Available  Capacity     Mounted on
POSIX.ROOT.FS       9600        8180       1420       85%     /
```

Related information

- “sys/stat.h” on page 75
- “mount() — Make a file system available” on page 1092

uname() — Display current operating system name

Standards

Standards / Extensions	C or C++	Dependencies
POSIX.1 XPG4 XPG4.2 Single UNIX Specification, Version 3	both	

Format

```
#define _POSIX_SOURCE
#include <sys/utsname.h>

int uname(struct utsname *name);
```

General description

The uname() function retrieves information identifying the operating system you are running on. The argument *name* points to a memory area where a structure describing the operating system the process is running on can be stored.

The information about the operating system is returned in a `utsname` structure, which has the following elements:

char *sysname;
The name of the implementation of the operating system.

char *nodename;
The node name of this particular machine. The node name is set by the SYSNAME sysparm (specified at IPL), and usually differentiates machines running at a single location.

char *release;
The current release level of the implementation.

char *version;
The current version level of the release.

char *machine;
The name of the hardware type the system is running on.

Each of the `utsname` structure elements is a normal C string, terminated with a NULL character.

As of OS/390 Release 2, the uname() function will return "OS/390" as the *sysname* value, even if the true name of the operating system is different. This is for compatibility purposes. The version value will increase at every new version of the operating system. The release value will increase at every new release of the operating system.

Table 60 lists the operating system names and corresponding values returned by the uname() function. To retrieve the true operating system name, version, and release, use the __osname() function.

Table 60. Operating system information returned by the uname() function

Operating system	Sysname	Release	Version
z/OS V2.2	OS/390	25.00	04
z/OS V2.1	OS/390	24.00	04
z/OS V1.13	OS/390	23.00	03
z/OS V1.12	OS/390	22.00	03
z/OS V1.11	OS/390	21.00	03
z/OS V1.10	OS/390	20.00	03
z/OS V1.9	OS/390	19.00	03
z/OS V1.8	OS/390	18.00	03
z/OS V1.7	OS/390	17.00	03
z/OS V1.6	OS/390	16.00	03
z/OS V1.5	OS/390	15.00	03
z/OS V1.4	OS/390	14.00	03
z/OS V1.3	OS/390	13.00	03
z/OS V1.2	OS/390	12.00	03
z/OS V1.1	OS/390	11.00	03
OS/390 V2.10	OS/390	10.00	02
OS/390 V2.9	OS/390	09.00	02
OS/390 V2.8	OS/390	08.00	02

Table 60. Operating system information returned by the uname() function (continued)

Operating system	Sysname	Release	Version
OS/390 V2.7	OS/390	07.00	02
OS/390 V2.6	OS/390	06.00	02
OS/390 V2.5	OS/390	05.00	02
OS/390 V2.4	OS/390	04.00	02
OS/390 V1.3	OS/390	03.00	01
OS/390 V1.2	OS/390	02.00	01
OS/390 V1.1	MVS	100	1
	MVS	2.2	5

Returned value

If successful, the uname() function returns a nonnegative value.

If unsuccessful, the uname() function returns -1. An errno might be set to indicate the reason for the failure, but no errno values are specified by the POSIX.1 standard.

Example

CELEBU03

```
/* CELEBU03

   This example gets information about the system you are running on.

 */
#define _POSIX_SOURCE
#include <sys/utsname.h>
#include <stdio.h>

main() {
  struct utsname uts;

  if (uname(&uts) < 0)
    perror("uname() error");
  else {
    printf("Sysname:  %s\n", uts.sysname);
    printf("Nodename: %s\n", uts.nodename);
    printf("Release:  %s\n", uts.release);
    printf("Version:  %s\n", uts.version);
    printf("Machine:  %s\n", uts.machine);
  }
}
```

Output

```
Sysname:  OS/390
Nodename: SY1
Release:  25.00
Version:  04
Machine:  2097
```

Related information

- “sys/utsname.h” on page 77
- “__osname() — Get true operating system name” on page 1166

uncaught_exception() — Determine if an exception is being processed

Standards

Standards / Extensions	C or C++	Dependencies
ANSI/ISO C++	C++ only	z/OS V1R2

Format

```
#include <exception>

bool uncaught_exception(void);
```

General description

The uncaught_exception() function returns true only if a thrown exception is currently being processed.

When uncaught_exception() is true, throwing an exception can result in a call of terminate().

Returned value

uncaught_exception() returns true after completing evaluation of a throw expression and before completing initialization of the exception declaration in the matching handler or calling unexpected() as a result of the throw expression.

Otherwise, uncaught_exception() returns false.

Example

```
#include <exception>
#include <iostream.h>

using namespace std;

class X
{
  public:
       ~X ();
};

X::~X()
{
  if (uncaught_exception ())
     printf (" X::~X called during stack unwind\n");
  else
     printf (" X::~X called normally\n");
}

int main()
{
   X x1;
   try
   {
      X x2;
      throw 1;
   }
   catch (...)  { /*...*/ }
   return 0;
}
// under a Standard-conforming implementation, this program yields
//  X::~X called during stack unwind
//  X::~X called normally
```

Related information

- "exception" on page 26
- "set_terminate() — Register a function for terminate()" on page 1590
- "set_unexpected() — Register a function for unexpected()" on page 1594
- "terminate() — Terminate after failures in C++ error handling" on page 1868
- "unexpected() — Handle exception not listed in exception specification" on page 1957

UnDoExportWorkUnit() — WLM undo export service

Standards

Standards / Extensions	C or C++	Dependencies
z/OS UNIX	both	OS/390 V2R9

Format

```
#include <sys/__wlm.h>

int UnDoExportWorkUnit(wlmxtok_t *exporttoken, unsigned long *conntoken);
```

AMODE 64:

```
#include <sys/__wlm.h>

int UnDoExportWorkUnit(wlmxtok_t *exporttoken, unsigned int *conntoken);
```

General description

Undoes an earlier request to export an enclave using the ExportWorkUnit() function. The caller is expected to invoke UnDoExportWorkUnit() after all importing systems have invoked the UnDoImportWorkUnit() function.

The UnDoExportWorkUnit() function uses the following parameters:

**exporttoken*
: Points to a work unit export token that was returned from a call to ExportWorkUnit().

**conntoken*
: Specifies the connect token that represents the connection to WLM.

Returned value

If successful, UnDoExportWorkUnit() returns 0.

If unsuccessful, UnDoExportWorkUnit() returns -1 and sets errno to one of the following values:

Error Code
: **Description**

EFAULT
: An argument of this function contained an address that was not accessible to the caller.

EINVAL
An argument of this function contained a value that is not correct.

EMVSSAF2ERR
An error occurred in the security product.

EMVSWLMERROR
A WLM service failed. Use __errno2() to obtain the WLM service reason code for the failure.

EPERM
The calling thread's address space is not permitted to the BPX.WLMSERVER Facility class. The caller's address space must be permitted to the BPX.WLMSERVER Facility class, if the BPX.WLMSERVER class is defined. If BPX.WLMSERVER is not defined, the calling process is not defined as a superuser (UID=0).

Related information

- "sys/__wlm.h" on page 77
- "ExportWorkUnit() — WLM export service" on page 452
- "ImportWorkUnit() — WLM import service" on page 840
- For more information, see *z/OS MVS Programming: Workload Management Services, SC34-2663*.

UnDoImportWorkUnit() — WLM undo import service

Standards

Standards / Extensions	C or C++	Dependencies
z/OS UNIX	both	OS/390 V2R9

Format

```
#include <sys/__wlm.h>

int UnDoImportWorkUnit(wlmxtok_t *exporttoken, unsigned long *conntoken);
```

AMODE 64:

```
#include <sys/__wlm.h>

int UnDoImportWorkUnit(wlmxtok_t *exporttoken, unsigned int *conntoken);
```

General description

Undoes an earlier request to import an enclave using the ImportWorkUnit() function.

The UnDoImportWorkUnit() function uses the following parameters:

**exporttoken*
Points to a work unit export token that was returned from a call to ExportWorkUnit().

**conntoken*
Specifies the connect token that represents the connection to WLM.

Returned value

If successful, UnDoImportWorkUnit() returns 0.

If unsuccessful, UnDoImportWorkUnit() returns -1 and sets errno to one of the following values:

Error Code
Description

EFAULT
An argument of this function contained an address that was not accessible to the caller.

EINVAL
An argument of this function contained a value that is not correct.

EMVSSAF2ERR
An error occurred in the security product.

EMVSWLMERROR
A WLM service failed. Use __errno2() to obtain the WLM service reason code for the failure.

EPERM
The calling thread's address space is not permitted to the BPX.WLMSERVER Facility class. The caller's address space must be permitted to the BPX.WLMSERVER Facility class, if the BPX.WLMSERVER class is defined. If BPX.WLMSERVER is not defined, the calling process is not defined as a superuser (UID=0).

Related information

- “sys/__wlm.h” on page 77
- “ExportWorkUnit() — WLM export service” on page 452
- “ImportWorkUnit() — WLM import service” on page 840
- For more information, see *z/OS MVS Programming: Workload Management Services, SC34-2663*.

unexpected() — Handle exception not listed in exception specification

Standards

Standards / Extensions	C or C++	Dependencies
ANSI/ISO C++	C++ only	

Format

```
#include <exception>

void unexpected(void);
```

General description

The unexpected() function is part of the z/OS XL C++ error handling mechanism. If unexpected() is called directly by the program, the unexpected_handler is the one most recently set by a call to set_unexpected(). If unexpected() is called when control leaves a function by a thrown exception of a type not permitted by an exception specification for the function, as in:

```
void f() throw()      // function may throw no exceptions
    {throw "bad"; }   // throw calls unexpected()
```

the unexpected_handler is the one in effect immediately after evaluating the throw expression.

An unexpected_handler may not return to its caller. It may terminate execution by:

- Throwing an object of a type listed in the exception specification (or an object of any type if the unexpected handler is called directly by the program).
- Throwing an object of type bad_exception.
- Calling terminate(), abort(), or exit(int).

If set_unexpected() has not yet been called, then unexpected() calls terminate().

In a multithreaded environment, if a thread throws an exception that is not listed in its exception specification, then unexpected() is called. The default for unexpected() is to call terminate(), which defaults to calling abort(), which then causes a SIGABRT signal to be generated to the thread issuing the throw. If the SIGABRT signal is not caught, the process is terminated. You can replace the default unexpected() behavior for all threads in the process by using the set_unexpected() function. One possible use of set_unexpected() is to call a function which issues a pthread_exit(). If this is done, a throw of a condition by a thread that is not in the exception specification results in thread termination, but not process termination.

Returned value

unexpected() returns no values.

Refer to *z/OS XL C/C++ Language Reference* for more information about z/OS XL C++ exception handling, including the unexpected() function.

Related information

- “exception” on page 26
- “set_unexpected() — Register a function for unexpected()” on page 1594
- “terminate() — Terminate after failures in C++ error handling” on page 1868

ungetc() — Push character onto input stream

Standards

Standards / Extensions	C or C++	Dependencies
ISO C POSIX.1 XPG4 XPG4.2 C99 Single UNIX Specification, Version 3 Language Environment	both	

Format

```
#include <stdio.h>

int ungetc(int c, FILE *stream);

#define _OPEN_SYS_UNLOCKED_EXT 1
#include <stdio.h>

int ungetc_unlocked(int c, FILE *stream);
```

General description

Pushes the character specified by the value of *c* converted to the unsigned char back onto the given input *stream*. The pushed-back characters are returned by any subsequent read on the same stream in the reverse order of their pushing. That is, the last character pushed will be returned first.

Up to 4 characters can be pushed back to a given input stream. You can call ungetc() up to 4 times consecutively; this will result in 4 characters being pushed back in total.

The *stream* must be open for reading. A subsequent read operation on the *stream* starts with *c*. You cannot push EOF back on the stream using ungetc(). A successful call to the ungetc() function clears the EOF indicator for the stream.

Characters pushed back by ungetc(), and subsequently not read in, will be erased if a fseek(), fsetpos(), rewind(), or fflush() function is called before the character is read from the *stream*. After all the pushed-back characters are read in, the file position indicator is the same as it was before the characters were pushed back.

Each character of pushback backs up the file position by one byte. This affects the value returned by ftell() or fgetpos(), the result of an fseek() using SEEK_CUR, or the result of an fflush(). For example, consider a file that contains: `a b c d e f g h`

After you have just read 'a', the current file position is at the start of 'b'. The following operations will all result in the file position being at the start of 'a', ready to read 'a' again.

```
/* 1 */       ungetc('a', fp);
              fflush(fp);  /* flushes ungetc char and keeps position */

/* 2 */       ungetc('a', fp);
              pos = ftell(fp); /* points to first character */
              fseek(fp, pos, SEEK_SET);

/* 3 */       ungetc('a',fp);
              fseek(fp, 0, SEEK_CUR) /* starts at new file pos'n */

/* 4 */       ungetc('a', fp);
              fgetpos(fp, &fpos); /* gets position of first char */
              fsetpos(fp, &fpos);
```

You can use the environment variable _EDC_COMPAT to cause a z/OS XL C/C++ application to ignore ungetc() characters for fflush(), fgetpos(), and fseek() using SEEK_CUR. For more details, see *z/OS XL C/C++ Programming Guide*.

The ungetc() function is not supported for files opened with `type=record` or `type=blocked`.

ungetc() has the same restriction as any read operation for a read immediately following a write or a write immediately following a read. Between a write and a subsequent read, there must be an intervening flush or reposition. Between a read and a subsequent write, there must also be an intervening flush or reposition unless an EOF has been reached.

ungetc_unlocked() is functionally equivalent to ungetc() with the exception that it is not thread-safe. This function can safely be used in a multithreaded application if and only if it is called while the invoking thread owns the (FILE*) object, as is the case after a successful call to either the flockfile() or ftrylockfile() function.

Returned value

If successful, ungetc() returns the integer argument *c* converted to an `unsigned char`

If *c* cannot be pushed back, ungetc() returns EOF.

ungetc() is treated as a read operation. A flush or reposition is required after a call to ungetc() and before the next write operation.

Example

CELEBU04

```
/* CELEBU04

   In this example, the while statement reads decimal digits
   from an input data stream by using arithmetic statements to
   compose the numeric values of the numbers as it reads them.
   When a nondigit character appears before the EOF, &ungetc.
   replaces it in the input stream so that later input functions
   can process it.

 */
#include <stdio.h>
#include <ctype.h>

int main(void)
{
   FILE *stream;
   int ch;
   unsigned int result = 0;

   stream = fopen("myfile.dat","r+");
   while ((ch = getc(stream)) != EOF && isdigit(ch))
   {
      result = result * 10 + ch - '0';
   }
   printf("result is %i\n",result);
   if (ch != EOF)
   {
      ungetc(ch,stream);         /* Put the nondigit character back */
      ch=getc(stream);
      printf("value put back was %c\n",ch);
   }
}
```

Related information

- “stdio.h” on page 68
- “fflush() — Write buffer to file” on page 523
- “fseek() — Change file position” on page 639
- “getc(), getchar() — Read a character” on page 688
- “putc(), putchar() — Write a character” on page 1347

ungetwc() — Push a wide character onto a stream

Standards

Standards / Extensions	C or C++	Dependencies
XPG4 XPG4.2 Single UNIX Specification, Version 3 Language Environment	both	

Format

```
#include <wchar.h>

wint_t ungetwc(wint_t wc, FILE *stream);

#define _OPEN_SYS_UNLOCKED_EXT 1
#include <wchar.h>

wint_t ungetwc_unlocked(wint_t wc, FILE *stream);
```

General description

Pushes the wide character specified by *wc* back onto the input stream pointed to by *stream*. The pushed-back wide characters will be returned by subsequent reads on that stream in the reverse order of their pushing. A successful intervening call (with the stream pointed to by stream) to a file positioning function (fseek(), fsetpos(), or rewind()) discards any pushed-back wide characters for the stream. The external storage corresponding to the stream is unchanged. There is always at least one wide character of push-back.

If the value of *wc* equals that of the macro WEOF, the operation fails and the input stream is unchanged.

A successful call to the ungetwc() function clears the EOF indicator for the stream. The value of the file position indicator for the stream after reading or discarding all pushed-back wide characters is the same as it was before the wide characters were pushed back.

For a text stream, the file position indicator is backed up by one wide character. This affects ftell(), fflush(), fseek() using SEEK_CUR, and fgetpos(). The environment variable, _EDC_COMPAT can be used to cause a pushed-back wide char to be ignored by fflush(), fseek() with SEEK_CUR, and fgetpos(). For details, see *z/OS XL C/C++ Programming Guide*.

For a binary stream, the position indicator is unspecified until all characters are read or discarded, unless the last character is pushed back, in which case the file position indicator is backed up by one wide character. This affects ftell() and fseek() with SEEK_CUR, fgetpos(), and fflush(). The environment variable _EDC_COMPAT can be used to cause the pushed-back wide character to be ignored by fflush(), fgetpos(), and fseek().

ungetwc() is not supported for files opened with `type=record` or `type=blocked`.

ungetwc() has the same restriction as any read operation for a read immediately following a write, or a write immediately following a read. Between a write and a subsequent read, there must be an intervening flush or reposition. Between a read and a subsequent write, there must also be an intervening flush or reposition unless an EOF has been reached.

ungetwc_unlocked() is functionally equivalent to ungetwc() with the exception that it is not thread-safe. This function can safely be used in a multithreaded application if and only if it is called while the invoking thread owns the (FILE*) object, as is the case after a successful call to either the flockfile() or ftrylockfile() function.

Returned value

If successful, ungetwc() returns the wide character pushed back after conversion.

If unsuccessful, ungetwc() returns WEOF.

Notes:

1. For z/OS XL C/C++ applications, only 1 wide character can be pushed back.
2. The position on the stream after a successful ungetwc() is one wide character before the current position. See *z/OS XL C/C++ Programming Guide* for details on backing up a wide char.

Example

```
#include <stdio.h>
#include <wchar.h>

int main(void)
{
   FILE    *stream;
   wint_t   wc;
   unsigned int result = 0;
⋮
   while ((wc = fgetwc(stream)) != WEOF && iswdigit(wc))
      result = result * 10 + wc - L'0';

   if (wc != WEOF)
      ungetwc(wc, stream);
         /* Put the nondigit wide character back */
}
```

Related information

- “wchar.h” on page 85
- “fflush() — Write buffer to file” on page 523
- “fgetwc() — Get next wide character” on page 532
- “fputwc() — Output a wide-character” on page 607
- “fseek() — Change file position” on page 639
- “fsetpos() — Set file position” on page 649

unlink() — Remove a directory entry

Standards

Standards / Extensions	C or C++	Dependencies
POSIX.1 XPG4 XPG4.2 Single UNIX Specification, Version 3	both	

Format

```
#define _POSIX_SOURCE
#include <unistd.h>

int unlink(const char *pathname);
```

General description

Removes a directory entry. This unlink() deletes the link named by *pathname* and decrements the link count for the file itself.

pathname can refer to a pathname, a link, or a symbolic link. If the *pathname* refers to a symbolic link, unlink() removes the symbolic link but not any file or directory named by the contents of the symbolic link.

If the link count becomes 0 and no process currently has the file open, the file itself is deleted. The space occupied by the file is freed for new use, and the current contents of the file are lost. If one or more processes have the file open when the last link is removed, unlink() removes the link, but the file itself is not removed until the last process closes the file.

unlink() cannot be used to remove a directory; use rmdir() instead.

If unlink() succeeds, the change and modification times for the parent directory are updated. If the file's link count is not 0, the change time for the file is also updated. If unlink() fails, the link is not removed.

Returned value

If successful, unlink() returns 0.

If unsuccessful, unlink() returns -1 and sets errno to one of the following values:

Error Code
: **Description**

EACCES
: The process did not have search permission for some component of *pathname*, or did not have write permission for the directory containing the link to be removed.

EBUSY
: *pathname* cannot be unlinked because it is currently being used by the system or some other process.

ELOOP
: A loop exists in symbolic links. This error is issued if more than POSIX_SYMLOOP symbolic links are detected in the resolution of *pathname*.

ENAMETOOLONG
: *pathname* is longer than **PATH_MAX** characters, or some component of *pathname* is longer than **NAME_MAX** characters while _POSIX_NO_TRUNC is in effect. For symbolic links, the length of the pathname string substituted for a symbolic link exceeds **PATH_MAX**. The **PATH_MAX** and **NAME_MAX** values can be determined using pathconf().

ENOENT
: *pathname* does not exist, or it is an empty string.

ENOTDIR
Some component of the *pathname* prefix is not a directory.

EPERM
pathname is a directory, and unlink() cannot be used on directories.

EROFS
The link to be removed is on a read-only file system.

Example

CELEBU06

```
/* CELEBU06

   This example removes a directory entry, using unlink().

 */
#define _POSIX_SOURCE
#include <fcntl.h>
#include <sys/stat.h>
#include <sys/types.h>
#include <unistd.h>
#undef _POSIX_SOURCE
#include <stdio.h>

main() {
  int fd;
  char fn[]="unlink.file";

  if ((fd = creat(fn, S_IWUSR)) < 0)
    perror("creat() error");
  else {
    close(fd);
    if (unlink(fn) != 0)
      perror("unlink() error");
  }
}
```

Related information

- "unistd.h" on page 82
- "close() — Close a file" on page 288
- "link() — Create a link to a file" on page 970
- "open() — Open a file" on page 1152
- "remove() — Delete file" on page 1428
- "rmdir() — Remove a directory" on page 1453

unlockpt() — Unlock a pseudoterminal master and slave pair

Standards

Standards / Extensions	C or C++	Dependencies
XPG4.2 Single UNIX Specification, Version 3	both	

Format

```
#define _XOPEN_SOURCE_EXTENDED 1
#include <stdlib.h>

int unlockpt(int fildes);
```

General description

The unlockpt() function unlocks the slave pseudoterminal device associated with the master to which *fildes* refers.

Portable applications must call unlockpt() before opening the slave side of a pseudoterminal device.

Returned value

If successful, unlockpt() returns 0.

If unsuccessful, unlockpt() returns -1 and sets errno to one of the following values:

Error Code
  Description

EACCESS
  Either a grantpt() has not yet been issued, or an unlockpt() has already been issued. An unlockpt() must be issued after a grantpt(), and can only be issued once.

EBADF
  The *fildes* argument is not a file descriptor open for writing.

EINVAL
  The *fildes* argument is not associated with a master pseudoterminal device.

Related information

- “stdlib.h” on page 70
- “grantpt() — Grant access to the slave pseudoterminal device” on page 814
- “open() — Open a file” on page 1152
- “ptsname() — Get name of the slave pseudoterminal device” on page 1346

unsetenv() — Delete an environment variable

Standards

Standards / Extensions	C or C++	Dependencies
Single UNIX Specification, version 3	both	z/OS V1R7

Format

```
#define _UNIX03_SOURCE
#include <stdlib.h>

int unsetenv(const char *name);
```

General description

unsetenv() deletes an environment variable from the environment of the calling process. The *name* argument points to a string, which is the name of the variable to be removed. If the string pointed to by *name* contains an '=' character, unsetenv() will fail. If the named variable does not exist in the current environment, the environment will not be changed and unsetenv() will succeed.

Returned value

If successful, unsetenv() returns 0. If unsuccessful, unsetenv() returns -1 and sets errno to indicate the error.

- EINVAL – The name argument is a null pointer, points to an empty string, or points to a string containing an '=' character.

Related information

- "Using Environment Variables" in the *z/OS XL C/C++ Programming Guide*.
- “stdlib.h” on page 70
- “clearenv() — Clear environment variables” on page 281
- “getenv() — Get value of environment variables” on page 704
- “__getenv() — Get an environment variable” on page 705
- “putenv() — Change or add an environment variable” on page 1348
- “setenv() — Add, delete, and change environment variables” on page 1528

usleep() — Suspend execution for an interval

Standards

Standards / Extensions	C or C++	Dependencies
XPG4.2 Single UNIX Specification, Version 3	both	POSIX(ON)

Format

```
#define _XOPEN_SOURCE_EXTENDED 1
#include <unistd.h>

int usleep(useconds_t useconds);
```

General description

The usleep() function suspends thread execution for the number of microseconds specified by the *useconds* argument. Because of other activity, or because of the time spent processing the call, the actual suspension time may be longer than the amount of time specified.

The *useconds* argument must be less than 1,000,000. If the value of *useconds* is 0, then the call has no effect.

The usleep() function will not interfere with a previous setting of the real-time interval timer. If the thread has set this timer before calling usleep(), and if the time specified by *useconds* equals or exceeds the interval timer's prior setting, then the thread will be awakened when the previously set timer interval expires.

Note: The ualarm() and usleep() functions have been moved to obsolescence in Single UNIX Specification, Version 3 and may be withdrawn in a future version. The setitimer(), timer_create(), timer_delete(), timer_getoverrun(), timer_gettime(), or timer_settime() functions are preferred for portability.

Returned value

If successful, usleep() returns 0.

If unsuccessful, usleep() returns -1 and sets errno to one of the following values:

Error Code
: **Description**

EINVAL
: The *useconds* argument was greater than or equal to 1,000,000.

Related information

- “unistd.h” on page 82
- “alarm() — Set an alarm” on page 156
- “setitimer() — Set value of an interval timer” on page 1544
- “sigaction() — Examine or change a signal action” on page 1611
- “sleep() — Suspend execution of a thread” on page 1678
- “ualarm() — Set the interval timer” on page 1939

utime() — Set file access and modification times

Standards

Standards / Extensions	C or C++	Dependencies
POSIX.1 XPG4 XPG4.2 Single UNIX Specification, Version 3	both	

Format

```
#define _POSIX_SOURCE
#include <utime.h>

int utime(const char *pathname, const struct utimbuf *newtimes);
```

General description

Sets the access and modification times of *pathname* to the values in the `utimbuf` structure. If *newtimes* is a NULL pointer, the access and modification times are set to the current time.

Normally, the effective user ID (UID) of the calling process must match the owner UID of the file, or the calling process must have appropriate privileges. However, if *newtimes* is a NULL pointer, the effective UID of the calling process must match the owner UID of the file, or the calling process must have write permission to the file or appropriate privileges.

The contents of a `utimbuf` structure are:

time_t actime
: The new access time (The time_t type gives the number of seconds since the epoch.)

time_t modtime
: The new modification time

Returned value

If successful, utime() returns 0 and updates the access and modification times of the file to those specified.

If unsuccessful, utime() returns -1, does not update file times, and sets errno to one of the following values:

Error Code
  Description

EACCES
  The process does not have search permission on some component of the *pathname* prefix; or all of the following are true:
  - *newtimes* is NULL.
  - The effective user ID of the process does not match the file's owner.
  - The process does not have write permission on the file.
  - The process does not have appropriate privileges.

ELOOP
  A loop exists in symbolic links. This error is issued if more than POSIX_SYMLOOP symbolic links (defined in the limits.h header file) are detected in the resolution of *pathname*.

ENAMETOOLONG
  pathname is longer than **PATH_MAX** characters, or some component of *pathname* is longer than **NAME_MAX** characters while _POSIX_NO_TRUNC is in effect. For symbolic links, the length of the pathname string substituted for a symbolic link exceeds **PATH_MAX**. The **PATH_MAX** and **NAME_MAX** values can be determined using pathconf().

ENOENT
  There is no file named *pathname*, or the *pathname* argument is an empty string.

ENOTDIR
  Some component of the *pathname* prefix is not a directory.

EPERM
  newtimes is not NULL, the effective user ID of the calling process does not match the owner of the file, and the calling process does not have appropriate privileges.

EROFS
  pathname is on a read-only file system.

Example

CELEBU07

```
/* CELEBU07 */
#define _POSIX_SOURCE
#include <fcntl.h>
#include <stdio.h>
#include <sys/stat.h>
#include <sys/types.h>
#include <time.h>
#include <unistd.h>
#include <utime.h>

main() {
  int fd;
  char fn[]="utime.file";
  struct utimbuf ubuf;

  if ((fd = creat(fn, S_IWUSR)) < 0)
    perror("creat() error");
  else {
    close(fd);
    puts("before utime()");
    system("ls -l utime.file");
    ubuf.modtime = 0;
    time(&ubuf.actime);
    if (utime(fn, &ubuf) != 0)
      perror("utime() error");
    else {
      puts("after utime()");
      system("ls -l utime.file");
    }
    unlink(fn);
  }
}
```

Output

```
before utime()
--w-------   1 WELLIE   SYS1           0 Apr 19 15:23 utime.file
after utime()
--w-------   1 WELLIE   SYS1           0 Dec 31  1969 utime.file
```

Related information

- “limits.h” on page 39
- “utime.h” on page 84
- “utimes() — Set file access and modification times”

utimes() — Set file access and modification times

Standards

Standards / Extensions	C or C++	Dependencies
XPG4.2 Single UNIX Specification, Version 3	both	

Format

```
#define _XOPEN_SOURCE_EXTENDED 1
#include <sys/time.h>

int utimes(const char *path, const struct timeval times[2]);
```

General description

The utimes() function sets the access and modification times of the file pointed to by the *path* argument to the value of the *times* argument.

The *times* argument is an array of two `timeval` structures. The first array member represents the date and time of the last access, and the second member represents the date and time of the last modification. The times in the `timeval` structure are measured in seconds and microseconds since the Epoch, but the actual time stored with the file are rounded to the nearest second. The `timeval` members are:

tv_sec seconds since January 1, 1970 (UTC)

tv_usec
microseconds

If the *times* argument is a NULL pointer, the access and modification times of the file are set to the current time. The same process privilege requirements of the utime() function are required by utimes(). The last file status change, field `st_ctime` in a stat(), is updated with the current time.

Note: The utimes() function has been moved to the Legacy Option group in Single UNIX Specification, Version 3 and may be withdrawn in a future version. The utime() function is preferred for portability.

Returned value

If successful, utimes() returns 0.

If unsuccessful, utimes() returns -1 and sets errno to one of the following values:

Error Code
Description

EACCES
The process does not have search permission on some component of the *path* prefix; or all of the following are true:
- *times* is NULL
- The effective user ID of the process does not match the file's owner
- The process does not have write permission on the file
- The process does not have appropriate privileges

ELOOP
A loop exists in symbolic links. This error is issued if more than **POSIX_SYMLOOP** symbolic links (defined in the limits.h header file) are detected in the resolution of *path*.

ENAMETOOLONG
path is longer than **PATH_MAX** characters or some component of *path* is longer than **NAME_MAX** characters while **_POSIX_NO_TRUNC** is in effect. For symbolic links, the length of the pathname string substituted for a symbolic link exceeds **PATH_MAX**. The **PATH_MAX** and **NAME_MAX** values can be determined using pathconf().

ENOTDIR
Some component of the *path* prefix is not a directory.

ENOTENT
There is no file named *path*, or the *path* argument is an empty string.

EPERM
times is not NULL, the effective user ID of the calling process does not match the owner of the file, and the calling process does not have appropriate privileges.

EROFS
path is on a read-only file system.

Related information

- “sys/time.h” on page 75
- “utime() — Set file access and modification times” on page 1967

__utmpxname() — Change the utmpx database name

Standards

Standards / Extensions	C or C++	Dependencies
z/OS UNIX	both	

Format

```
#define _XOPEN_SOURCE_EXTENDED 1
#include <utmpx.h>

int __utmpxname(char *file);
```

General description

The __utmpxname() function changes the name of the utmpx database file for the current thread from default /etc/utmpx to the name specified by *file*. The __utmpxname() function does not open the file. It closes the old utmpx database file, if it is currently opened for the current thread, and saves the new utmpx database file name. If the file does not exist no indication is given.

Because the __utmpxname() function processes thread-specific data the __utmpxname() function can be used safely from a multithreaded application. If multiple threads in the same process open the database, then each thread opens the database with a different file descriptor. The thread's database file descriptor is closed when the calling thread terminates or the endutxent() function is called by the calling thread.

Programs must not reference the data passed back by getutxline(), getutxid(), getutxent(), or pututxline() after __utmpxname() has been called (the storage has been freed.) The endutxent() function resets the name of the utmpx database back to the default value. If you must do additional utmpx operations on a nonstandard utmpx database after calling endutxent(), then call __utmpxname() again, to reestablish the nonstandard name.

Returned value

If successful, __utmpxname() returns 0.

If unsuccessful, __utmpxname() returns -1.

Related information

- "utmpx.h" on page 84
- "endutxent() — Close the utmpx database" on page 425
- "getutxent() — Read next entry in utmpx database" on page 793
- "getutxid() — Search by ID utmpx database" on page 795
- "getutxline() — Search by line utmpx database" on page 796
- "pututxline() — Write entry to utmpx database" on page 1354
- "setutxent() — Reset to start of utmpx database" on page 1595

utoa() — Convert unsigned int into a string

Standards

Standards / Extensions	C or C++	Dependencies
z/OS UNIX	both	z/OS V1R5

Format

```
#define _OPEN_SYS_ITOA_EXT
#include <stdlib.h>

char * utoa(unsigned int n, char * buffer, int radix);
```

General description

The utoa() function coverts the unsigned integer n into a character string. The string is placed in the buffer passed, which must be large enough to hold the output. The radix values can be OCTAL, DECIMAL, or HEX. When the radix is DECIMAL, utoa() produces the same result as the following statement:

```
(void) sprintf(buffer, "%u", n);
```

with buffer the returned character string. When the radix is OCTAL, utoa() formats unsigned integer n into an octal constant. When the radix is HEX, utoa() formats unsigned integer n into a hexadecimal constant. The hexadecimal value will include lower case abcdef, as necessary.

Returned value

String pointer (same as buffer) will be returned. When passed an invalid radix argument, function will return NULL and set errno to EINVAL.

Portability considerations

This is a non-standard function. Even though the prototype given is commonly used by compilers on other platforms, there is no guarantee that this function will behave the same on all platforms, in all cases. You can use this function to help port applications from other platforms, but you should avoid using it when writing new applications, in order to ensure maximum portability.

Example

CELEBU10

```
/* CELEBU10

   This example reads an unsigned int and formats it to decimal,
   octal, hexadecimal constants converted to a character string.

*/

#define _OPEN_SYS_ITOA_EXT
#include <stdio.h>
#include <stdlib.h>

int main ()
{
   unsigned int i;
   char buffer [sizeof(unsigned int)*8+1];
   printf ("Enter a number: ");
   if (scanf ("%u",&i) == 1) {
      utoa (i,buffer,DECIMAL);
      printf ("decimal: %s\n",buffer);
      utoa (i,buffer,HEX);
      printf ("hexadecimal: %s\n",buffer);
      utoa (i,buffer,OCTAL);
      printf ("octal: %s\n",buffer);
   }
   return 0;
}
```

Output

If the input is 1234, then the output should be:

```
decimal: 1234
hexadecimal: 4d2
octal: 2322
```

Related information

- “stdlib.h” on page 70
- “itoa() — Convert int into a string” on page 925
- “lltoa() — Convert long long into a string” on page 979
- “ltoa() — Convert long into a string” on page 1027
- “ulltoa() — Convert unsigned long long into a string” on page 1943
- “ultoa() — Convert unsigned long into a string” on page 1945

va_arg(), va_copy(), va_end(), va_start() — Access function arguments

Standards

Standards / Extensions	C or C++	Dependencies
ISO C C99 Single UNIX Specification, Version 3	both	

Format

```
#include <stdarg.h>

var_type va_arg(va_list arg_ptr, var_type);
void va_end(va_list arg_ptr);
void va_start(va_list arg_ptr, variable_name);
```

C99:

```
#define _ISOC99_SOURCE
#include <stdarg.h>

var_type va_arg(va_list arg_ptr, var_type);
void va_end(va_list arg_ptr);
void va_start(va_list arg_ptr, variable_name);
void va_copy(va_list dest, va_list src);
```

General description

The va_arg(), va_end(), and va_start() macros access the arguments to a function when it takes a fixed number of required arguments and a variable number of optional arguments. You declare required arguments as ordinary parameters to the function and access the arguments through the parameter names.

The va_start() macro initializes the *arg_ptr* pointer for subsequent calls to va_arg() and va_end().

The argument *variable_name* is the identifier of the rightmost named parameter in the parameter list (preceding `, ...`). Use the va_start() macro before the va_arg() macro. Corresponding va_start() and va_end() macro calls must be in the same function. If *variable_name* is declared as a register, with a function or an array type, or with a type that is not compatible with the type that results after application of the default argument promotions, then the behavior is undefined.

The va_arg() macro retrieves a value of the given *var_type* from the location given by *arg_ptr* and increases *arg_ptr* to point to the next argument in the list. The va_arg() macro can retrieve arguments from the list any number of times within the function.

The macros also provide fixed-point decimal support under z/OS XL C. The `sizeof(xx)` operator is used to determine the size and type casting that is used to generate the values. Therefore, a call, such as, `x = va_arg(ap, _Decimal(5,2));` is valid. The size of a fixed-point decimal number, however, cannot be made a variable. Therefore, a call, such as, `z = va_arg(ap, _Decimal(x,y))` where `x = 5` and `y = 2` is invalid.

The va_end() macro is needed by some systems to indicate the end of parameter scanning.

va_start() and va_arg() do not work with parameter lists of functions whose linkages were changed with the `#pragma linkage` directive.

stdarg.h and varargs.h are mutually exclusive. Whichever #include comes first, determines the form of macro that is visible.

The type definition for the va_list type is normally "char *va_list[2]". Some applications (especially ported applications) require that the va_list type be defined as "char *va_list". This alternate va_list type is available if the user defines the feature test macro _VARARG_EXT_ before the inclusion of any system header file. If the _VARARG_EXT_ feature test macro is defined, va_list will be typed as char * va_list, and the functions vprintf(), vfprintf(), vsprintf(), and vswprintf() will use this alternate va_list type.

The va_copy() function creates a copy (*dest*) of a variable of type va_list (*src*). The copy appear as if it has gone through a va_start() and the exact set of sequences of va_arg() as that of *src*.

After va_copy() initializes dest, the va_copy() macro shall not be invoked to reinitialize *dest* without an intervening invocation of the va_end() macro for the same *dest*.

Returned value

The va_arg() macro returns the current argument.

The va_end(), va_copy(), and va_start() macros return no values.

Example

CELEBV01

```
/* CELEBV01

   This example passes a variable number of arguments to a function,
   stores each argument in an array, and prints each argument.

 */
#include <stdio.h>
#include <stdarg.h>

void vout(int max, ...);

int main(void)
{
   vout(3, "Sat", "Sun", "Mon");
   printf("\n");
   vout(5, "Mon", "Tues", "Wed", "Thurs", "Fri");
}

void vout(int max, ...)
{
   va_list arg_ptr;
   int args = 0;
   char *days[7];

   va_start(arg_ptr, max);
   while(args < max)
   {
      days[args] = va_arg(arg_ptr, char *);
      printf("Day:  %s  \n", days[args++]);
      }
   va_end(arg_ptr);
}
```

Output

```
Day:  Sat
Day:  Sun
Day:  Mon

Day:  Mon
Day:  Tues
Day:  Wed
Day:  Thurs
Day:  Fri
```

```
/* This example uses a variable number of arguments for
   fixed-point decimal data types.
   The example works in z/OS XL C only.
 */
#include <stdio.h>
#include <stdarg.h>
#include <decimal.h>

void vprnt(int, ...);

int main(void) {
  int i = 168;
  decimal(10,2) pd01 = 12345678.12d;
  decimal(20,5) pd02 = -987.65d;
  decimal(31,20) pd03 = 12345678901.12345678900987654321d;
  int j = 135;

  vprnt(0, i, pd01, pd02, pd03, j);

  return(0);
}

void vprnt(int whichcase, ...) {
  va_list arg_ptr;
  int m, n;
  decimal(10,2) va01;
  decimal(20,5) va02;
  decimal(31,20) va03;

  va_start(arg_ptr, whichcase);

  switch (whichcase) {
    case 0:
      m = va_arg(arg_ptr, int);
      va01 = va_arg(arg_ptr, decimal(10,2));
      va02 = va_arg(arg_ptr, decimal(20,5));
      va03 = va_arg(arg_ptr, decimal(31,20));
      n = va_arg(arg_ptr, int);
      printf("m    = %d\n", m);
      printf("va01 = %D(10,2)\n", va01);
      printf("va02 = %D(20,5)\n", va02);
      printf("va03 = %D(31,20)\n", va03);
      printf("n    = %d\n", n);
      break;
    default:
      printf("Illegal case number : %d\n", whichcase);
  }

  va_end(arg_ptr);
}
```

Output:

```
m    = 168
va01 = 12345678.12
va02 = -987.65000
va03 = 12345678901.12345678900987654321
n    = 135
```

CELEBV02

```
/* CELEBV02

   These examples use the _XOPEN_SOURCE feature test macro,
   This ecample passes a variable number of arguments to a function,
   stores each argument in an array, and prints each argument.

 */
#define _XOPEN_SOURCE
#include <stdio.h>
#include <varargs.h>

void vout(va_alist)
va_dcl
{
   va_list arg_ptr;
   int args = 0;
   int max;
   char *days[7];

   va_start(arg_ptr);
   max = va_arg(arg_ptr, int);
   while(args < max) {
        days[args] = va_arg(arg_ptr, char *);
        printf("Days:       %s  \n", days[args++]);
        }
   va_end(arg_ptr);
}

int main(void)
{
    vout(3,"Sat","Sun","Mon");
    printf("\n");
    vout(5,"Mon","Tues","Wed","Thurs","Fri");
}
```

```
/* This example uses a variable number of arguments for
   fixed-point decimal data types.
   The example works in z/OS XL C only.
 */
#define  _XOPEN_SOURCE
#include <stdio.h>
#include <varargs.h>
#include <decimal.h>


void vprnt(va_alist)
va_dcl
{
  va_list arg_ptr;
  int m, n, whichcase;
  decimal(10,2) va01;
  decimal(20,5) va02;
  decimal(31,20) va03;

  va_start(arg_ptr);
  whichcase = va_arg(arg_ptr, int);

  switch (whichcase) {
    case 0:
      m = va_arg(arg_ptr, int);
      va01 = va_arg(arg_ptr, decimal(10,2));
      va02 = va_arg(arg_ptr, decimal(20,5));
      va03 = va_arg(arg_ptr, decimal(31,20));
      n = va_arg(arg_ptr, int);
      printf("m    = %d\n", m);
      printf("va01 = %D(10,2)\n", va01);
      printf("va02 = %D(20,5)\n", va02);
      printf("va03 = %D(31,20)\n", va03);
      printf("n    = %d\n", n);
      break;
    default:
      printf("Illegal case number : %d\n", whichcase);
  }

  va_end(arg_ptr);
}


int main(void) {
  int i = 168;
  decimal(10,2) pd01 = 12345678.12d;
  decimal(20,5) pd02 = -987.65d;
  decimal(31,20) pd03 = 12345678901.12345678900987654321d;
  int j = 135;

  vprnt(0, i, pd01, pd02, pd03, j);

  return(0);
}
}
```

```
#define _ISOC99_SOURCE
#include <stdio.h>
#include <stdarg.h>
void prnt(int max, ...);

int main(void)
      {
```

```
      prnt(8, "0", "1", "1", "2", "3", "5", "8", "13");
     }

void prnt(int max, ...)
    {
      va_list src;
      va_list dest;
      int args = 0;
      char *fib[8];
      va_start(src, max);
      va_copy(dest, src);
      while(args < max) {
        fib[args] = va_arg(dest, char *);
        printf("fib[%d]:  %s  \n", args, fib[args++]);
         }
      va_end(dest);
    }
```

```
Output

fib[0]:  0
fib[1]:  1
fib[2]:  1
fib[3]:  2
fib[4]:  3
fib[5]:  5
fib[6]:  8
fib[7]:  13
```

Related information

- “stdarg.h” on page 64
- “varargs.h” on page 84
- “vfprintf() — Format and print data to stream” on page 1982
- “vprintf() — Format and print data to stdout” on page 1988
- “vsprintf() — Format and print data to buffer” on page 1991

valloc() — Page-aligned memory allocator

Standards

Standards / Extensions	C or C++	Dependencies
XPG4.2	both	

Format

```
#define _XOPEN_SOURCE_EXTENDED 1
#include <stdlib.h>

void *valloc(size_t size);
```

General description

Restriction: This function is not supported in AMODE 64.

The valloc() function has the same effect as malloc(), except that the allocated memory will be aligned to a multiple of the value returned by sysconf(_SC_PAGESIZE).

Note: When free() is used to release storage obtained by valloc(), the storage is not made available for reuse. The storage will not be freed until the enclave goes away.

Special behavior for C++: The C++ keywords new and delete are not interoperable with valloc(), calloc(), free(), malloc(), or realloc().

Note:

This function is kept for historical reasons. It was part of the Legacy Feature in Single UNIX Specification, Version 2, but has been withdrawn and is not supported as part of Single UNIX Specification, Version 3. The malloc() or mmap() functions are preferred for portability.

If it is necessary to continue using this function in an application written for Single UNIX Specification, Version 3, define the feature test macro _UNIX03_WITHDRAWN before including any standard system headers. The macro exposes all interfaces and symbols removed in Single UNIX Specification, Version 3.

Returned value

If successful, valloc() returns a pointer to the reserved storage. The storage space to which the returned value points is guaranteed to be aligned on a page boundary.

If unsuccessful, valloc() returns NULL if there is not enough storage available, or if *size* is `0`. If valloc() returns NULL because there is not enough storage, it sets errno to one of the following values:

Error Code
  Description

ENOMEM
  Insufficient memory is available

Related information

- “stdlib.h” on page 70
- “malloc() — Reserve storage block” on page 1031
- “sysconf() — Determine system configuration options” on page 1798

vfork() — Create a new process

Standards

Standards / Extensions	C or C++	Dependencies
XPG4.2 Single UNIX Specification, Version 3	both	

Format

```
#define _XOPEN_SOURCE_EXTENDED 1
#include <sys/types.h>
#include <unistd.h>

pid_t vfork(void);
```

Note: Although POSIX.1 does not require that the <sys/types.h> include file be included, XPG4 has it as an optional header. Therefore it is recommended that you include it for portability.

General description

The vfork() function creates a new process. The vfork() function has the same effect as fork(), except that the behavior is undefined, if the process created by vfork() attempts to call any other C/370 function before calling either exec() or _exit(). The new process (the *child process*) is an exact duplicate of the process that calls vfork() (the *parent process*), except for the following:

- The child process has a unique process ID (PID), which does not match any active process group ID.
- The child has a different parent process ID, that is, the process ID of the process that called vfork().
- The child has its own copy of the parent's file descriptors. Each file descriptor in the child refers to the same open file description as the corresponding file descriptor in the parent.
- The child has its own copy of the parent's open directory streams. Each child's open directory stream may share directory stream positioning with the corresponding parent's directory stream.
- The following elements in the `tms` structure are set to 0 in the child:

  ```
  tms_utime
  tms_stime
  tms_cutime
  tms_cstime
  ```

  For more information about these elements, see "times() — Get process and child process times" on page 1879.
- The child does not inherit any file locks previously set by the parent.
- The child process has no alarms set (similar to the results of a call to alarm() with an argument value of 0).
- The child has no pending signals.
- The child process may have its own copy of the parent's message catalog descriptors.
- All semadj values are cleared.
- Interval timers are reset in the child process.

In all other respects, the child is identical to the parent. Because the child is a duplicate, it contains the same call to vfork() that was in the parent. Execution begins with this vfork() call, which returns a value of 0; the child then proceeds with normal execution.

The vfork() function is not supported from a multithread environment.

For more information on vfork() from a z/OS perspective, refer to *z/OS UNIX System Services Programming: Assembler Callable Services Reference, SA23-2281*.

You can use z/OS memory files from a z/OS UNIX program. However, use of the vfork() function from the program removes access from a hiperspace memory file for the child process. Use of an exec function from the program clears a memory file when the process address space is cleared.

Special behavior for C: For POSIX resources, vfork() behaves as just described. But in general, MVS resources that existed in the parent do *not* exist in the child. This is true for open streams in MVS data sets and assembler-accessed z/OS facilities, such as STIMERS. In addition, z/OS allocations (through JCL, SVC99, or ALLOCATE) are not passed to the child process.

Note: The vfork() function has been moved to obsolescence in Single UNIX Specification, Version 3 and may be withdrawn in a future version. The fork() function is preferred for portability.

Returned value

If successful, vfork() returns 0 to the child process and the process ID of the newly created child to the parent process.

If unsuccessful, vfork() fails to create a child process and returns -1 to the parent. vfork() sets errno to one of the following values:

Error Code
Description

EAGAIN
There are insufficient resources to create another process, or else the process has already reached the maximum number of processes you can run.

ELEMSGERR
Language Environment message file not available.

ELEMULTITHREAD
vfork() was invoked from a multi-threaded environment.

ELENOFORK
Application contains a language that does not support fork().

ENOMEM
The process requires more space than is available.

Related information

- “sys/types.h” on page 75
- “unistd.h” on page 82
- “alarm() — Set an alarm” on page 156
- “fcntl() — Control open file descriptors” on page 474
- “getrlimit() — Get current or maximum resource consumption” on page 765
- “nice() — Change priority of a process” on page 1145
- “putenv() — Change or add an environment variable” on page 1348
- “rexec() — Execute commands one at a time on a remote host” on page 1447
- “semop() — Semaphore operations” on page 1488
- “setlocale() — Set locale” on page 1552
- “shmat() — Shared memory attach operation” on page 1598
- “sigaction() — Examine or change a signal action” on page 1611
- “signal() — Handle interrupts” on page 1643
- “sigpending() — Examine pending signals” on page 1650
- “sigprocmask() — Examine or change a thread” on page 1651
- “stat() — Get file information” on page 1720
- “system() — Execute a command” on page 1805
- “times() — Get process and child process times” on page 1879
- “ulimit() — Get or set process file size limits” on page 1942

vfprintf() — Format and print data to stream

Standards

Standards / Extensions	C or C++	Dependencies
ISO C XPG4 XPG4.2 C99 Single UNIX Specification, Version 3 Language Environment	both	

Format

```
#include <stdarg.h>
#include <stdio.h>

int vfprintf(FILE *  __restrict__stream,
             const char *  __restrict__format, va_list arg_ptr);

#define _OPEN_SYS_UNLOCKED_EXT 1
#include <stdio.h>

int vfprintf_unlocked(FILE *  __restrict__stream,
                      const char *  __restrict__format, va_list arg_ptr);
```

General description

The vfprintf() function is similar to fprintf(), except that *arg_ptr* points to a list of arguments whose number can vary from call to call in the program. These arguments should be initialized by va_start() for each call. In contrast, fprintf() can have a list of arguments, but the number of arguments in that list is fixed when you compile the program. For a specification of the *format* string, see “fprintf(), printf(), sprintf() — Format and write data” on page 588.

vfprintf() is not supported for files opened with `type=record` or `type=blocked`.

vfprintf() has the same restriction as any write operation for a read immediately following a write or a write immediately following a read. Between a write and a subsequent read, there must be an intervening flush or reposition. Between a read and a subsequent write, there must also be an intervening flush or reposition unless an EOF has been reached.

vfprintf_unlocked() is functionally equivalent to vfprintf() with the exception that it is not thread-safe. This function can safely be used in a multithreaded application if and only if it is called while the invoking thread owns the (FILE*) object, as is the case after a successful call to either the flockfile() or ftrylockfile() function.

Returned value

If successful, vfprintf() returns the number of characters written to *stream*.

If unsuccessful, vfprintf() returns a negative value.

Note: In contrast to some UNIX-based implementations of the C language, the z/OS XL C/C++ implementation of the vprintf() family increments the pointer to the variable arguments list. To control whether the pointer to the argument is incremented, call the `va_end` macro after each call to vsprintf().

Example

CELEBV03

```
/* CELEBV03

   This example prints out a variable number of strings to the
   file myfile.dat, using &vfprt..

 */
#include <stdarg.h>
#include <stdio.h>

void vout(FILE *stream, char *fmt, ...);
char fmt1 [] = "%s  %s  %s\n";

int main(void)
{
   FILE *stream;
   stream = fopen("myfile.dat", "w");

   vout(stream, fmt1, "Sat", "Sun", "Mon");
}

void vout(FILE *stream, char *fmt, ...)

{
   va_list arg_ptr;

   va_start(arg_ptr, fmt);
   vfprintf(stream, fmt, arg_ptr);
   va_end(arg_ptr);
}
```

Output:

```
Sat  Sun  Mon
```

Related information

- “stdarg.h” on page 64
- “stdio.h” on page 68
- “va_arg(), va_copy(), va_end(), va_start() — Access function arguments” on page 1973
- “vprintf() — Format and print data to stdout” on page 1988
- “vsprintf() — Format and print data to buffer” on page 1991

vfscanf(), vscanf(), vsscanf() — Format input of a STDARG argument list

Standards

Standards / Extensions	C or C++	Dependencies
C99 Single UNIX Specification, Version 3 Language Environment C++ TR1 C99	both	z/OS V1R7

Format

```
#define _ISOC99_SOURCE
#include <stdarg.h>
#include <stdio.h>
```

```
int vfscanf(FILE *__restrict__ stream,
            const char *__restrict__ format, va_list arg);

int vscanf(const char *__restrict__ format, va_list arg);

int vsscanf(const char *__restrict__ s,
            const char *__restrict__ format, va_list arg);

#define _OPEN_SYS_UNLOCKED_EXT 1
#include <stdio.h>

int vfscanf_unlocked(FILE *__restrict__ stream,
            const char *__restrict__ format, va_list arg);

int vscanf_unlocked(const char *__restrict__ format, va_list arg);
```

General description

The vfscanf(), vscanf(), and vsscanf() functions are equivalent to the fscanf(), scanf(), and sscanf() functions, respectively, except that instead of being called with a variable number of arguments, they are called with an argument list as defined in stdarg.h.

The argument list should be initialized using the **va_start** macro before each call. These functions do not invoke the **va_end** macro, but instead invoke the **va_arg** macro causing the value of *arg* after the return to be unspecified.

vfscanf() and vscanf() are not supported for files opened with a record type. They also have the same restrictions as a write immediately following a read or a read immediately following a write. This is because, between a write and a subsequent read, there must be an intervening flush or reposition and between a read and a subsequent write, there must also be an intervening flush or reposition unless EOF has been reached.

Note: In contrast to some UNIX-based implementations of the C language, the z/OS XL C/C++ implementation of the vscanf() family increments the pointer to the variable arguments list. To control whether the pointer is incremented, call the **va_end** macro after each function call.

vfscanf_unlocked() and vscanf_unlocked() are functionally equivalent to vfscanf() and vscanf() with the exception that they are not thread-safe. These functions may safely be used in a multithreaded application if and only they are called while the invoking thread owns the (FILE*) object, as is the case after a successful call to either the flockfile() or ftrylockfile() function.

Returned value

Refer to fscanf().

Related information

- stdarg.h
- stdio.h
- fscanf()

vfwprintf(), vswprintf(), vwprintf() — Format and write wide characters of a STDARG argument list

Standards

Standards / Extensions	C or C++	Dependencies
ISO C Amendment C99 Single UNIX Specification, Version 3 Language Environment	both	

Format

Non-XPG4:

```
#include <stdarg.h>
#include <wchar.h>

int vfwprintf(FILE * __restrict__ stream,
              const wchar_t * __restrict__ format, va_list arg);
int vswprintf(wchar_t * __restrict__ wcs, size_t n,
              const wchar_t * __restrict__ format, va_list arg);
int vwprintf(const wchar_t * __restrict__ format, va_list arg);

#define _OPEN_SYS_UNLOCKED_EXT 1
#include <wchar.h>

int vfwprintf_unlocked(FILE * __restrict__ stream,
              const wchar_t * __restrict__ format, va_list arg);
int vwprintf_unlocked(const wchar_t * __restrict__ format, va_list arg);
```

XPG4:

```
#define _XOPEN_SOURCE
#define _MSE_PROTOS
#include <stdarg.h>
#include <wchar.h>

int vfwprintf(FILE * __restrict__ stream,
              const wchar_t * __restrict__ format, va_list arg);
int vswprintf(wchar_t * __restrict__ wcs, size_t n,
              const wchar_t * __restrict__ format, va_list arg);
int vwprintf(const wchar_t * __restrict__ format, va_list arg);

#define _OPEN_SYS_UNLOCKED_EXT 1
#include <wchar.h>

int vfwprintf_unlocked(FILE * __restrict__ stream,
              const wchar_t * __restrict__ format, va_list arg);
int vwprintf_unlocked(const wchar_t * __restrict__ format, va_list arg);
```

General description

vfwprintf(), vswprintf(), and vwprintf() functions are equivalent to fprintf(), sprintf(), and printf() functions, respectively, except for the following:

- Instead of being called with a variable number of arguments, they are called with an argument list as defined in stdarg.h.
- For vswprintf(), the argument *wcs* specifies an array of type `wchar_t`, rather than an array of type char, into which the generated output is to be written.
- The argument *format* specifies an array of type `wchar_t`, rather than an array of type char, which describes how subsequent arguments are converted for output.

- %c without an l prefix means an integer argument is to be converted to wchar_t, as if by calling mbtowc(), and then written.
- %c with l prefix means a wint_t is converted to wchar_t and then written.
- %s without an l prefix means a character array containing a multibyte character sequence is to be converted to an array of wchar_t and written. The conversion will take place as if mbrtowc() were called repeatedly.
- %s with l prefix means an array of wchar_t will be written. The array is written up to but not including the terminating NULL character, unless the precision specifies a shorter output.

For vswprintf(), a NULL wide character is written at the end of the wide characters written; the NULL wide character is not counted as part of the returned sum. If copying takes place between objects that overlap, the behavior is undefined.

Note: The vfwprintf() and vwprintf() functions have a dependency on the level of the Enhanced ASCII Extensions. See "Enhanced ASCII support" on page 2113 for details.

vfwprintf_unlocked() and vwprintf_unlocked() are functionally equivalent to vfwprintf() and vwprintf() with the exception that they are not thread-safe. These functions may safely be used in a multithreaded application if and only if they are called while the invoking thread owns the (FILE*) object, as is the case after a successful call to either the flockfile() or ftrylockfile() function.

Special behavior for XPG4: If you define any feature test macro specifying XPG4 behavior before the statement in your program source file to include the wchar header, then you must also define the _MSE_PROTOS feature test macro to make the declaration of the vfwprintf(), vswprintf(), or vwprintf() function in the wchar header available when you compile your program. Please see Table 2 on page 4 for a list of XPG4 and other feature test macros.

Returned value

If successful, vfwprintf(), vswprintf(), and vwprintf() return the number of wide characters written, not counting the terminating NULL wide character.

If unsuccessful, a negative value is returned.

If *n* or more wide characters were requested to be written, vswprintf() returns a negative value and sets errno to indicate the error.

Note: In contrast to some UNIX-based implementations of the C language, the z/OS XL C/C++ implementation of the vprintf() family increments the pointer to the variable arguments list. To control whether the pointer to the argument is incremented, call the va_end macro after each call to vfwprintf(), vswprintf(), or vwprintf().

Example

CELEBV06

```
/* CELEBV06 */
#define _XOPEN_SOURCE
#define _MSE_PROTOS
#include <stdio.h>
#include <stdarg.h>
#include <wchar.h>
```

```
void vout(wchar_t *, size_t, wchar_t *, ...);

wchar_t *format3 = L"%S %d  %S";
wchar_t *format5 = L"%S  %d  %S  %d  %S";

int main(void)
{
   wchar_t wcstr[100];

   vout(wcstr, 100, format3, L"ONE", 2L, L"THREE");
   printf("%S\n",wcstr);
   vout(wcstr, 100, format5, L"ONE", 2L, L"THREE", 4L, L"FIVE");
   printf("%S\n",wcstr);
}

void vout(wchar_t *wcs, size_t n, wchar_t *fmt, ...)
{
   va_list arg_ptr;

   va_start(arg_ptr, fmt);
   if(vswprintf(wcs, n, fmt, arg_ptr)<0)
      perror("vswprintf() error");
   va_end(arg_ptr);
}
```

Related information

- “stdarg.h” on page 64
- “wchar.h” on page 85
- “fwprintf(), swprintf(), wprintf() — Format and write wide characters” on page 672
- “vsprintf() — Format and print data to buffer” on page 1991

vfwscanf(), vwscanf(), vswscanf() — Wide-character formatted input of a STDARG argument list

Standards

Standards / Extensions	C or C++	Dependencies
C99 Single UNIX Specification, Version 3 Language Environment C++ TR1 C99	both	z/OS V1R7

Format

```
#define _ISOC99_SOURCE
#include <stdarg.h>
#include <stdio.h>
#include <wchar.h>

int vfwscanf(FILE *__restrict__ stream,
             const wchar_t *__restrict__ format, va_list arg);

int vwscanf(const wchar_t *__restrict__ format, va_list arg);

int vswscanf(const wchar_t *__restrict__ ws,
             const wchar_t *__restrict__ format, va_list arg);

#define _OPEN_SYS_UNLOCKED_EXT 1
#include <wchar.h>
```

```
int vfwscanf_unlocked(FILE *__restrict__ stream,
            const wchar_t *__restrict__ format, va_list arg);
int vwscanf_unlocked(const wchar_t *__restrict__ ws,
            const wchar_t *__restrict__ format, va_list arg);
```

General description

The vfwscanf(), vswscanf(), and vwscanf() functions are equivalent to the fwscanf(), swscanf(), and wscanf() functions, respectively, except that instead of being called with a variable number of arguments, they are called with an argument list as defined in stdarg.h.

The argument list should be initialized using the **va_start** macro before each call. These functions do not invoke the **va_end** macro, but instead invoke the **va_arg** macro causing the value of *arg* after the return to be unspecified.

vfwscanf() and vwscanf() are not supported for files opened with a record type. They also have the same restrictions as a write immediately following a read or a read immediately following a write. This is because, between a write and a subsequent read, there must be an intervening flush or reposition and between a read and a subsequent write, there must also be an intervening flush or reposition unless EOF has been reached.

Note: In contrast to some UNIX-based implementations of the C language, the z/OS XL C/C++ implementation of the vwscanf() family increments the pointer to the variable arguments list. To control whether the pointer is incremented, call the **va_end** macro after each function call.

vfwscanf_unlocked() and vwscanf_unlocked() are functionally equivalent to vfwscanf() and vwscanf() with the exception that they are not thread-safe. These functions may safely be used in a multithreaded application if and only if they are called while the invoking thread owns the (FILE*) object, as is the case after a successful call to either the flockfile() or ftrylockfile() function.

Returned value

Refer to fwscanf().

Related information

- stdarg.h
- stdio.h
- wchar.h
- fwscanf()

vprintf() — Format and print data to stdout

Standards

Standards / Extensions	C or C++	Dependencies
ISO C XPG4.2 C99 Single UNIX Specification, Version 3 Language Environment	both	

Format

```
#include <stdarg.h>
#include <stdio.h>

int vprintf(const char *  __restrict__format, va_list arg_ptr);

#define _OPEN_SYS_UNLOCKED_EXT 1
#include <stdio.h>

int vprintf_unlocked(const char *  __restrict__format, va_list arg_ptr);
```

General description

The vprintf() function is similar to printf(), except that *arg_ptr* points to a list of arguments whose number can vary from call to call in the program. These arguments should be initialized by va_start() for each call. In contrast, printf() can have a list of arguments, but the number of arguments in that list is fixed when you compile the program. For a specification of the *format* string, see "fprintf(), printf(), sprintf() — Format and write data" on page 588.

vprintf() is not supported for files opened with `type=record` or `type=blocked`.

vprintf() has the same restriction as any write operation for a read immediately following a write or a write immediately following a read. Between a write and a subsequent read, there must be an intervening flush or reposition. Between a read and a subsequent write, there must also be an intervening flush or reposition unless an EOF has been reached.

vprintf_unlocked() is functionally equivalent to vprintf() with the exception that it is not thread-safe. This function can safely be used in a multithreaded application if and only if it is called while the invoking thread owns the (FILE*) object, as is the case after a successful call to either the flockfile() or ftrylockfile() function.

Returned value

If successful, vprintf() returns the number of characters written to *stdout*.

If unsuccessful, vprintf() returns a negative value.

Note: In contrast to some UNIX-based implementations of the C language, the z/OS XL C/C++ implementation of the vprintf() family increments the pointer to the variable arguments list. To control whether the pointer to the argument is incremented, call the `va_end` macro after each call to vprintf().

Example

CELEBV04

```
/* CELEBV04

   This example prints out a variable number of strings to stdout,
   using &vprintf..

 */
#include <stdarg.h>
#include <stdio.h>

void vout(char *fmt, ...);
char fmt1 [] = "%s  %s  %s   %s   %s \n";
```

```
int main(void)
{
   FILE *stream;
   stream = fopen("myfile.dat", "w");

   vout(fmt1, "Mon", "Tues", "Wed", "Thurs", "Fri");
}

void vout(char *fmt, ...)
{
   va_list arg_ptr;

   va_start(arg_ptr, fmt);
   vprintf(fmt, arg_ptr);
   va_end(arg_ptr);
}
```

Output

```
Mon  Tues  Wed   Thurs   Fri
```

Related information

- “stdarg.h” on page 64
- “stdio.h” on page 68
- “va_arg(), va_copy(), va_end(), va_start() — Access function arguments” on page 1973
- “vfprintf() — Format and print data to stream” on page 1982
- “vsprintf() — Format and print data to buffer” on page 1991

vsnprintf() — Format and print data to fixed length buffer

Standards

Standards / Extensions	C or C++	Dependencies
C99 Single UNIX Specification, Version 3 C++ TR1 C99	both	z/OS V1R6

Format

```
#define _ISOC99_SOURCE
#include <stdio.h>
#include <stdarg.h>

int  vsnprintf(char *__restrict__ s, size_t n,
               const char *__restrict__ format, va_list arg);
```

General description

Equivalent to snprintf(), except that instead of being called with a variable number of arguments, it is called with an argument list as defined by <stdarg.h>.

Returned value

Returns the number of characters that would have been written had *n* been sufficiently large, not counting the terminating null character, or a negative value if an encoding error occurred. Thus, the null-terminated output has been completely written if and only if the returned value is nonnegative and less than *n*.

vsprintf() — Format and print data to buffer

Standards

Standards / Extensions	C or C++	Dependencies
ISO C XPG4.2 C99 Single UNIX Specification, Version 3	both	

Format

```
#include <stdarg.h>
#include <stdio.h>

int vsprintf(char *  __restrict__target-string,
             const char *  __restrict__format, va_list arg_ptr);
```

General description

The vsprintf() function is similar to sprintf(), except that *arg_ptr* points to a list of arguments whose number can vary from call to call in the program. In contrast, sprintf() can have a list of arguments, but the number of arguments in that list is fixed when you compile the program. For a specification of the *format* string, see "fprintf(), printf(), sprintf() — Format and write data" on page 588.

Returned value

If successful, vsprintf() returns the number of characters written *target-string*.

If unsuccessful, vsprintf() returns a negative value.

Note: In contrast to some UNIX-based implementations of the C language, the z/OS XL C/C++ implementation of the vprintf() family increments the pointer to the variable arguments list. To control whether the pointer to the argument is incremented, call the va_end macro after each call to vsprintf().

Example

CELEBV05

```
/* CELEBV05

   This example assigns a variable number of strings to string
   and prints the resultant string, using &vsprintf..

 */
#include <stdarg.h>
#include <stdio.h>

void vout(char *string, char *fmt, ...);
char fmt1 [] = "%s  %s  %s\n";

int main(void)
{
   char string[100];

   vout(string, fmt1, "Sat", "Sun", "Mon");
   printf("The string is:  %s\n", string);
}
void vout(char *string, char *fmt, ...)
```

```
{
   va_list arg_ptr;

   va_start(arg_ptr, fmt);
   vsprintf(string, fmt, arg_ptr);
   va_end(arg_ptr);
}
```

Output

```
The string is:  Sat  Sun  Mon
```

Related information

- “stdarg.h” on page 64
- “stdio.h” on page 68
- “va_arg(), va_copy(), va_end(), va_start() — Access function arguments” on page 1973
- “vfprintf() — Format and print data to stream” on page 1982
- “vprintf() — Format and print data to stdout” on page 1988

vswprintf() — Format and write wide characters of a stdarg argument list

The information for this function is included in “vfwprintf(), vswprintf(), vwprintf() — Format and write wide characters of a STDARG argument list” on page 1985.

vwprintf() — Format and write wide characters of a stdarg argument list

The information for this function is included in “vfwprintf(), vswprintf(), vwprintf() — Format and write wide characters of a STDARG argument list” on page 1985.

wait() — Wait for a child process to end

Standards

Standards / Extensions	C or C++	Dependencies
POSIX.1 XPG4 XPG4.2 Single UNIX Specification, Version 3	both	

Format

```
#define _POSIX_SOURCE
#include <sys/wait.h>

pid_t wait(int *status_ptr);
```

General description

Suspends the calling process until any one of its child processes ends. More precisely, wait() suspends the calling process until the system obtains status information on the ended child. If the system already has status information on a completed child process when wait() is called, wait() returns immediately. wait() is also ended if the calling process receives a signal whose action is either to execute a signal handler or to end the process.

The argument *status_ptr* points to a location where wait() can store a status value. This status value is zero if the child process explicitly returns zero status. If it is not zero, it can be analyzed with the status analysis macros, described in "Status Analysis Macros," below.

The *status_ptr* pointer may also be NULL, in which case wait() ignores the child's return status.

The following function calls are equivalent:

```
wait(status_ptr);
waitpid(-1,status_ptr,0);
wait3(status_ptr,0,NULL);
```

For more information, see "waitpid() — Wait for a specific child process to end" on page 1997.

Special behavior for XPG4.2: If the calling process has SA_NOCLDWAIT set or has SIGCHLD set to SIG_IGN, and the process has no unwaited for children that were transformed into zombie processes, it will block until all of the children terminate, and wait() will fail and set errno to ECHILD.

Status analysis macros: If the *status_ptr* argument is not NULL, wait() places the child's return status in **status_ptr*. You can analyze this return status with the following macros, defined in the sys/wait.h header file:

WIFEXITED(**status_ptr***)**
: This macro evaluates to a nonzero (true) value if the child process ended normally, that is, if it returned from main() or called one of the exit() or _exit() functions.

WEXITSTATUS(**status_ptr***)**
: When WIFEXITED() is nonzero, WEXITSTATUS() evaluates to the low-order 8 bits of the child's return status passed on the exit() or _exit() function.

WIFSIGNALED(**status_ptr***)**
: This macro evaluates to a nonzero (true) value if the child process ended because of a signal that was not caught.

WIFSTOPPED(**status_ptr***)**
: This macro evaluates to a nonzero (true) value if the child process is currently stopped. This should only be used after a waitpid() with the WUNTRACED option.

WSTOPSIG(**status_ptr***)**
: When WIFSTOPPED() is nonzero, WSTOPSIG() evaluates to the number of the signal that stopped the child.

WTERMSIG(**status_ptr***)**
: When WIFSIGNALED() is nonzero, WTERMSIG() evaluates to the number of the signal that ended the child process.

Returned value

If successful, wait() returns a value that is the process ID (PID) of the child whose status information has been obtained.

If unsuccessful, wait() returns -1 and sets errno to one of the following values:

Error Code
Description

ECHILD
The caller has no appropriate child processes, that is, it has no child processes whose status has not been obtained by previous calls to wait(), waitid(), waitpid(), or wait3(). ECHILD is also returned when the SA_NOCLDWAIT flag is set.

EINTR
wait() was interrupted by a signal. The value of **status_ptr* is undefined.

Example

CELEBW01

```
/* CELEBW01

   This example suspends the calling process until any child processes ends.

 */
#define _POSIX_SOURCE
#include <sys/types.h>
#include <sys/wait.h>
#include <unistd.h>
#include <stdio.h>
#include <time.h>

main() {
  pid_t pid;
  time_t t;
  int status;

  if ((pid = fork()) < 0)
    perror("fork() error");
  else if (pid == 0) {
    time(&t);
    printf("child (pid %d) started at %s", (int) getpid(), ctime(&t));
    sleep(5);
    time(&t);
    printf("child exiting at %s", ctime(&t));
    exit(42);
  }
  else {
    printf("parent has forked child with pid of %d\n", (int) pid);
    time(&t);
    printf("parent is starting wait at %s", ctime(&t));
    if ((pid = wait(&status)) == -1)
      perror("wait() error");
    else {
      time(&t);
      printf("parent is done waiting at %s", ctime(&t));
      printf("the pid of the process that ended was %d\n", (int) pid);
      if (WIFEXITED(status))
        printf("child exited with status of %d\n", WEXITSTATUS(status));
      else if (WIFSIGNALED(status))
        printf("child was terminated by signal %d\n",
               WTERMSIG(status));
      else if (WIFSTOPPED(status))
        printf("child was stopped by signal %d\n", WSTOPSIG(status));
      else puts("reason unknown for child termination");
    }
  }
}
```

Output:

```
parent has forked child with pid of 65546
parent is starting wait at Fri Jun 16 10:53:03 2006
child (pid 65546) started at Fri Jun 16 10:53:04 2006
child exiting at Fri Jun 16 10:53:09 2006
parent is done waiting at Fri Jun 16 10:53:10 2006
the pid of the process that ended was 65546
child exited with status of 42
```

Related information

- "signal.h" on page 63
- "sys/types.h" on page 75
- "sys/wait.h" on page 77
- "exit() — End program" on page 443
- "_exit() — End a process and bypass the cleanup" on page 445
- "fork() — Create a new process" on page 571
- "pause() — Suspend a process pending a signal" on page 1173
- "waitid() — Wait for child process to change state"
- "waitpid() — Wait for a specific child process to end" on page 1997
- "wait3() — Wait for child process to change state" on page 2000

waitid() — Wait for child process to change state

Standards

Standards / Extensions	C or C++	Dependencies
XPG4.2 Single UNIX Specification, Version 3	both	

Format

```
#define _XOPEN_SOURCE_EXTENDED 1
#include <sys/wait.h>

int waitid(idtype_t idtype, id_t id, siginfo_t *infop, int options);
```

General description

The waitid() function suspends the calling process until one of its children changes state. It records the current state of a child in the structure pointed to by *infop*. If a child process changed state before the call, waitid() returns immediately.

The *idtype* and *id* arguments are used to specify which children waitid() will wait for.

If *idtype* is P_PID waitid() will wait for the child with a process ID equal to (pid_t)*id*.

If *idtype* is P_GID waitid() will wait for any child with a process group ID equal to (pid_t)*id*.

If *idtype* is P_ALL waitid() will wait for any children and *id* is ignored.

The *options* argument is used to specify which state changes to wait for. It is formed by OR-ing together one or more of the following flags:

WCONTINUED
Status will be returned for any child that has stopped and has been continued.

WEXITED
Wait for processes that have exited.

WNOHANG
Return immediately if there are no children to wait for.

WNOWAIT
Keep the process whose status is returned in *infop* in a waitable state. This will not affect the state of the process; the process may be waited for again after this call completes.

WSTOPPED
Status will be returned for any child that has stopped upon receipt of a signal.

The *infop* argument must point to a siginfo_t structure. If waitid() returns because a child process was found that specified the conditions indicated by the arguments *idtype* and *options* then the structure pointed to by *infop* will be filled in by the system with the status of the process. The si_signo member will always be equal to SIGCHLD.

Returned value

If waitid() returns due to the change of state of one of its children, it returns 0.

If unsuccessful, waitid() returns -1 and sets errno to one of the following values:

Error Code
Description

ECHILD
The calling process has no existing unwaited-for child processes.

EINTR
The waitid() function was interrupted due to the receipt of a signal by the calling process.

EINVAL
An invalid value was specified for *options*, or *idtype* and *id* specify an invalid set of processes.

Related information

- "sys/wait.h" on page 77
- "exec functions" on page 436
- "exit() — End program" on page 443
- "wait() — Wait for a child process to end" on page 1992

waitpid() — Wait for a specific child process to end

Standards

Standards / Extensions	C or C++	Dependencies
POSIX.1 XPG4 XPG4.2 Single UNIX Specification, Version 3	both	

Format

```
#define _POSIX_SOURCE
#include <sys/wait.h>

pid_t waitpid(pid_t pid, int *status_ptr, int options);
```

General description

Suspends the calling process until a child process ends or is stopped. More precisely, waitpid() suspends the calling process until the system gets status information on the child. If the system already has status information on an appropriate child when waitpid() is called, waitpid() returns immediately. waitpid() is also ended if the calling process receives a signal whose action is either to execute a signal handler or to end the process.

pid_t *pid*

Specifies the child processes the caller wants to wait for:

- If *pid* is greater than 0, waitpid() waits for termination of the specific child whose process ID is equal to *pid*.
- If *pid* is equal to zero, waitpid() waits for termination of any child whose process group ID is equal to that of the caller.
- If *pid* is -1, waitpid() waits for any child process to end.
- If *pid* is less than -1, waitpid() waits for the termination of any child whose process group ID is equal to the absolute value of *pid*.

int **status_ptr*

Points to a location where waitpid() can store a status value. This status value is zero if the child process explicitly returns zero status. Otherwise, it is a value that can be analyzed with the status analysis macros described in “Status Analysis Macros”, below.

The *status_ptr* pointer may also be NULL, in which case waitpid() ignores the child's return status.

int *options*

Specifies additional information for waitpid(). The *options* value is constructed from the bitwise inclusive-OR of zero or more of the following flags defined in the sys/wait.h header file:

WCONTINUED

Special behavior for XPG4.2: Reports the status of any continued child processes as well as terminated ones. The WIFCONTINUED macro lets a process distinguish between a continued process and a terminated one.

WNOHANG

Demands status information immediately. If status information is immediately available on an appropriate child process, waitpid() returns this information. Otherwise, waitpid() returns immediately with an error code indicating that the information was not available. In other words, WNOHANG checks child processes without causing the caller to be suspended.

WUNTRACED
Reports on stopped child processes as well as terminated ones. The WIFSTOPPED macro lets a process distinguish between a stopped process and a terminated one.

Special behavior for XPG4.2: If the calling process has SA_NOCLDWAIT set or has SIGCHLD set to SIG_IGN, and the process has no unwaited for children that were transformed into zombie processes, it will block until all of the children terminate, and waitpid() will fail and set errno to ECHILD.

Status analysis macros: If the *status_ptr* argument is not NULL, waitpid() places the child's return status in **status_ptr*. You can analyze this return status with the following macros, defined in the sys/wait.h header file:

WEXITSTATUS(**status_ptr***)**
When WIFEXITED() is nonzero, WEXITSTATUS() evaluates to the low-order 8 bits of the status argument that the child passed to the exit() or _exit() function, or the value the child process returned from main().

WIFCONTINUED(**status_ptr***)**
Special behavior for XPG4.2: This macro evaluates to a nonzero (true) value if the child process has continued from a job control stop. This should only be used after a waitpid() with the WCONTINUED option.

WIFEXITED(**status_ptr***)**
This macro evaluates to a nonzero (true) value if the child process ended normally (that is, if it returned from main(), or else called the exit() or _exit() function).

WIFSIGNALED(**status_ptr***)**
This macro evaluates to a nonzero (true) value if the child process ended because of a signal that was not caught.

WIFSTOPPED(**status_ptr***)**
This macro evaluates to a nonzero (true) value if the child process is currently stopped. This should only be used after a waitpid() with the WUNTRACED option.

WSTOPSIG(**status_ptr***)**
When WIFSTOPPED() is nonzero, WSTOPSIG() evaluates to the number of the signal that stopped the child.

WTERMSIG(**status_ptr***)**
When WIFSIGNALED() is nonzero, WTERMSIG() evaluates to the number of the signal that ended the child process.

Returned value

If successful, waitpid() returns a value of the process (usually a child) whose status information has been obtained.

If WNOHANG was given, and if there is at least one process (usually a child) whose status information is not available, waitpid() returns 0.

If unsuccessful, waitpid() returns -1 and sets errno to one of the following values:

Error Code
Description

ECHILD
The process specified by *pid* does not exist or is not a child of the calling process, or the process group specified by *pid* does not exist or does not have any member process that is a child of the calling process.

EINTR
waitpid() was interrupted by a signal. The value of **status_ptr* is undefined.

EINVAL
The value of *options* is incorrect.

Example

CELEBW02

```
/* CELEBW02

   The following function suspends the calling process using &waitpid.
   until a child process ends.

 */
#define _POSIX_SOURCE
#include <sys/types.h>
#include <sys/wait.h>
#include <unistd.h>
#include <stdio.h>
#include <time.h>

main() {
  pid_t pid;
  time_t t;
  int status;

  if ((pid = fork()) < 0)
    perror("fork() error");
  else if (pid == 0) {
    sleep(5);
    exit(1);
  }
  else do {
    if ((pid = waitpid(pid, &status, WNOHANG)) == -1)
      perror("wait() error");
    else if (pid == 0) {
      time(&t);
      printf("child is still running at %s", ctime(&t));
      sleep(1);
    }
    else {
      if (WIFEXITED(status))
        printf("child exited with status of %d\n", WEXITSTATUS(status));
      else puts("child did not exit successfully");
    }
  } while (pid == 0);
}
```

Output:

```
child is still running at Fri Jun 16 11:05:43 2006
child is still running at Fri Jun 16 11:05:44 2006
child is still running at Fri Jun 16 11:05:45 2006
child is still running at Fri Jun 16 11:05:46 2006
child is still running at Fri Jun 16 11:05:47 2006
child is still running at Fri Jun 16 11:05:48 2006
child is still running at Fri Jun 16 11:05:49 2006
child exited with status of 1
```

Related information

- "signal.h" on page 63
- "sys/types.h" on page 75
- "sys/wait.h" on page 77
- "exit() — End program" on page 443
- "_exit() — End a process and bypass the cleanup" on page 445
- "fork() — Create a new process" on page 571
- "pause() — Suspend a process pending a signal" on page 1173
- "wait() — Wait for a child process to end" on page 1992
- "waitid() — Wait for child process to change state" on page 1995
- "wait3() — Wait for child process to change state"

wait3() — Wait for child process to change state

Standards

Standards / Extensions	C or C++	Dependencies
XPG4.2	both	

Format

```
#define _XOPEN_SOURCE_EXTENDED 1
#include <sys/wait.h>

pid_t wait3(int *stat_loc, int options, struct rusage *resource_usage);
```

General description

The wait3() function allows the calling process to obtain status information for specified child processes.

The following call:

```
   wait3(stat_loc, options, resource_usage)
```

is equivalent to the call:

```
   waitpid((pid_t)-1, stat_loc, options);
```

except that on successful completion, if the *resource_usage* argument to wait3() is not a NULL pointer, the rusage structure that the third argument points to is filled in for the child process identified by the return value.

Note:

This function is kept for historical reasons. It was part of the Legacy Feature in Single UNIX Specification, Version 2, but has been withdrawn and is not supported as part of Single UNIX Specification, Version 3. The waitpid() function is preferred for portability.

If it is necessary to continue using this function in an application written for Single UNIX Specification, Version 3, define the feature test macro

_UNIX03_WITHDRAWN before including any standard system headers. The macro exposes all interfaces and symbols removed in Single UNIX Specification, Version 3.

Returned value

See “waitpid() — Wait for a specific child process to end” on page 1997.

In addition to the error conditions specified on waitpid(), under the following conditions, wait3() may fail and set errno to one of the following values:

Error Code
  Description

ECHILD
  The calling process has no existing unwaited-for child processes, or if the set of processes specified by the argument *pid* can never be in the states specified by the argument *options*.

Related information

- “sys/wait.h” on page 77
- “exec functions” on page 436
- “exit() — End program” on page 443
- “fork() — Create a new process” on page 571
- “pause() — Suspend a process pending a signal” on page 1173
- “waitpid() — Wait for a specific child process to end” on page 1997

wcrtomb() — Convert a wide character to a multibyte character

Standards

Standards / Extensions	C or C++	Dependencies
ISO C Amendment C99 Single UNIX Specification, Version 3	both	

Format

Non-XPG4:

```
#include <wchar.h>

size_t wcrtomb(char * __restrict__s, wchar_t wchar, mbstate_t * __restrict__pss);
```

XPG4:

```
#define _XOPEN_SOURCE
#define _MSE_PROTOS
#include <wchar.h>

size_t wcrtomb(char *s, wchar_t wchar, mbstate_t *pss);
```

General description

If *s* is a NULL pointer, the wcrtomb() function determines the number of bytes necessary to enter the initial shift state (zero if encodings are not state-dependent or if the initial conversion state is described). The resulting state described is the initial conversion state.

If *s* is not a NULL pointer, the wcrtomb() function determines the number of bytes needed to represent the multibyte character that corresponds to the wide character given by `wchar` (including any shift sequences), and stores the resulting bytes in the array whose first element is pointed to by *s*. At most, MB_CUR_MAX bytes are stored. If `wchar` is a NULL wide character, the resulting state described is the initial conversion state.

wcrtomb() is a “restartable” version of wctomb(). That is, shift state information is passed as one of the arguments and is updated on return. With wcrtomb(), you can switch from one multibyte string to another, provided that you have kept the shift-state information for each multibyte string.

The behavior of this wide-character function is affected by the LC_CTYPE category of the current locale. If you change the category, undefined results can occur.

Special behavior for XPG4: If you define any feature test macro specifying XPG4 behavior before the statement in your program source file to include the wchar header, then you must also define the _MSE_PROTOS feature test macro to make the declaration of the wcrtomb() function in the wchar header available when you compile your program. Please see Table 2 on page 4 for a list of XPG4 and other feature test macros.

Returned value

If *s* is a NULL pointer, wcrtomb() returns the number of bytes needed to enter the initial shift state. The value returned will not be greater than that of MB_CUR_MAX.

If *s* is not a NULL pointer, wcrtomb() returns the number of bytes stored in the array object (including any shift sequences) when `wchar` is a valid wide character. Otherwise, when `wchar` is not a valid wide character, an encoding error occurs, the value of the macro EILSEQ is stored in errno and -1 is returned, but the conversion state remains unchanged.

Example

```
#include <stdio.h>
#include <wchar.h>

int main(void)
{
   char    *string;
   wchar_t wc;
   size_t  length;
   length = wcrtomb(string, wc, NULL);
}
```

Related information

- “wchar.h” on page 85
- “mblen() — Calculate length of multibyte character” on page 1040
- “mbrlen() — Calculate length of multibyte character” on page 1043
- “mbrtowc() — Convert a multibyte character to a wide character” on page 1049
- “mbsrtowcs() — Convert a multibyte string to a wide-character string” on page 1052
- “wcsrtombs() — Convert wide-character string to multibyte string” on page 2021
- “wctomb() — Convert wide character to multibyte character” on page 2053

wcscat() — Append to wide-character string

Standards

Standards / Extensions	C or C++	Dependencies
ISO C Amendment XPG4 XPG4.2 C99 Single UNIX Specification, Version 3	both	

Format

```
#include <wchar.h>

wchar_t *wcscat(wchar_t * __restrict__string1, const wchar_t * __restrict__string2);
```

General description

Appends a copy of the string pointed to by *string2* to the end of the string pointed to by *string1*.

The wcscat() function operates on NULL-terminated wide-character strings. The string arguments to this function must contain a wide NULL character marking the end of the string. Bounds checking is not performed.

The behavior of this wide-character function is affected by the LC_CTYPE category of the current locale. If you change the category, undefined results can occur.

Returned value

wcscat() returns the value of *string1*.

Example

CELEBW04

```
/* CELEBW04

   This example creates the wide character string "computer
   program" using &wcscat..

 */
#include <stdio.h>
#include <wchar.h>

#define SIZE 40

int main(void)
{
  wchar_t buffer1[SIZE] = L"computer";
  wchar_t * string      = L" program";
  wchar_t * ptr;

  ptr = wcscat( buffer1, string );
  printf( "buffer1 = %ls\n", buffer1 );
}
```

Output:

```
buffer1 = computer program
```

Related information

- "wchar.h" on page 85
- "wcstr.h" on page 86
- "strcat() — Concatenate strings" on page 1728
- "wcschr() — Search for wide-character substring"
- "wcscmp() — Compare wide-character strings" on page 2005
- "wcscpy() — Copy wide-character string" on page 2008
- "wcscspn() — Find offset of first wide-character match" on page 2009
- "wcslen() — Calculate length of wide-character string" on page 2013
- "wcsncat() — Append to wide-character string" on page 2014

wcschr() — Search for wide-character substring

Standards

Standards / Extensions	C or C++	Dependencies
ISO C Amendment SAA XPG4 XPG4.2 C99 Single UNIX Specification, Version 3	both	

Format

```
#include <wchar.h>

wchar_t *wcschr(const wchar_t *string1, wchar_t character);
```

General description

Searches *string* for the occurrence of *character*. The *character* may be a wide NULL character (\0). The wide NULL character at the end of *string* is included in the search. The wcschr() function operates on NULL-terminated wide-character strings. The argument to this function must contain a wide NULL character marking the end of the string.

The behavior of this wide-character function is affected by the LC_CTYPE category of the current locale. If you change the category, undefined results can occur.

Returned value

If successful, wcschr() returns a pointer to the first occurrence of *character* in *string*.

If the character is not found, wcschr() returns a NULL pointer.

Example

CELEBW05

```
/* CELEBW05

   This example finds the first occurrence of the character p in
   the wide character string "computer program" using &wcschr..

 */
#include <stdio.h>
#include <wchar.h>
```

```
#define SIZE 40

int main(void)
{
  wchar_t buffer1[SIZE] = L"computer program";
  wchar_t * ptr;
  wint_t ch = L'p';

  ptr = wcschr( buffer1, ch );
  printf( "The first occurrence of %lc in '%ls' is '%ls'\n",
                          ch, buffer1, ptr );

}
```

Output:

```
The first occurrence of p in 'computer program' is 'puter program'
```

Related information

- "wchar.h" on page 85
- "wcstr.h" on page 86
- "strchr() — Search for character" on page 1729
- "wcscat() — Append to wide-character string" on page 2003
- "wcscmp() — Compare wide-character strings"
- "wcscpy() — Copy wide-character string" on page 2008
- "wcscspn() — Find offset of first wide-character match" on page 2009
- "wcslen() — Calculate length of wide-character string" on page 2013
- "wcsncmp() — Compare wide-character strings" on page 2015

wcscmp() — Compare wide-character strings

Standards

Standards / Extensions	C or C++	Dependencies
ISO C Amendment XPG4 XPG4.2 C99 Single UNIX Specification, Version 3	both	

Format

```
#include <wchar.h>

int wcscmp(const wchar_t *string1, const wchar_t *string2);
```

General description

Compares two wide-character strings. The wcscmp() function operates on NULL-terminated wide-character strings. The string arguments to this function must contain a wide NULL character marking the end of the string.

The behavior of this wide-character function is affected by the LC_CTYPE category of the current locale. If you change the category, undefined results can occur.

Returned value

wcscmp() returns a value indicating the relationship between the two strings, as follows:

Value	Meaning
< 0	string pointed to by *string1* less than string pointed to by *string2*
= 0	string pointed to by *string1* identical to string pointed to by *string2*
> 0	string pointed to by *string1* greater than string pointed to by *string2*

Example

CELEBW06

```
/* CELEBW06

   This example compares the wide character string string1 to
   string2 using &wcscmp..

 */
#include <stdio.h>
#include <wchar.h>

int main(void)
{
  int  result;
  wchar_t string1[] = L"abcdef";
  wchar_t string2[] = L"abcdefg";

  result = wcscmp( string1, string2 );

  if ( result == 0 )
    printf( "\"%ls\" is identical to \"%ls\"\n", string1, string2);
  else if ( result < 0 )
    printf( "\"%ls\" is less than \"%ls\"\n", string1, string2 );
  else
    printf( "\"%ls\" is greater than \"%ls\"\n", string1, string2);
}
```

Output:

```
"abcdef" is less than "abcdefg"
```

Related information

- “wchar.h” on page 85
- “wcstr.h” on page 86
- “strcmp() — Compare strings” on page 1731
- “wcscat() — Append to wide-character string” on page 2003
- “wcschr() — Search for wide-character substring” on page 2004
- “wcscpy() — Copy wide-character string” on page 2008
- “wcscspn() — Find offset of first wide-character match” on page 2009
- “wcslen() — Calculate length of wide-character string” on page 2013
- “wcsncmp() — Compare wide-character strings” on page 2015

wcscoll() — Language collation string comparison

Standards

Standards / Extensions	C or C++	Dependencies
ISO C Amendment XPG4 XPG4.2 C99 Single UNIX Specification, Version 3	both	

Format

```
#include <wchar.h>

int wcscoll(const wchar_t *wcs1, const wchar_t *wcs2);
```

General description

Compares the wide-character string pointed to by wcs1 to the wide-character string pointed to by wcs2, both interpreted as appropriate to the LC_COLLATE category of the current locale.

Returned value

wcscoll() returns an integer greater than, equal to, or less than zero, according to whether the wide string pointed to by wcs1 is greater than, equal to, or less than the wide-character string pointed to by wcs2, when both wide-character strings are interpreted as appropriate to the LC_COLLATE category of the current locale.

wcscoll() differs from wcscmp(). wcscoll() function performs a comparison between two wide character strings based on language collation rules as controlled by the LC_COLLATE category. On the other hand, wcscmp() performs a wide-character code to wide-character code comparison.

wcscoll() indicates error conditions by setting errno; however, there is no returned value to indicate an error. To check for errors, errno should be set to zero, and then checked upon return from wcscoll(). If errno is nonzero, an error has occurred.

The EILSEQ error can be set to indicate that the wcs1 or wcs2 arguments contain characters outside the domain of the collating sequence.

Note: The ISO/C Multibyte Support Extensions do not indicate that the wcscoll() function may return with an error.

Example

CELEBW07

```
/* CELEBW07 */
#include <stdio.h>
#include <wchar.h>

int main(void)
{
   int result;
   wchar_t *wcs1 = L"first_wide_string";
   wchar_t *wcs2 = L"second_wide_string";

   result = wcscoll(wcs1, wcs2);

   if ( result == 0)
      printf("\"%ls\" is identical to \"%ls\"\n", wcs1, wcs2);
   else if ( result < 0)
      printf("\"%ls\" is less than \"%ls\"\n", wcs1, wcs2);
   else
      printf("\"%ls\" is greater than \"%ls\"\n", wcs1, wcs2);
}
```

Output:

```
"first_wide_string" is less than "second_wide_string"
```

Related information

- "wchar.h" on page 85
- "setlocale() — Set locale" on page 1552
- "strcoll() — Compare strings" on page 1732

wcscpy() — Copy wide-character string

Standards

Standards / Extensions	C or C++	Dependencies
ISO C Amendment XPG4 XPG4.2 C99 Single UNIX Specification, Version 3	both	

Format

```
#include <wchar.h>

wchar_t *wcscpy(wchar_t * __restrict__string1, const wchar_t * __restrict__string2);
```

General description

Copies the contents of *string2* (including the ending wide NULL character) into *string1*. The wcscpy() function operates on NULL-terminated wide-character strings. The string arguments to this function must contain a wide NULL character marking the end of the string. Bounds checking is not performed.

The behavior of this wide-character function is affected by the LC_CTYPE category of the current locale. If you change the category, undefined results can occur.

Returned value

wcscpy() returns the value of *string1*.

Example

CELEBW08

```
/* CELEBW08

   This example copies the contents of source to destination using
   wcscpy().

 */
#include <stdio.h>
#include <wchar.h>

#define SIZE    40

int main(void)
{
  wchar_t source[ SIZE ] = L"This is the source string";
  wchar_t destination[ SIZE ] = L"And this is the destination string";
  wchar_t * return_string;

  printf( "destination is originally = \"%ls\"\n", destination );
  return_string = wcscpy( destination, source );
  printf( "After wcscpy, destination becomes \"%ls\"\n", destination );
}
```

Output:

```
destination is originally = "And this is the destination string"
After wcscpy, destination becomes "This is the source string"
```

Related information

- “wchar.h” on page 85
- “wcstr.h” on page 86
- “strcpy() — Copy string” on page 1734
- “wcscat() — Append to wide-character string” on page 2003
- “wcschr() — Search for wide-character substring” on page 2004
- “wcscmp() — Compare wide-character strings” on page 2005
- “wcscspn() — Find offset of first wide-character match”
- “wcslen() — Calculate length of wide-character string” on page 2013
- “wcsncpy() — Copy wide-character string” on page 2017

wcscspn() — Find offset of first wide-character match

Standards

Standards / Extensions	C or C++	Dependencies
ISO C Amendment XPG4 XPG4.2 C99 Single UNIX Specification, Version 3	both	

Format

```
#include <wchar.h>

size_t wcscspn(const wchar_t *string1, const wchar_t *string2);
```

General description

Determines the number of wide characters in the initial segment of the string pointed to by *string1* that do not appear in the string pointed to by *string2*. The wcscspn() function operates on NULL-terminated wide-character strings. The string arguments to these functions must contain a NULL wide character marking the end of the string.

The behavior of this wide-character function is affected by the LC_CTYPE category of the current locale. If you change the category, undefined results can occur.

Returned value

wcscspn() returns the number of wide characters in the segment.

Example

CELEBW09

```
/* CELEBW09

   This example uses &wcscspn. to find the first occurrence of
   any of the characters a, x, l, or e in string.

 */
#include <stdio.h>
```

```
#include <wchar.h>

#define SIZE    40

int main(void)
{
  wchar_t string[ SIZE ] = L"This is the source string";
  wchar_t * substring = L"axle";

  printf( "The first %i characters in the string \"%ls\" are not in the "
          "string \"%ls\" \n", wcscspn( string, substring),
          string, substring );
}
```

Output:

```
The first 10 characters in the string "This is the source string" are not
in the string "axle"
```

Related information

- "wchar.h" on page 85
- "wcstr.h" on page 86
- "strcspn() — Compare strings" on page 1735
- "wcscat() — Append to wide-character string" on page 2003
- "wcschr() — Search for wide-character substring" on page 2004
- "wcscmp() — Compare wide-character strings" on page 2005
- "wcscpy() — Copy wide-character string" on page 2008
- "wcslen() — Calculate length of wide-character string" on page 2013

wcsftime() — Format date and time

Standards

Standards / Extensions	C or C++	Dependencies
ISO C Amendment XPG4 XPG4.2 C99 Single UNIX Specification, Version 3	both	

Format

Non-XPG4:

```
#include <wchar.h>

size_t wcsftime(wchar_t * __restrict__ wcs, size_t maxsize,
                const wchar_t * __restrict__ format,
                const struct tm * __restrict__ time_ptr)
```

XPG4:

```
#define _XOPEN_SOURCE
#include <wchar.h>

size_t wcsftime(wchar_t * __restrict__ wcs, size_t maxsize,
                const char * __restrict__ format,
                const struct tm * __restrict__ time_ptr)
```

XPG4 and MSE:

```
#define _XOPEN_SOURCE
#define _MSE_PROTOS
#include <wchar.h>

size_t wcsftime(wchar_t * __restrict__ wcs, size_t maxsize,
                const wchar_t * __restrict__ format,
                const struct tm * __restrict__ time_ptr)
```

General description

Format date and time into a wide character string. The wcsftime() function is equivalent to the strftime() function, except that:

- The argument *wcs* specifies an array of a wide string into which the generated output is to be placed.
- The argument *maxsize* indicates a number of wide characters.
- The argument **format* specifies an array of wide characters comprising the format string.
- The returned value indicates a number of wide characters.

Special behavior for XPG4: If you define any feature test macro specifying XPG4 behavior before the statement in your program source file to include the wchar header, then the compiler assumes that your program is using the XPG4 variety of the wcsftime() function unless you also define the _MSE_PROTOS feature test macro. Please see Table 2 on page 4 for a list of XPG4 and other feature test macros.

The prototype for the XPG4 variety of the wcsftime() function is:

```
size_t wcsftime(wchar_t *wcs, size_t maxsize, const char *format,
                const struct tm *time_ptr)
```

The difference between this variety and the MSE variety of the wcsftime() function is that the third argument **format* specifies an array of characters rather than an array of wide characters comprising the format string.

Returned value

If the total number of resulting wide characters including the terminating NULL wide character is not more than *maxsize,* wcsftime() returns the number of wide characters placed into the array pointed to by *wcs* not including the terminating NULL wide character.

If unsuccessful, wcsftime() returns 0 and the contents of the array are indeterminate.

Example

CELEBW10

```
/* CELEBW10 */
#include <stdio.h>
#include <time.h>
#include <wchar.h>

int main(void)
{
   struct tm *timeptr;
   wchar_t dest[100];
   time_t  temp;
   size_t  rc;
   temp = time(NULL);
   timeptr = localtime(&temp);
   rc = wcsftime(dest, sizeof(dest)-1, L" Today is %A,"
                 L" %b %d.\n Time: %I:%M %p", timeptr);
   printf("%d characters placed in string to make:\n\n%S", rc, dest);
}
```

Output:

```
42 characters placed in string to make:

 Today is Friday, Jun 16.
 Time: 01:48 pm
```

Related information

- "wchar.h" on page 85
- "strftime() — Convert to formatted time" on page 1743

wcsid() — Character set ID for wide character

Standards

Standards / Extensions	C or C++	Dependencies
C Library	both	

Format

```
#include <stdlib.h>

int wcsid(const wchar_t c)
```

External entry point: @@WCSID, __wcsid

General description

Determines the character set identifier for the specified wide character.

To avoid infringing on the user's name space, this nonstandard function has two names. One name is prefixed with two underscore characters, and one name is not. The name without the prefix underscore characters is exposed only when you use LANGLVL(EXTENDED).

To use this function, you must either invoke the function using its external entry point name (that is, the name that begins with two underscore characters), or compile with LANGLVL(EXTENDED). When you use LANGLVL(EXTENDED) any relevant information in the header is also exposed.

Returned value

If successful, wcsid() returns the character set identifier for the wide character.

If the wide character is not valid, wcsid() returns -1.

Example

CELEBW11

```
/* CELEBW11

   This example checks character set id for wide character.

*/
#include <locale.h>
#include <stdio.h>
#include <stdlib.h>

main() {
   wchar_t wc = L'A';
   int   rc;

   rc = wcsid(wc);
   printf("wide character '%lc' is in character set id %i\n", wc, rc);
}
```

Related information

- "stdlib.h" on page 70

wcslen() — Calculate length of wide-character string

Standards

Standards / Extensions	C or C++	Dependencies
ISO C Amendment XPG4 XPG4.2 C99 Single UNIX Specification, Version 3	both	

Format

```
#include <wchar.h> /*  or #include <wcstr.h>  */

size_t wcslen(const wchar_t *string);
```

General description

Computes the number of wide characters in the string pointed to by *string*.

The behavior of this wide-character function is affected by the LC_CTYPE category of the current locale. If you change the category, undefined results can occur.

Returned value

wcslen() returns the number of wide characters that precede the terminating wide NULL character.

Example

CELEBW12

```
/* CELEBW12

   This example computes the length of a wide-character string,
   using &wcslen.

 */
#include <stdio.h>
#include <wchar.h>
```

```
int main(void)
{
  wchar_t * string = L"abcdef";

  printf( "Length of \"%ls\" is %i\n", string, wcslen( string ));
}
```

Output:

```
Length of "abcdef" is 6
```

Related information

- "wchar.h" on page 85
- "wcstr.h" on page 86
- "mblen() — Calculate length of multibyte character" on page 1040
- "strlen() — Determine string length" on page 1748
- "wcsncat() — Append to wide-character string"
- "wcsncmp() — Compare wide-character strings" on page 2015
- "wcsncpy() — Copy wide-character string" on page 2017

wcsncat() — Append to wide-character string

Standards

Standards / Extensions	C or C++	Dependencies
ISO C Amendment XPG4 XPG4.2 C99 Single UNIX Specification, Version 3	both	

Format

```
#include <wchar.h>

wchar_t *wcsncat(wchar_t * __restrict__string1,
                 const wchar_t * __restrict__string2, size_t count);
```

General description

Appends up to *count* wide characters from *string2* to the end of *string1* and appends a NULL wide character to the result. The wcsncat() function operates on NULL-terminated wide-character strings. The string arguments to this function must contain a NULL wide character marking the end of the string.

The behavior of this wide-character function is affected by the LC_CTYPE category of the current locale. If you change the category, undefined results can occur.

Returned value

wcsncat() returns *string1*.

Example

CELEBW13

```
/* CELEBW13

   This example demonstrates the difference between &wcscat. and
```

```
      &wcsncat..
      &wcscat. appends the entire second string to the first
      whereas &wcsncat. appends only the specified number of
      characters in the second string to the first.

    */
   #include <stdio.h>
   #include <wchar.h>
   #include <string.h>

   #define SIZE 40

   int main(void)
   {
     wchar_t buffer1[SIZE] = L"computer";
     wchar_t * ptr;

     /* Call wcscat with buffer1 and " program" */

     ptr = wcscat( buffer1, L" program" );
     printf( "wcscat : buffer1 = \"%ls\"\n", buffer1 );

     /* Reset buffer1 to contain just the string "computer" again */

     memset( buffer1, L'\0', sizeof( buffer1 ));
     ptr = wcscpy( buffer1, L"computer" );

     /* Call wcsncat with buffer1 and " program" */
     ptr = wcsncat( buffer1, L" program", 3 );
     printf( "wcsncat: buffer1 = \"%ls\"\n", buffer1 );
   }
```

Output:

```
wcscat : buffer1 = "computer program"
wcsncat: buffer1 = "computer pr"
```

Related information

- “wchar.h” on page 85
- “wcstr.h” on page 86
- “strncat() — Concatenate strings” on page 1750
- “wcsncmp() — Compare wide-character strings”
- “wcsncpy() — Copy wide-character string” on page 2017

wcsncmp() — Compare wide-character strings

Standards

Standards / Extensions	C or C++	Dependencies
ISO C Amendment XPG4 XPG4.2 C99 Single UNIX Specification, Version 3	both	

Format

```
#include <wchar.h>

int wcsncmp(const wchar_t *string1, const wchar_t *string2, size_t count);
```

General description

Compares up to *count* wide characters in *string1* to *string2*. The wcsncmp() function operates on NULL-terminated wide-character strings. The string arguments to this function must contain a NULL wide character marking the end of the string.

The behavior of this wide-character function is affected by the LC_CTYPE category of the current locale. If you change the category, undefined results can occur.

Returned value

wcsncmp() returns a value indicating the relationship between the two strings, as follows:

Value	Meaning
< 0	string pointed to by *string1* less than the string pointed to by *string2*
= 0	string pointed to by *string1* identical to string pointed to by *string2*
> 0	string pointed to by *string1* greater than string pointed to by *string2*

Example

CELEBW14

```
/* CELEBW14

   This example demonstrates the difference between &wcscmp.
   and &wcsncmp..

 */
#include <stdio.h>
#include <wchar.h>

#define SIZE 10

int main(void)
{
  int  result;
  int  index = 3;
  wchar_t buffer1[SIZE] = L"abcdefg";
  wchar_t buffer2[SIZE] = L"abcfg";
  void print_result( int, wchar_t *, wchar_t * );

  result = wcscmp( buffer1, buffer2 );
  printf( "Comparison of each character\n" );
  printf( "  wcscmp: " );
  print_result( result, buffer1, buffer2 );

  result = wcsncmp( buffer1, buffer2, index);
  printf( "\nComparison of only the first %i characters\n", index );
  printf( "  wcsncmp: " );
  print_result( result, buffer1, buffer2 );
}

void print_result( int res, wchar_t * p_buffer1, wchar_t * p_buffer2 )
{
  if ( res == 0 )
    printf( "\"%ls\" is identical to \"%ls\"\n", p_buffer1, p_buffer2);
  else if ( res < 0 )
    printf( "\"%ls\" is less than \"%ls\"\n", p_buffer1, p_buffer2 );
  else
    printf( "\"%ls\" is greater than \"%ls\"\n", p_buffer1, p_buffer2 );
}
```

Output:

```
Comparison of each character
  wcscmp: "abcdefg" is less than "abcfg"

Comparison of only the first 3 characters
  wcsncmp: "abcdefg" is identical to "abcfg"
```

Related information

- "wchar.h" on page 85
- "wcstr.h" on page 86
- "strncmp() — Compare strings" on page 1751
- "wcscmp() — Compare wide-character strings" on page 2005
- "wcsncat() — Append to wide-character string" on page 2014
- "wcsncpy() — Copy wide-character string"

wcsncpy() — Copy wide-character string

Standards

Standards / Extensions	C or C++	Dependencies
ISO C Amendment XPG4 XPG4.2 C99 Single UNIX Specification, Version 3	both	

Format

```
#include <wchar.h>

wchar_t *wcsncpy(wchar_t * __restrict__string1,
                 const wchar_t * __restrict__string2, size_t count);
```

General description

Copies up to *count* wide characters from *string2* to *string1*. If *string2* is shorter than *count* characters, *string1* is padded out to *count* characters with NULL wide characters. The wcsncpy() function operates on NULL-terminated wide-character strings. The string arguments to this function must contain a NULL wide character marking the end of the string.

The behavior of this wide-character function is affected by the LC_CTYPE category of the current locale. If you change the category, undefined results can occur.

Returned value

wcsncpy() returns *string1*.

Example

CELEBW15

```
/* CELEBW15

   This example demonstrates the difference between &wcscpy. and &wcsncpy..

 */
#include <stdio.h>
#include <wchar.h>
```

```
#define SIZE    40

int main(void)
{
  wchar_t source[ SIZE ] = L"123456789";
  wchar_t source1[ SIZE ] = L"123456789";
  wchar_t destination[ SIZE ] = L"abcdefg";
  wchar_t destination1[ SIZE ] = L"abcdefg";
  wchar_t * return_string;
  int    index = 5;

  /* This is how wcscpy works */
  printf( "destination is originally = '%ls'\n", destination );
  return_string = wcscpy( destination, source );
  printf( "After wcscpy, destination becomes '%ls'\n\n", destination );

  /* This is how wcsncpy works */
  printf( "destination1 is originally = '%ls'\n", destination1 );
  return_string = wcsncpy( destination1, source1, index );
  printf( "After wcsncpy, destination1 becomes '%ls'\n", destination1 );
}
```

Output:

```
destination is originally = 'abcdefg'
After wcscpy, destination becomes '123456789'

destination1 is originally = 'abcdefg'
After wcsncpy, destination1 becomes '12345fg'
```

Related information

- “wchar.h” on page 85
- “wcstr.h” on page 86
- “strncpy() — Copy string” on page 1753
- “wcscpy() — Copy wide-character string” on page 2008
- “wcsncat() — Append to wide-character string” on page 2014
- “wcsncmp() — Compare wide-character strings” on page 2015

wcspbrk() — Locate first wide characters in string

Standards

Standards / Extensions	C or C++	Dependencies
ISO C Amendment XPG4 XPG4.2 C99 Single UNIX Specification, Version 3	both	

Format

```
#include <wchar.h>

wchar_t *wcspbrk(const wchar_t *string1, const wchar_t *string2);
```

General description

Locates the first occurrence in the string pointed to by *string1* of any character from the string pointed to by *string2*.

The behavior of this wide-character function is affected by the LC_CTYPE category of the current locale. If you change the category, undefined results can occur.

Returned value

If successful, wcspbrk() returns a pointer to the character.

If no `wchar_t` from *string2* occurs in *string1*, wcspbrk() returns NULL.

Example

CELEBW16

```
/* CELEBW16

   This example returns a pointer to the first occurrence in the
   array string of either a or b, using &wcspbrk..

 */
#include <stdio.h>
#include <wchar.h>

int main(void)
{
  wchar_t * result;
  wchar_t * string = L"The Blue Danube";
  wchar_t *chars = L"ab";

  result = wcspbrk( string, chars);
   printf("The first occurrence of any of the characters \"%ls\" in "
          "\"%ls\" is \"%ls\"\n", chars, string, result);
}
```

Output:

```
The first occurrence of any of the characters "ab" in "The Blue Danube" is "anube"
```

Related information

- “wchar.h” on page 85
- “wcstr.h” on page 86
- “strpbrk() — Find characters in string” on page 1754
- “wcschr() — Search for wide-character substring” on page 2004
- “wcscmp() — Compare wide-character strings” on page 2005
- “wcscspn() — Find offset of first wide-character match” on page 2009
- “wcsncmp() — Compare wide-character strings” on page 2015
- “wcsrchr() — Locate last wide character in string”
- “wcswcs() — Locate wide-character substring in wide-character string” on page 2048

wcsrchr() — Locate last wide character in string

Standards

Standards / Extensions	C or C++	Dependencies
ISO C Amendment XPG4 XPG4.2 C99 Single UNIX Specification, Version 3	both	

Format

```
#include <wchar.h>

wchar_t *wcsrchr(const wchar_t *string, wchar_t character);
```

General description

Locates the last occurrence of *character* in the string pointed to by *string*. The terminating NULL wide character is considered to be part of the string.

The behavior of this wide-character function is affected by the LC_CTYPE category of the current locale. If you change the category, undefined results can occur.

Returned value

If successful, wcsrchr() returns a pointer to the character.

If *character* does not occur in the string, wcsrchr() returns NULL.

Example

CELEBW17

```
/* CELEBW17

   This example compares the use of wcschr() and wcsrchr().
   It searches the string for the first and last occurrence of p in the
   wide character string.

 */
#include <stdio.h>
#include <wchar.h>

#define SIZE 40

int main(void)
{
  wchar_t buf[SIZE] = L"computer program";
  wchar_t * ptr;
  int    ch = 'p';

  /* This illustrates wcschr */
  ptr = wcschr( buf, ch );
  printf( "The first occurrence of %c in '%ls' is '%ls'\n", ch, buf, ptr );

  /* This illustrates wscrchr */
  ptr = wcsrchr( buf, ch );
  printf( "The last occurrence of %c in '%ls' is '%ls'\n", ch, buf, ptr );
}
```

Output:

```
The first occurrence of p in 'computer program' is 'puter program'
The last occurrence of p in 'computer program' is 'program'
```

Related information

- “wchar.h” on page 85
- “wcstr.h” on page 86
- “strrchr() — Find last occurrence of character in string” on page 1759
- “wcscmp() — Compare wide-character strings” on page 2005
- “wcscspn() — Find offset of first wide-character match” on page 2009
- “wcsncmp() — Compare wide-character strings” on page 2015

- "wcspbrk() — Locate first wide characters in string" on page 2018
- "wcswcs() — Locate wide-character substring in wide-character string" on page 2048

wcsrtombs() — Convert wide-character string to multibyte string

Standards

Standards / Extensions	C or C++	Dependencies
ISO C Amendment C99 Single UNIX Specification, Version 3	both	

Format

Non-XPG4:

```
#include <wchar.h>

size_t wcsrtombs(char * __restrict__dst,
                 const wchar_t ** __restrict__src, size_t len,
                 mbstate_t * __restrict__ps);
```

XPG4:

```
#define _XOPEN_SOURCE
#define _MSE_PROTOS
#include <wchar.h>

size_t wcsrtombs(char *dst,
                 const wchar_t **src, size_t len, mbstate_t *ps);
```

General description

Converts a sequence of wide characters from the array indirectly pointed to by *src* into a sequence of corresponding multibyte characters that begin in the shift state described by *ps*, which, if *dst* is not a NULL pointer, are then stored into the array pointed to by *dst*. Conversion continues up to and including the terminating NULL wide character; the terminating NULL wide character (byte) shall be stored.

Conversion shall stop earlier in two cases:

- When a code is reached that does not correspond to a valid multibyte character.
- If *dst* is not a NULL pointer, conversion stops when the next multibyte element would exceed the limit of *len* total bytes to be stored into the array pointed to by *dst*.

Each conversion takes places as if by a call to the wcrtomb() function.

If *dst* is not NULL a pointer, the object pointed to by *src* shall be assigned either a NULL pointer (if conversion stopped due to reaching a terminating NULL wide character) or the address of the code just past the last wide character converted. If conversion stopped due to reaching a terminating NULL wide character, the resulting state described shall be the initial conversion state.

wcsrtombs() is a "restartable" version of wcstombs(). That is, shift state information is passed as on of the arguments, and gets updated on exit. With wcsrtombs(), you may switch from one multibyte string to another, provided that you have kept the shift state information.

The behavior of this wide-character function is affected by the LC_CTYPE category of the current locale. If you change the category, undefined results can occur.

Special behavior for XPG4: If you define any feature test macro specifying XPG4 behavior before the statement in your program source file to include the wchar header, then you must also define the _MSE_PROTOS feature test macro to make the declaration of the wcsrtombs() function in the wchar header available when you compile your program. Please see Table 2 on page 4 for a list of XPG4 and other feature test macros.

Returned value

If successful, wcsrtombs() returns the number of bytes in the resulting multibyte character sequence, which is the same as the number of array elements modified when *dst* is not a NULL pointer.

If the string contains an invalid wide character, an encoding error occurs. wcsrtombs() returns (size_t)-1 and stores the value of the macro EILSEQ in errno, but the conversion state shall be unchanged.

Example

CELEBW18

```
/* CELEBW18 */
#include <stdio.h>
#include <string.h>
#include <wchar.h>

#define SIZE 20

int main(void)
{
   char     dest[SIZE];
   wchar_t *wcs = L"string";
   const wchar_t *ptr;
   size_t   count = SIZE;
   size_t   length;

   ptr = (wchar_t *) wcs;
   length = wcsrtombs(dest, &ptr, count, NULL);
   printf("%d characters were converted.\n", length);
   printf("The converted string is \"%s\"\n\n", dest);

   /* Reset the destination buffer */
   memset(dest, '\0', sizeof(dest));

   /* Now convert only 3 characters */
   ptr = (wchar_t *) wcs;
   length = wcsrtombs(dest, &ptr, 3, NULL);
   printf("%d characters were converted.\n", length);
   printf("The converted string is \"%s\"\n\n", dest);
}
```

Output:

```
6 characters were converted.
The converted string is "string"

3 characters were converted.
The converted string is "str"
```

Related information

- "wchar.h" on page 85
- "mblen() — Calculate length of multibyte character" on page 1040
- "mbrlen() — Calculate length of multibyte character" on page 1043
- "mbrtowc() — Convert a multibyte character to a wide character" on page 1049
- "mbsrtowcs() — Convert a multibyte string to a wide-character string" on page 1052
- "wcrtomb() — Convert a wide character to a multibyte character" on page 2001
- "wcstombs() — Convert wide-character string to multibyte character string" on page 2041

wcsspn() — Search for wide characters in a string

Standards

Standards / Extensions	C or C++	Dependencies
ISO C Amendment XPG4 XPG4.2 C99 Single UNIX Specification, Version 3	both	

Format

```
#include <wchar.h>

size_t wcsspn(const wchar_t *string1, const wchar_t *string2);
```

General description

Computes the number of wide characters in the initial segment of the string pointed to by *string1*, which consists entirely of wide characters from the string pointed to by *string2*.

The behavior of this wide-character function is affected by the LC_CTYPE category of the current locale. If you change the category, undefined results can occur.

Returned value

wcsspn() returns the number of wide characters in the segment.

Example

CELEBW19

```
/* CELEBW19

   This example finds the first occurrence in the array string
   of a character that is neither an a, b, nor c. Because the
   string in this example is cabbage, &wcsspn. returns 5, the
   index of the segment of cabbage before a character that is
   not an a, b, or c.

 */
#include <stdio.h>
#include <wchar.h>

int main(void)
{
  wchar_t * string = L"cabbage";
```

```
   wchar_t * source = L"abc";
   int index;

   index = wcsspn( string, L"abc" );
   printf( "The first %d characters of \"%ls\" are found in \"%ls\"\n",
               index, string, source );
}
```

Output:

```
The first 5 characters of "cabbage" are found in "abc"
```

Related information

- "wchar.h" on page 85
- "wcstr.h" on page 86
- "strspn() — Search string" on page 1761
- "wcscat() — Append to wide-character string" on page 2003
- "wcschr() — Search for wide-character substring" on page 2004
- "wcscmp() — Compare wide-character strings" on page 2005
- "wcscspn() — Find offset of first wide-character match" on page 2009
- "wcsncmp() — Compare wide-character strings" on page 2015

wcsstr() — Locate a wide character sequence

Standards

Standards / Extensions	C or C++	Dependencies
ISO C Amendment C99 Single UNIX Specification, Version 3	both	

Format

Non-XPG4:

```
#include <wchar.h>

wchar_t *wcsstr(const wchar_t *__restrict__ wcs1,
                const wchar_t *__restrict__ wcs2);
```

XPG4:

```
#define _XOPEN_SOURCE
#define _MSE_PROTOS
#include <wchar.h>

wchar_t *wcsstr(const wchar_t *__restrict__ wcs1,
                const wchar_t *__restrict__ wcs2);
```

General description

Locates the first occurrence in the wide-character string pointed to by *wcs1* of the sequence of wide characters (excluding the terminating NULL character) in the wide-character string pointed to by *wcs2*.

The behavior of this wide-character function is affected by the LC_CTYPE category of the current locale. If you change the category, undefined results can occur.

Special behavior for XPG4: If you define any feature test macro specifying XPG4 behavior before the statement in your program source file to include the wchar header, then you must also define the _MSE_PROTOS feature test macro to make the declaration of the wcsstr() function in the wchar header available when you compile your program. Please see Table 2 on page 4 for a list of XPG4 and other feature test macros.

Returned value

If successful, wcsstr() returns a pointer to the located wide string. If *wcs2* points to a wide-character string with zero length, wcsstr() returns *wcs1*.

If the wide-character string is not found, wcsstr() returns NULL.

Example

CELEBW20

```
/* CELEBW20 */
#include <stdio.h>
#include <wchar.h>

int main(void)
{
   wchar_t *wcs1 = L"needle in a haystack";
   wchar_t *wcs2 = L"hay";
   wchar_t *result;

   result = wcsstr(wcs1, wcs2);

   /* result = a pointer to "hatstack" */

   printf("result: `%S`\n", result);
}
```

Related information

- “wchar.h” on page 85
- “strstr() — Locate substring” on page 1762

wcstod() — Convert wide-character string to a double floating-point

Standards

Standards / Extensions	C or C++	Dependencies
ISO C Amendment XPG4 XPG4.2 C99 Single UNIX Specification, Version 3	both	

Format

```
#include <wchar.h>

double wcstod(const wchar_t * __restrict__nptr, wchar_t ** __restrict__endptr);
```

General description

The wcstod() function converts a wchar_t * type floating-point number input string to a double value.

See the "*fscanf* Family of Formatted Input Functions" on fscanf(), scanf(), sscanf() — Read and format data for a description of special infinity and NaN sequences recognized by z/OS formatted input functions, including wcstod() in IEEE Binary Floating-Point mode.

Converts the initial portion of the wide-character string pointed to by *nptr* to `double` representation. First it decomposes the input string into three parts:

1. An initial, possibly empty, sequence of white space characters (as specified by the iswspace() function)
2. A subject sequence interpreted as a floating-point constant or representing infinity or a NAN.
3. A final string of one or more unrecognized characters, including the terminating NULL character of the input string.

Then it attempts to convert the subject sequence to a floating-point number, and returns the result.

The expected form of the subject sequence is an optional plus or minus sign, then one of the following:

- A non-empty sequence of decimal digits optionally containing a radix character, then an optional exponent part. Where radix character is the character that separates the integer part of a number from the fractional part.
- A 0x or 0X, then a non-empty sequence of hexadecimal digits optionally containing a radix character, then an optional binary exponent part. Where radix character is the character that separates the integer part of a number from the fractional part.
- One of INF or INFINITY, ignoring case.
- One of NANQ or NANQ(n-char-sequence), ignoring case.
- One of NANS or NANS(n-char-sequence), ignoring case.
- One of NAN or NAN(n-char-sequence), ignoring case.

The subject sequence is defined as the longest initial subsequence of the input wide-character string, starting with the first non-white space wide character, that is of the expected form. The subject sequence contains no wide characters if the input wide-character string is empty or consists entirely of white space wide characters, or if the first non-white space wide character is other than a sign, a digit, or a decimal-point wide character.

If the subject sequence has the expected form, the sequence of wide characters starting with the first digit or the decimal-point wide character (whichever occurs first) is interpreted as a floating constant according to the rules of ISO/IEC 9899: subclause 6.1.3.1, except the decimal-point wide character is used in place of a period, and if neither an exponent part nor a decimal-point wide character appears, a decimal-point is assumed to follow the last digit in the wide-character string. If the subject sequence begins with a minus sign, the value resulting from the conversion is negated. A pointer to the final wide-character string is stored in the object pointed to by *endptr*, provided that *endptr* is not a NULL pointer.

In a locale other than the C locale, additional implementation-defined subject sequence forms may be accepted.

If the subject sequence is empty or does not have the expected form, no conversion is performed; the value of *nptr* is stored in the object pointed to by *endptr*, provided that *endptr* is not a NULL pointer.

The behavior of this wide-character function is affected by the LC_CTYPE category of the current locale. If you change the category, undefined results can occur.

Returned value

If successful, wcstod() returns the converted value, if any.

If no conversion could be performed, wcstod() returns 0.

The double value is hexadecimal floating-point or IEEE Binary Floating-Point format depending on the floating-point mode of the thread invoking wcstod(). The wcstod() function uses __isBFP() to determine the floating-point format (hexadecimal floating-point or IEEE Binary Floating-Point) of the invoking thread.

If the correct value is outside the range of representable values, wcstod() returns ±HUGE_VAL—according to the sign of the value—and the value of the macro ERANGE is stored in errno.

Example

CELEBW21

```
/* CELEBW21 */
#include <stdio.h>
#include <wchar.h>

int main(void)
{
   wchar_t *wcs;
   wchar_t *stopwcs;
   double   d;

   wcs = L"3.1415926This stopped it";
   d = wcstod(wcs, &stopwcs);
   printf("wcs = `%ls`\n", wcs);
   printf("   wcstod = %f\n", d);
   printf("   Stopped scan at `%ls`\n", stopwcs);
}
```

Related information

- "wchar.h" on page 85
- "__isBFP() — Determine application floating-point format" on page 904
- "wcstof() — Convert a wide-character string to float" on page 2030
- "wcstold() — Convert a wide-character string to long double" on page 2037

wcstod32(), wcstod64(), wcstod128() — Convert wide-character string to decimal floating point

Standards

Standards / Extensions	C or C++	Dependencies
C/C++ DFP	both	z/OS V1.8

Format

```
#define __STDC_WANT_DEC_FP__
#include <wchar.h>

_Decimal32 wcstod32(const wchar_t * __restrict__ nptr,
                          wchar_t ** __restrict__ endptr);

_Decimal64 wcstod64(const wchar_t * __restrict__ nptr,
                          wchar_t ** __restrict__ endptr);

_Decimal128 wcstod128(const wchar_t * __restrict__ nptr,
                            wchar_t ** __restrict__ endptr);
```

General description

The wcstod32() wcstod64(), and wcstod128() functions convert the initial portion of the wide-character string pointed to by *nptr* to _Decimal32, _Decimal64, and _Decimal128 representation, respectively.

First, they decompose the input wide-character string into three parts:

1. An initial, possibly empty, sequence of white-space wide characters (as specified by the iswspace() function).
2. A subject sequence resembling a floating-point constant or representing an infinity or NaN.
3. A final wide-character string of one or more unrecognized wide characters, including the terminating null wide character of the input wide-character string.

Then, they attempt to convert the subject sequence to a floating-point number, and return the result.

The expected form of the subject sequence is an optional plus or minus sign, then one of the following:

- a nonempty sequence of decimal digits optionally containing a decimal-point wide character, then an optional exponent part
- INF or INFINITY, ignoring case
- NAN, NAN(n-char-sequence), NANQ, NANQ(n-char-sequence), NANS, or NANS(n-char-sequence), ignoring case in the NAN, NANQ, or NANS part, where n-char-sequence is one or more decimal numeric digits

Note: If the input string is not one of these forms (for example "INFINITE"), the output results are undefined.

The subject sequence is defined as the longest initial subsequence of the input wide-character string, starting with the first non-white-space wide character, that is of the expected form. The subject sequence contains no wide characters if the input wide character string is not of the expected form.

If the subject sequence has the expected form for a floating-point number, the sequence of wide characters starting with the first digit or the decimal-point wide character (whichever occurs first) is interpreted as a floating constant. If neither an exponent nor a decimal-point character appears in a decimal floating point number, an exponent with value zero is assumed to follow the last digit in the string. If the subject sequence begins with a minus sign, the sequence is interpreted as negated. A wide character sequence INF or INFINITY is interpreted as an infinity. A wide character sequence NAN, NAN(),or NAN(n-char-sequence) is interpreted as a quiet NAN. A wide character sequence of NANS, NANS(),or NANS(n-char-sequence), is interpreted as a signalling NaN.

A pointer to the final wide-character string is stored in the object pointed to by *endptr*, provided that *endptr* is not a null pointer.

The converted value keeps the same precision as the input if possible, and the value may be denormalized. Otherwise, rounding may occur. Rounding happens after any negation.

In other than the "C" locale, additional locale-specific subject sequence forms are accepted.

If the subject sequence is empty or does not have the expected form, no conversion is performed; the value of *nptr* is stored in the object pointed to by *endptr*, provided that *endptr* is not a null pointer.

Argument	**Description**
nptr	Input pointer to start of the wide-character string to be converted
endptr	NULL, or a pointer to a output pointer field that is filled in with the address of the first wide character in the input wide-character string that is not used in the conversion.

Note: To use IEEE decimal floating-point, the hardware must have the Decimal Floating-Point Facility installed.

Returned value

The functions return the converted value, if any. If no conversion could be performed, the value +0.E0DF, +0.E0DD, or +0.E0DL is returned. If the correct value is outside the range of representable values, plus or minus HUGE_VAL_D32, HUGE_VAL_D64, or HUGE_VAL_D128 is returned (according to the return type and sign of the value), and errno is set to ERANGE. If the result underflows, the functions return a value whose magnitude is no greater than the smallest normalized positive number in the return type. No signal is raised at the point of returning a signaling NaN.

errno	**Description**
ERANGE	The input wide-character string represents a value too large to fit in the output Decimal Floating Point type.

Example

See “wcstod() — Convert wide-character string to a double floating-point” on page 2025 for an example.

Related information

- “wchar.h” on page 85
- “wcstod() — Convert wide-character string to a double floating-point” on page 2025
- “strtod32(), strtod64(), strtod128() — Convert character string to decimal floating point” on page 1767

wcstof() — Convert a wide-character string to float

Standards

Standards / Extensions	C or C++	Dependencies
C99 Single UNIX Specification, Version 3 C++ TR1 C99	both	z/OS V1R7

Format

```
#define _ISOC99_SOURCE
#include <wchar.h>

float wcstof(const wchar_t *__restrict__ nptr, wchar_t **__restrict__ endptr);
```

General description

wcstof() converts a wchar_t * floating-point number input string to a float value. The parameter *nptr* points to a sequence of wide-characters that can be interpreted as a numerical float value.

It decomposes the input wide-character string into three parts:

1. An initial, possibly empty, sequence of white-space character codes (as specified by iswspace()).
2. A subject sequence resembling a floating-point constant, infinity, or NaN.
3. A final wide-character string of one or more unrecognized wide-characters codes, including the terminating NULL wide-character of the input wide-character string.

The function then attempts to convert the subject string into the floating-point number, and returns the result.

The expected form of the subject sequence is an optional plus or minus sign, then one of the following:

- A non-empty sequence of decimal digits optionally containing a radix character, then an optional exponent. A radix character is the character that separates the integer part of a number from the fractional part.
- A 0x or 0X, a non-empty sequence of hexadecimal digits optionally containing a radix character, then a base 2 decimal exponent part with a p or P as prefix, a plus or minus sign, then a sequence of at least one decimal digit. (Example [-]0xh.hhhhp+/-d). A radix character is the character that separates the integer part of a number from the fractional part.
- One INF or INFINITY, ignoring case.
- One NANQ or NANQ(n-char-sequence), ignoring case.
- One NANS or NANS(n-char-sequence), ignoring case.
- One NAN or NAN(n-char-sequence), ignoring case.

See the "scanf() Family of Formatted Input Functions" for a description of special infinity and NaN sequences recognized by z/OS formatted input functions in IEEE Binary Floating-Point mode. A pointer to the final wide-character string is stored in the object pointed to by *endptr*, provided that *endptr* is not a NULL pointer.

Returned value

If successful, wcstof() returns the converted value, if any. If no conversion could be performed, it returns 0.

The float value is a hexadecimal floating-point or IEEE Binary Floating-Point format depending on the floating-point mode of the thread invoking wcstof(). This function uses __isBFP() to determine the floating-point mode of the invoking thread.

If the correct value is outside the range of representable values, then +/-HUGE_VALF, according to the sign of the value, is returned and the value of the ERANGE macro is stored in errno. If the correct value would cause an underflow, 0 is returned and the value of the ERANGE macro is stored in errno.

Related information

- "wchar.h" on page 85
- "__isBFP() — Determine application floating-point format" on page 904
- "wcstod() — Convert wide-character string to a double floating-point" on page 2025
- "wcstold() — Convert a wide-character string to long double" on page 2037
- "wcstol() — Convert a wide-character string to a long integer" on page 2035
- "wcstoul() — Convert a wide-character string to an unsigned long integer" on page 2043
- "wcstoimax() — Convert a wide-character string to a intmax_t"
- "wcstoumax() — Convert a wide-character string to a intmax_t" on page 2047

wcstoimax() — Convert a wide-character string to a intmax_t

Standards

Standards / Extensions	C or C++	Dependencies
C99 Single UNIX Specification, Version 3 C++ TR1 C99	both	z/OS V1R7

Format

```
#define _ISOC99_SOURCE
#include <inttypes.h>

intmax_t wcstoimax(const wchar_t * __restrict__ nptr,
                   wchar_t ** __restrict__ endptr, int base);
```

Compile requirement: Function wcstoimax() requires long long to be available.

General description

The wcstoimax() function converts the wide-character string nptr to an intmax_t integer type. Valid input values for base are 0 and in the range 2-36. The wcstoimax() function is equivalent to wcstol() and wcstoll(). The only difference being that the return value is of type intmax_t. See wcstoll() for more information.

Returned value

If successful, wcstoimax() returns the converted value, if any.

If unsuccessful, wcstoimax() returns 0 if no conversion could be performed. If the correct value is outside the range of representable values, wcstoimax() returns INTMAX_MAX or INTMAX_MIN, according to the sign of the value. If the value of base is not supported, wcstoimax() returns 0.

If unsuccessful wcstoimax() sets errno to one of the following values:

Error Code
Description

EINVAL
The value of *base* is not supported.

ERANGE
The conversion caused an overflow.

Example

```
#define _ISOC99_SOURCE
#include <inttypes.h>
#include <stdio.h>

 int main(void)
{
   wchar_t *nptr;
   wchar_t *endptr;
   intmax_t  j;
   int base = 10;
   nptr = L"10110134932";
   printf("nptr = `%ls`\n", nptr);
      j = wcstoimax(nptr, &endptr, base);
      printf("wcstoimax = %jd\n", j);
      printf("Stopped scan at `%ls`\n\n", endptr);
}



Output

nptr = `10110134932`
wcstoimax = 10110134932
Stopped scan at ``
```

Related information

- “inttypes.h” on page 33
- “stdint.h” on page 65
- “imaxdiv() — Quotient and remainder for intmax_t” on page 839
- “strtoimax() — Convert character string to intmax_t integer type” on page 1770
- “strtoumax() — Convert character string to uintmax_t integer type” on page 1784
- “wcstoumax() — Convert a wide-character string to a intmax_t” on page 2047

wcstok() — Break a wide-character string into tokens

Standards

Standards / Extensions	C or C++	Dependencies
ISO C Amendment XPG4 XPG4.2 C99 Single UNIX Specification, Version 3	both	

Format

Non-XPG4:

```
#include <wchar.h>

wchar_t *wcstok(wchar_t * __restrict__wcs1,
                const wchar_t * __restrict__wcs2, wchar_t ** __restrict__ptr);
```

XPG4:

```
#define _XOPEN_SOURCE
#include <wchar.h>

wchar_t *wcstok(wchar_t *wcs1, const wchar_t *wcs2);
```

XPG4 and MSE:

```
#define _XOPEN_SOURCE
#define _MSE_PROTOS
#include <wchar.h>

wchar_t *wcstok(wchar_t *wcs1,
                const wchar_t *wcs2, wchar_t **ptr);
```

General description

A sequence of calls to the wcstok() function breaks the wide string pointed to by *wcs1* into a sequence of tokens, each of which is delimited by a wide character from the wide string pointed to by *wcs2*. The third argument points to a caller-provided wide-character pointer into which the wcstok() function stores information necessary for it to continue scanning the same string.

The first call in the sequence, *wcs1* shall point to a wide-character string, while in subsequent calls for the same wide string, *wcs1* shall be a NULL pointer. If *wcs1* is a NULL pointer, the value pointed to by ptr shall match that set by the previous call for the same wide-character string; otherwise its value is ignored. The separator wide-character string pointed to by *wcs2* may be different from call to call.

The first call in the sequence, searches the wide-character string pointed to by *wcs1* for the first wide character that is not contained in the current separator wide-character string pointed to by *wcs2*. If no such wide character is found, then there are no tokens in the wide-character string pointed to by *wcs1* and wcstok() returns a NULL pointer. If such a wide character is found, it is the start of the first token.

wcstok() then searches from there for a wide character that is contained in the current separator wide string. If no such wide character is found, the current token extends to the end of the wide-character string pointed to by *wcs1*, and subsequent searches for a token will return a NULL pointer. If such a wide character is found, it is overwritten by a NULL character, which terminates the current token.

In all cases, the wcstok() function stores sufficient information in the pointer ptr so that subsequent calls, with a NULL pointer as the value of the first argument and the unmodified pointer value as the third, will start searching just past the end of the previously returned token (if any).

The behavior of this wide-character function is affected by the LC_CTYPE category of the current locale. If you change the category, undefined results can occur.

Special behavior for XPG4: If you define any feature test macro specifying XPG4 behavior before the statement in your program source file to include the wchar header, then the compiler assumes that your program is using the XPG4 variety of the wcstok() function, unless you also define the _MSE_PROTOS feature test macro. Please see Table 2 on page 4 for a list of XPG4 and other feature test macros.

The prototype for the XPG4 variety of the wcstok() function is:

```
wchar_t *wcstok(wchar_t *wcs1, const wchar_t *wcs2);
```

This variety of the wcstok() function is missing a third parameter to specify the address of restart information in your program storage. Instead, C/370 provides comparable restart information in runtime library storage. Please note that this library storage is provided on a per thread basis making the XPG4 wcstok() function thread-specific for a threaded application.

Returned value

If successful, wcstok() returns a pointer to the first wide character of a token.

If there is no token, wcstok() returns a NULL pointer.

Example

CELEBW22

```
/* CELEBW22 */
#include <wchar.h>
int main(void)
{

    static wchar_t str1[] = L"?a??b,,,#c";
    static wchar_t str2[] = L"\t \t";
    wchar_t *t, *ptr1, *ptr2;

    t = wcstok(str1, L"?", &ptr1);     /* t points to the token L"a"    */
    t = wcstok(NULL, L",", &ptr1);     /* t points to the token L"??b"  */
    t = wcstok(str2, L" \t,", &ptr2);  /* t is a null pointer           */
    t = wcstok(NULL, L"#,", &ptr1);    /* t points to the token L"c"    */
    t = wcstok(NULL, L"?", &ptr1);     /* t is a null pointer           */

}
```

Related information

- “wchar.h” on page 85
- “strtok() — Tokenize string” on page 1772
- “strtok_r() — Split string into tokens” on page 1774

wcstol() — Convert a wide-character string to a long integer

Standards

Standards / Extensions	C or C++	Dependencies
ISO C Amendment XPG4 XPG4.2 C99 Single UNIX Specification, Version 3	both	

Format

```
#include <wchar.h>

long int wcstol(const wchar_t * __restrict__nptr,
                wchar_t ** __restrict__endptr, int base);
```

General description

Converts the initial portion of the wide-character string pointed to by *nptr* to long int representation. First it decomposes the input wide-character string into three parts:

1. An initial, possibly empty, sequence of white space wide characters (as specified by the iswspace() function).
2. A subject sequence resembling an integer determined by the value of base.
3. A final wide-character string of one or more unrecognized wide characters, including the terminating NULL character of the input wide-character string.

Then it attempts to convert the subject sequence to an integer, and returns the result.

If the value of base is zero, the expected form of the subject sequence is that of an integer constant as described in ISO/IEC 9899: subclause 6.1.3.2, optionally preceded by a plus or minus sign, but not including an integer suffix. If the value of base is between 2 and 36, the expected form of the subject sequence is a sequence of letters and digits from the portable character set representing an integer with the radix specified by base, optionally preceded by a plus or minus sign, but not including an integer suffix. The letters from a (or A) through z (or Z) are ascribed the values 10 to 35; only letters whose ascribed values are less than that of base are permitted. If the value of base is 16, the characters 0x or 0X may optionally precede the sequence of letters and digits, following the sign if present.

The subject sequence is defined as the longest initial subsequence of the input wide-character string, starting with the first non-white space wide character, that is of the expected form. The subject sequence contains no wide characters if the input wide-character string is empty or consists entirely of white space, or if the first non-white space wide character is other than a sign or a permissible letter or digit.

If the subject sequence has the expected form and the value of base is zero, the sequence of wide characters starting with the first digit is interpreted as an integer constant according to the rules of ISO/IEC 9899: subclause 6.1.3.2. If the subject sequence has the expected form and the value of base is between 2 and 36, it is used as the base for conversion, ascribing to each letter its value as given above. If the subject sequence begins with a minus sign, the value resulting from the conversion is negated. A pointer to the final wide-character string is stored in the object pointed to by *endptr*, provided that *endptr* is not a NULL pointer.

In a locale other than the C or POSIX locale, additional implementation-defined subject sequence forms may be accepted.

If the subject sequence is empty or does not have the expected form, no conversion is performed; the value of *nptr* is stored in the object pointed to by *endptr*, provided that *endptr* is not a NULL pointer.

The behavior of this wide-character function is affected by the LC_CTYPE category of the current locale. If you change the category, undefined results can occur.

Since 0, {LONG_MIN} and {LONG_MAX} are returned on error and are also valid returns on success, an application wishing to check for error situations should set errno to 0, then call wcstol(), then check errno.

Returned value

If successful, wcstol() returns the converted value, if any.

If unsuccessful, wcstol() returns 0 if no conversion could be performed. If the correct value is outside the range of representable values, wcstol() returns LONG_MAX or LONG_MIN, according to the sign of the value. If the value of base is not supported, wcstol() returns 0.

If unsuccessful wcstol() sets errno to one of the following values:

Error Code
    Description

EINVAL
    The value of *base* is not supported.

ERANGE
    The conversion caused an overflow.

Example

CELEBW23

```
/* CELEBW23 */
#include <stdio.h>
#include <wchar.h>

int main(void)
{
   wchar_t *wcs;
   wchar_t *stopwcs;
   long     l;
   int      base;

   wcs = L"10110134932";
   printf("wcs = `%ls`\n", wcs);
   for (base=2; base<=8; base*=2) {
      l = wcstol(wcs, &stopwcs, base);
      printf("   wcstol = %ld\n", l);
      printf("   Stopped scan at `%ls`\n\n", stopwcs);
   }
}
```

Related information

- “wchar.h” on page 85
- “strtol() — Convert character string to long” on page 1775
- “wcstoimax() — Convert a wide-character string to a intmax_t” on page 2031
- “wcstoumax() — Convert a wide-character string to a intmax_t” on page 2047

wcstold() — Convert a wide-character string to long double

Standards

Standards / Extensions	C or C++	Dependencies
C99 Single UNIX Specification, Version 3 C++ TR1 C99	both	z/OS V1R7

Format

```
#define _ISOC99_SOURCE
#include <wchar.h>

long double wcstold(const wchar_t *__restrict__ nptr,
                    wchar_t **__restrict__ endptr);
```

General description

wcstold() converts a wchar_t * floating-point number input string to a long double value. The parameter *nptr* points to a sequence of wide-characters that can be interpreted as a numerical long double value.

It decomposes the input wide-character string into three parts:

1. An initial, possibly empty, sequence of white-space wide-character codes (as specified by iswspace()).
2. A subject sequence resembling a floating-point constant, infinity, or NaN.
3. A final wide-character string of one or more unrecognized wide-characters codes, including the terminating NULL wide-character of the input wide-character string.

The function then attempts to convert the subject string into the floating-point number, and returns the result.

The expected form of the subject sequence is an optional plus or minus sign, then one of the following:

- A non-empty sequence of decimal digits optionally containing a radix character, then an optional exponent. A radix character is the character that separates the integer part of a number from the fractional.
- A 0x or 0X, a non-empty sequence of hexadecimal digits optionally containing a radix character, then a base 2 decimal exponent part with a p or P as prefix, a plus or minus sign, then a sequence of at least one decimal digit. (Example [-]0xh.hhhhp+/-d). A radix character is the character that separates the integer part of a number from the fractional part.
- One INF or INFINITY, ignoring case.
- One NANQ or NANQ(n-char-sequence), ignoring case.
- One NANS or NANS(n-char-sequence), ignoring case.
- One NAN or NAN(n-char-sequence), ignoring case.

See the "scanf() Family of Formatted Input Functions" for a description of special infinity and NaN sequences recognized by z/OS formatted input functions in IEEE Binary Floating-Point mode. The pointer to the final wide-character string is stored in the object pointed to by *endptr*, provided that *endptr* is not a NULL pointer.

Returned value

If successful, wcstold() returns the converted value, if any. If no conversion could be performed, it returns 0.

The long double value is a hexadecimal floating-point or IEEE Binary Floating-Point format depending on the floating-point mode of the invoking thread. This function uses __isBFP() to determine the floating-point mode of the invoking thread.

If the correct value is outside the range of representable values, then +/-HUGE_VALL, according to the sign of the value, is returned and the value of the ERANGE macro is stored in errno.

Related information

- "wchar.h" on page 85
- "__isBFP() — Determine application floating-point format" on page 904
- "wcstof() — Convert a wide-character string to float" on page 2030
- "wcstod() — Convert wide-character string to a double floating-point" on page 2025
- "wcstol() — Convert a wide-character string to a long integer" on page 2035
- "wcstoul() — Convert a wide-character string to an unsigned long integer" on page 2043

wcstoll() — Convert a wide-character string to a long long integer

Standards

Standards / Extensions	C or C++	Dependencies
z/OS UNIX System Services C99 Single UNIX Specification, Version 3 C++ TR1 C99	both	OS/390 V2R10

Format

```
#include <wchar.h>

long long wcstoll(const wchar_t * __restrict__nptr,
                  wchar_t ** __restrict__endptr, int base);
```

Compile requirement: Use of this function requires the long long data type. See *z/OS C/C++ Language Reference* for information on how to make long long available.

General description

Converts the initial portion of the wide-character string pointed to by *nptr* to long long representation. First it decomposes the input wide-character string into three parts:

1. An initial, possibly empty, sequence of white space wide characters (as specified by the iswspace() function).

2. A subject sequence resembling an integer determined by the value of *base*.
3. A final wide-character string of one or more unrecognized wide characters, including the terminating NULL character of the input wide-character string.

Then it attempts to convert the subject sequence to a long long integer, and returns the result.

If the value of *base* is zero, the expected form of the subject sequence is that of an integer constant as described in ISO/IEC 9899: subclause 6.1.3.2, optionally preceded by a plus or minus sign, but not including an integer suffix. If the value of *base* is between 2 and 36, the expected form of the subject sequence is a sequence of letters and digits from the portable character set representing an integer with the radix specified by *base*, optionally preceded by a plus or minus sign, but not including an integer suffix. The letters from a (or A) through z (or Z) are ascribed the values 10 to 35; only letters whose ascribed values are less than that of *base* are permitted. If the value of *base* is 16, the characters 0x or 0X may optionally precede the sequence of letters and digits, following the sign if present.

The subject sequence is defined as the longest initial subsequence of the input wide-character string, starting with the first non-white space wide character, that is of the expected form. The subject sequence contains no wide characters if the input wide-character string is empty or consists entirely of white space, or if the first non-white space wide character is other than a sign or a permissible letter or digit.

If the subject sequence has the expected form and the value of *base* is zero, the sequence of wide characters starting with the first digit is interpreted as an integer constant according to the rules of ISO/IEC 9899: subclause 6.1.3.2. If the subject sequence has the expected form and the value of *base* is between 2 and 36, it is used as the *base* for conversion, ascribing to each letter its value as given above. If the subject sequence begins with a minus sign, the value resulting from the conversion is negated. A pointer to the final wide-character string is stored in the object pointed to by endptr, provided that endptr is not a NULL pointer.

In a locale other than the C or POSIX locale, additional implementation-defined subject sequence forms may be accepted.

If the subject sequence is empty or does not have the expected form, no conversion is performed; the value of *nptr* is stored in the object pointed to by endptr, provided that endptr is not a NULL pointer. The behavior of this wide-character function is affected by the LC_CTYPE category of the current locale. If you change the category, undefined results can occur.

Since 0, {LLONG_MIN} and {LLONG_MAX} are returned on error and are also valid returns on success, an application wishing to check for error situations should set errno to 0, then call wcstoll(), then check errno.

Note: The wcstoll() function has a dependency on the level of the Enhanced ASCII Extensions. See “Enhanced ASCII support” on page 2113 for details.

Returned value

If successful, wcstoll() returns the converted value, if any.

If unsuccessful, wcstoll() returns 0 if no conversion could be performed. If the correct value is outside the range of representable values, wcstoll() returns

LLONG_MAX (LONGLONG_MAX) or LLONG_MIN (LONGLONG_MIN), according to the sign of the value. If the value of base is not supported, wcstoll() returns 0.

If unsuccessful wcstoll() sets errno to one of the following values:

Error Code
Description

EINVAL
The value of *base* is not supported.

ERANGE
The conversion caused an overflow.

Example

```
/* Long Long example */
 #define _LONG_LONG 1
 #include <stdio.h>
 #include <wchar.h>

 int main(void)
 {
    wchar_t *wcs;
    wchar_t *stopwcs;
    long long     l;
    int      base;

    wcs = L"10110134932";
    printf("wcs = `%ls`\n", wcs);
    for (base=2; base<=8; base*=2) {
       l = wcstoll(wcs, &stopwcs, base);
       printf("   wcstoll = %lld\n", l);
       printf("   Stopped scan at `%ls`\n\n", stopwcs);
    }
 }
```

Related information

- “wchar.h” on page 85
- “strtol() — Convert character string to long” on page 1775
- “strtoll() — Convert string to signed long long” on page 1778
- “strtoul() — Convert string to unsigned integer” on page 1780
- “strtoull() — Convert string to unsigned long long” on page 1782
- “wcstoimax() — Convert a wide-character string to a intmax_t” on page 2031
- “wcstol() — Convert a wide-character string to a long integer” on page 2035
- “wcstoul() — Convert a wide-character string to an unsigned long integer” on page 2043
- “wcstoull() — Convert a wide-character string to an unsigned long long integer” on page 2045
- “wcstoumax() — Convert a wide-character string to a intmax_t” on page 2047

wcstombs() — Convert wide-character string to multibyte character string

Standards

Standards / Extensions	C or C++	Dependencies
ISO C Amendment XPG4 XPG4.2 C99 Single UNIX Specification, Version 3	both	

Format

```
#include <stdlib.h>

size_t wcstombs(char * __restrict__dest,
                const wchar_t * __restrict__string, size_t count);
```

General description

Converts the wide-character string pointed to by *string* into the multibyte array pointed to by *dest*. The converted string begins in the initial shift state. The conversion stops after *count* bytes in *dest* are filled up or a NULL wide character is encountered.

The behavior of this wide-character function is affected by the LC_CTYPE category of the current locale. If you change the category, undefined results can occur.

If unsuccessful, mbstowcs() returns (size_t) -1 and sets errno to one of the following values:

Error Code
Description

EINVAL
The input argument is a NULL rather than a pointer to a wide character.

EILSEQ
wcstombs() encountered a sequence that is not a valid wide character code.

The mbstowcs() interface checks for an invalid input arg. If the third arg, which should be a pointer to a string of wchar_t elements, is NULL, the interface will set errno to EINVAL as well as returning -1. This clearly differentiates from the situation in which the third arg is a pointer to null, in which case should return 1 and the multibyte target string will contain a null byte.

Returned value

Returns the length in bytes of the multibyte character string, not including a terminating NULL wide character. The value `(size_t)-1` is returned if an invalid multibyte character is encountered or if `*string` is a NULL pointer.

If *count* is the returned value, the array is not NULL-terminated.

If **dest* is a NULL pointer, the number of characters required to convert the wide-character string is returned.

If the area pointed to by **dest* is too small (as indicated by the value of *count*) to contain the wide character codes represented as multibyte characters, the number of bytes containing complete multibyte characters is returned.

Note: wcstombs() does not generate redundant shift characters between the DBCS characters. When the wctomb() function is called for each character, redundant shift characters are generated.

Example

CELEBW24

```
/* CELEBW24

   In this example, a wide-character string is converted to a
   char string twice.  The first call converts the entire string, while
   the second call only converts three characters.  The results are
   printed each time.

 */
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#define SIZE 20

int main(void)
{
  char dest[SIZE];
  wchar_t * dptr = L"string";
  size_t count = SIZE;
  size_t length;

  length = wcstombs( dest, dptr, count );
  printf( "%d characters were converted.\n", length );
  printf( "The converted string is \"%s\"\n\n", dest );

  /* Reset the destination buffer */
  memset( dest, '\0', sizeof(dest));

  /* Now convert only 3 characters */
  length = wcstombs( dest, dptr, 3 );
  printf( "%d characters were converted.\n", length );
  printf( "The converted string is \"%s\"\n", dest );
}
```

Output:

```
6 characters were converted.
The converted string is "string"

3 characters were converted.
The converted string is "str"
```

Related information

- "stdlib.h" on page 70
- "mbstowcs() — Convert multibyte characters to wide characters" on page 1055
- "wcslen() — Calculate length of wide-character string" on page 2013
- "wcsrtombs() — Convert wide-character string to multibyte string" on page 2021
- "wctomb() — Convert wide character to multibyte character" on page 2053

wcstoul() — Convert a wide-character string to an unsigned long integer

Standards

Standards / Extensions	C or C++	Dependencies
ISO C Amendment XPG4 XPG4.2 C99 Single UNIX Specification, Version 3	both	

Format

```
#include <wchar.h>

unsigned long int wcstoul(const wchar_t * __restrict__nptr,
                          wchar_t ** __restrict__endptr, int base);
```

General description

Converts the initial portion of the wide-character string pointed to by *nptr* to unsigned long integer representation. First it decomposes the input wide-character string into three parts:

1. An initial, possibly empty, sequence of white space wide characters (as specified by the iswspace() function).
2. A subject sequence resembling an unsigned integer represented in some radix determined by the value of base.
3. A final wide-character string of one or more unrecognized wide characters, including the terminating NULL character of the input wide-character string.

Then it attempts to convert the subject sequence to an unsigned integer, and returns the result.

If the value of base is zero, the expected form of the subject sequence is that of an integer constant as described in ISO/IEC 9899: subclause 6.1.3.2, optionally preceded by a plus or minus sign, but not including an integer suffix. If the value of base is between 2 and 36, the expected form of the subject sequence is a sequence of letters and digits from the portable character set representing an integer with the radix specified by base, optionally preceded by a plus or minus sign, but not including an integer suffix. The letters from a (or A) through z (or Z) are ascribed the values 10 to 35; only letters whose ascribed values are less than that of base are permitted. If the value of base is 16, the characters 0x or 0X may optionally precede the sequence of letters and digits, following the sign if present.

The *subject sequence* is defined as the longest initial sub-sequence of the input wide-character string, starting with the first non-white space wide character, that is of the expected form. The subject sequence contains no wide characters if the input wide-character string is empty or consists entirely of white space, or if the first non-white space wide character is other than a sign or a permissible letter or digit.

If the subject sequence has the expected form and the value of base is zero, the sequence of wide characters starting with the first digit is interpreted as an integer constant according to the rules of ISO/IEC 9899: subclause 6.1.3.2. If the subject sequence has the expected form and the value of base is between 2 and 36, it is used as the base for conversion, ascribing to each letter its value as given above. If the subject sequence begins with a minus sign, the value resulting from the conversion is negated. A pointer to the final string is stored in the object pointed to by *endptr*, provided that *endptr* is not a NULL pointer.

In a locale other than the C or POSIX locale, additional implementation-defined subject sequence forms may be accepted.

If the subject sequence is empty or does not have the expected form, no conversion is performed; the value of *nptr* is stored in the object pointed to by *endptr*, provided that *endptr* is not a NULL pointer.

The behavior of this wide-character function is affected by the LC_CTYPE category of the current locale. If you change the category, undefined results can occur.

Since 0 and {ULONG_MAX} are returned on error and 0 is also a valid return on success, an application wishing to check for error situations should set errno to 0, then call wcstoul(), then check errno.

Returned value

If successful, wcstoul() returns the converted value, if any.

If unsuccessful, wcstoul() returns 0 if no conversion could be performed. If the correct value is outside the range of representable values, wcstoul() returns ULONG_MAX. If the value of base is not supported, wcstoul() returns 0.

If unsuccessful wcstoul() sets errno to one of the following values:

Error Code
 Description

EINVAL
 The value of *base* is not supported.

ERANGE
 The conversion caused an overflow.

Example

CELEBW25

```
/* CELEBW25 */
#include <stdio.h>
#include <wchar.h>

#define BASE 2

int main(void)
{
   wchar_t *wcs = L"1000e13 camels";
   wchar_t **endptr;
   unsigned long int answer;

   answer = wcstoul(wcs, endptr, BASE);
   printf("The input wide string used: `%ls`\n", wcs);
   printf("The unsigned long int produced: %lu\n", answer);
   printf("The substring of the input wide string that was not");
   printf(" converted to unsigned long: `%ls`\n", *endptr);
}
```

Related information

- “wchar.h” on page 85
- “strtoul() — Convert string to unsigned integer” on page 1780
- “wcstoimax() — Convert a wide-character string to a intmax_t” on page 2031
- “wcstoumax() — Convert a wide-character string to a intmax_t” on page 2047

wcstoull() — Convert a wide-character string to an unsigned long long integer

Standards

Standards / Extensions	C or C++	Dependencies
z/OS UNIX C99 Single UNIX Specification, Version 3 C++ TR1 C99	both	OS/390 V2R10

Format

```
#include <wchar.h>

unsigned long long wcstoull(const wchar_t * __restrict__nptr,
                            wchar_t ** __restrict__endptr, int base);
```

Compile requirement: Use of this function requires the long long data type. See *z/OS C/C++ Language Reference* for information on how to make long long available.

General description

Converts the initial portion of the wide-character string pointed to by nptr to unsigned long long integer representation. First it decomposes the input wide-character string into three parts:

1. An initial, possibly empty, sequence of white space wide characters (as specified by the iswspace() function).
2. A subject sequence resembling an unsigned integer represented in some radix determined by the value of *base*.
3. A final wide-character string of one or more unrecognized wide characters, including the terminating NULL character of the input wide-character string.

Then it attempts to convert the subject sequence to an unsigned long long integer, and returns the result.

If the value of *base* is zero, the expected form of the subject sequence is that of an integer constant as described in ISO/IEC 9899: subclause 6.1.3.2, optionally preceded by a plus or minus sign, but not including an integer suffix. If the value of *base* is between 2 and 36, the expected form of the subject sequence is a sequence of letters and digits from the portable character set representing an integer with the radix specified by *base*, optionally preceded by a plus or minus sign, but not including an integer suffix. The letters from a (or A) through z (or Z) are ascribed the values 10 to 35; only letters whose ascribed values are less than that of *base* are permitted. If the value of *base* is 16, the characters 0x or 0X may optionally precede the sequence of letters and digits, following the sign if present.

The subject sequence is defined as the longest initial sub-sequence of the input wide-character string, starting with the first non-white space wide character, that is of the expected form. The subject sequence contains no wide characters if the input wide-character string is empty or consists entirely of white space, or if the first non-white space wide character is other than a sign or a permissible letter or digit.

If the subject sequence has the expected form and the value of *base* is zero, the sequence of wide characters starting with the first digit is interpreted as an integer constant according to the rules of ISO/IEC 9899: subclause 6.1.3.2. If the subject sequence has the expected form and the value of *base* is between 2 and 36, it is used as the *base* for conversion, ascribing to each letter its value as given above. If the subject sequence begins with a minus sign, the value resulting from the conversion is negated. A pointer to the final string is stored in the object pointed to by *endptr*, provided that *endptr* is not a NULL pointer. In a locale other than the C or POSIX locale, additional implementation-defined subject sequence forms may be accepted.

If the subject sequence is empty or does not have the expected form, no conversion is performed; the value of nptr is stored in the object pointed to by *endptr*, provided that *endptr* is not a NULL pointer.

The behavior of this wide-character function is affected by the LC_CTYPE category of the current locale. If you change the category, undefined results can occur.

Since 0 and {ULLONG_MAX} are returned on error and 0 is also a valid return on success, an application wishing to check for error situations should set errno to 0, then call wcstoull(), then check errno.

Note: The wcstoull() function has a dependency on the level of the Enhanced ASCII Extensions. See "Enhanced ASCII support" on page 2113 for details.

Returned value

If successful, wcstoull() returns the converted value, if any.

If unsuccessful, wcstoull() returns 0 if no conversion could be performed. If the correct value is outside the range of representable values, wcstoull() returns ULLONG_MAX (ULONGLONG_MAX). If the value of base is not supported, wcstoull() returns 0.

If unsuccessful wcstoull() sets errno to one of the following values:

Error Code
Description

EINVAL
The value of *base* is not supported.

ERANGE
The conversion caused an overflow.

Example

```
/* LongLong conversion */
 #define _LONG_LONG 1
 #include <stdio.h>
 #include <wchar.h>

 #define BASE 2

 int main(void)
 {
    wchar_t *wcs = L"1000e13 camels";
    wchar_t **endptr;
    unsigned long long answer;

    answer = wcstoull(wcs, endptr, BASE);
    printf("The input wide string used: `%ls`\n", wcs);
    printf("The unsigned long long produced: %llu\n", answer);
    printf("The substring of the input wide string that was not");
    printf(" converted to unsigned long long: `%ls`\n", *endptr);
 }
```

Related information

- “wchar.h” on page 85
- “strtol() — Convert character string to long” on page 1775
- “strtoll() — Convert string to signed long long” on page 1778
- “strtoul() — Convert string to unsigned integer” on page 1780
- “strtoull() — Convert string to unsigned long long” on page 1782
- “wcstoimax() — Convert a wide-character string to a intmax_t” on page 2031
- “wcstol() — Convert a wide-character string to a long integer” on page 2035
- “wcstoll() — Convert a wide-character string to a long long integer” on page 2038
- “wcstoul() — Convert a wide-character string to an unsigned long integer” on page 2043
- “wcstoumax() — Convert a wide-character string to a intmax_t”

wcstoumax() — Convert a wide-character string to a intmax_t

Standards

Standards / Extensions	C or C++	Dependencies
C99 Single UNIX Specification, Version 3 C++ TR1 C99	both	z/OS V1R7

Format

```
#define _ISOC99_SOURCE
#include <inttypes.h>

uintmax_t wcstoumax(const wchar_t * __restrict__ nptr,
                    wchar_t ** __restrict__ endptr, int base);
```

Compile requirement: Function wcstoumax() requires long long to be available.

General description

The wcstoumax() function converts the wide-character string nptr to an uintmax_t integer type. Valid input values for base are 0 and in the range 2-36. The wcstoumax() function is equivalent to wcstoul() and wcstoull(). The only difference being that the return value is of type uintmax_t. See wcstoull() for more information.

Returned value

If successful, wcstoumax() returns the converted value, if any.

If unsuccessful, wcstoumax() returns 0 if no conversion could be performed. If the correct value is outside the range of representable values, wcstoumax() returns UINTMAX_MAX. If the value of base is not supported, wcstoumax() returns 0.

If unsuccessful wcstoumax() sets errno to one of the following values:

Error Code
Description

EINVAL
The value of *base* is not supported.

ERANGE
The conversion caused an overflow.

Example

```
#define _ISOC99_SOURCE
#include <inttypes.h>
#include <stdio.h>

 int main(void)
{
   wchar_t *nptr;
   wchar_t *endptr;
   uintmax_t  j;
   int base = 10;
   nptr = L"10110134932";
   printf("nptr = `%ls`\n", nptr);
      j = wcstoumax(nptr, &endptr, base);
      printf("wcstoumax = %ju\n", j);
      printf("Stopped scan at `%ls`\n\n", endptr);
}

Output

nptr = `10110134932`
wcstoumax = 10110134932
Stopped scan at ``
```

Related information

- “inttypes.h” on page 33
- “stdint.h” on page 65
- “imaxdiv() — Quotient and remainder for intmax_t” on page 839
- “strtoimax() — Convert character string to intmax_t integer type” on page 1770
- “strtoumax() — Convert character string to uintmax_t integer type” on page 1784
- “wcstoimax() — Convert a wide-character string to a intmax_t” on page 2031

wcswcs() — Locate wide-character substring in wide-character string

Standards

Standards / Extensions	C or C++	Dependencies
ISO C Amendment XPG4 XPG4.2 C99 Single UNIX Specification, Version 3	both	

Format

```
#include <wchar.h>

wchar_t *wcswcs(const wchar_t *string1, const wchar_t *string2);
```

General description

Locates the first occurrence in the string pointed to by *string1* of the sequence of wide characters (excluding the terminating wide NULL character) in the string pointed to by *string2*.

The behavior of this wide-character function is affected by the LC_CTYPE category of the current locale. If you change the category, undefined results can occur.

Note: The wcswcs() function has been moved to the Legacy Option group in Single UNIX Specification, Version 3 and may be withdrawn in a future version. The wcsstr() function is preferred for portability.

Returned value

If successful, wcswcs() returns a pointer to the located string.

If *string2* points to a string with zero length, wcswcs() returns *string1*.

If the string is not found, wcswcs() returns NULL.

Example

CELEBW26

```
/* CELEBW26

   This example finds the first occurrence of the wide character string pr
   in buffer1, using wcswcs().

 */
#include <stdio.h>
#include <wcstr.h>

#define SIZE 40

int main(void)
{
  wchar_t buffer1[SIZE] = L"computer program";
  wchar_t *ptr;
  wchar_t *wch = L"pr";

  ptr = wcswcs( buffer1, wch );
  printf( "The first occurrence of %ls in '%ls' is '%ls'\n",
                          wch, buffer1, ptr );
}
```

Output:

```
The first occurrence of pr in 'computer program' is 'program'
```

Related information

- “wchar.h” on page 85
- “strstr() — Locate substring” on page 1762
- “wcschr() — Search for wide-character substring” on page 2004
- “wcscmp() — Compare wide-character strings” on page 2005

- "wcscspn() — Find offset of first wide-character match" on page 2009
- "wcspbrk() — Locate first wide characters in string" on page 2018
- "wcsrchr() — Locate last wide character in string" on page 2019

wcswidth() — Determine the display width of a wide-character string

Standards

Standards / Extensions	C or C++	Dependencies
XPG4 XPG4.2 Single UNIX Specification, Version 3	both	

Format

```
#include <wchar.h>

int wcswidth(const wchar_t *wcs, size_t n);
```

General description

Determines the number of printing positions that a graphic representation of *n* wide characters (or fewer than *n* wide characters, if a NULL wide character is encountered before *n* wide characters have been exhausted), in the wide-character string pointed to by *wcs*, occupies on a display device. The number of printing positions is independent of its location on the device.

Returned value

If successful, wcswidth() returns the number of printing positions occupied by the wide-character string pointed to by *wcs*.

If *wcs* points to a NULL wide character, wcswidth() returns 0.

If any wide character in the wide-character string pointed to by *wcs* is not a printing wide character, wcswidth() returns -1.

The behavior of wcswidth() is affected by the LC_CTYPE category.

Note: Under z/OS XL C/C++ applications, the width returned will be 1 for each single-byte character and 2 for each double-byte character.

Example

CELEBW27

```
/* CELEBW27 */
#include <stdio.h>
#include <wchar.h>

int main(void)
{
   wchar_t *wcs = L"ABC";

   printf("wcs has a width of: %d\n", wcswidth(wcs,3));
}
```

Output:

```
wcs has a width of: 3
```

Related information

- "wchar.h" on page 85

wcsxfrm() — Transform a wide-character string

Standards

Standards / Extensions	C or C++	Dependencies
ISO C Amendment XPG4 XPG4.2 C99 Single UNIX Specification, Version 3	both	

Format

```
#include <wchar.h>

size_t wcsxfrm(wchar_t * __restrict__wcs1,
               const wchar_t * __restrict__wcs2, size_t n);
```

General description

Transforms the wide-character string pointed to by *wcs2* to values which represent character collating weights and places the resulting wide-character string into the array pointed to by *wcs1*. The transformation is such that if the wcscmp() function is applied to two transformed wide-character strings, it returns a value greater than, equal to, or less than zero, corresponding to the result of the wcscoll() function applied to the same two original wide-character strings. No more than *n* elements are placed into the resulting array pointed to by *wcs1*, including the terminating NULL wide-character code. If *n* is zero, *wcs1* is permitted to be a NULL pointer. If copying takes place between objects that overlap, the behavior is undefined.

Since no return value is reserved to indicate an error, an application wishing to check for error situations should set errno to 0, then call wcsxfrm(), then check errno.

Returned value

Returns the length of the transformed wide-character string (not including the terminating NULL wide character code). If the value returned is *n* or more, the contents of the array pointed to by *wcs1* are indeterminate.

If *wcs1* is a NULL pointer, wcsxfrm() returns the number of elements required to contain the transformed wide string.

The transformed value of invalid wide-character codes shall be either less than or greater than the transformed values of valid wide-character codes depending on the option chosen for the particular locale definition. In this case wcsxfrm() returns (size_t)-1.

wcsxfrm() is controlled by the LC_COLLATE category.

The EILSEQ error may be set, indicating that the wide character string pointed to by *wcs2* contains wide character codes outside the domain of the collating sequence.

Note: The ISO/C Multibyte Support Extensions do not indicate that the wcsxfrm() function may return with an error.

Example

CELEBW28

```
/* CELEBW28 */
#include <stdio.h>
#include <wchar.h>

int main(void)
{
   wchar_t *wcs;
   wchar_t  buffer[80];
   int      length;

   printf("Type in a string of characters.\n");
   wcs = fgetws(buffer, 80, stdin);
   length = wcsxfrm(NULL, wcs, 0);
   printf("You would need a %d element array to hold the wide string", length);
   printf("\n\n%ls\n\ntransformed according", wcs);
   printf(" to this program's locale.\n");
}
```

Related information

- “wchar.h” on page 85
- “strxfrm() — Transform string” on page 1785

wctob() — Convert wide character to byte

Standards

Standards / Extensions	C or C++	Dependencies
ISO C Amendment C99 Single UNIX Specification, Version 3	both	

Format

Non-XPG4:

```
#include <wchar.h>

int wctob(wint_t c);
```

XPG4:

```
#define _XOPEN_SOURCE
#define _MSE_PROTOS
#include <wchar.h>

int wctob(wint_t c);
```

General description

Determines whether *c* corresponds to a member of the extended character set whose multibyte character corresponds to a single byte when in initial shift state.

The behavior of this wide-character function is affected by the LC_CTYPE category of the current locale. If you change the category, undefined results can occur.

Special behavior for XPG4: If you define any feature test macro specifying XPG4 behavior before the statement in your program source file to include the wchar header, then you must also define the _MSE_PROTOS feature test macro to make the declaration of the wctob() function in the wchar header available when you compile your program. Please see Table 2 on page 4 for a list of XPG4 and other feature test macros.

Returned value

If successful, wctob() returns the single-byte representation.

If *c* does not correspond to a multibyte character with length one, wctob() returns EOF.

Example

CELEBW29

```
/* CELEBW29 */
#include <stdio.h>
#include <wchar.h>

int main(void)
{
   wint_t wc = L'A';

   if (wctob(wc) == wc)
      printf("wc is a valid single byte character\n");
   else
      printf("wc is not a valid single byte character\n");
}
```

Output:

```
wc is a valid single-byte character
```

Related information

- “wchar.h” on page 85

wctomb() — Convert wide character to multibyte character

Standards

Standards / Extensions	C or C++	Dependencies
ISO C Amendment XPG4 XPG4.2 C99 Single UNIX Specification, Version 3	both	

Format

```
#include <stdlib.h>

int wctomb(char *string, wchar_t character);
```

General description

Converts the wchar_t value of *character* into a multibyte array pointed to by *string*. If the value of *character* is 0, the function is left in the initial shift state. At most, wctomb() stores **MB_CUR_MAX** characters in *string*.

The behavior of this wide-character function is affected by the LC_CTYPE category of the current locale. If you change the category, undefined results can occur.

Returned value

If successful, wctomb() returns the length in bytes of the multibyte character.

If *character* is not a valid multibyte character, wctomb() returns -1.

If *string* is a NULL pointer, wctomb() returns nonzero if shift-dependent encoding is used, or 0 otherwise.

Example

CELEBW30

```
/* CELEBW30

   This example converts the wide character c to a character using wctomb().

 */
#include <stdio.h>
#include <stdlib.h>

#define SIZE 40

int main(void)
{
  static char  buffer[ SIZE ];
  wchar_t wch = L'c';
  int length;

  length = wctomb( buffer, wch );
  printf( "The number of bytes that comprise the multibyte "
            "character is %i\n", length );
  printf( "And the converted string is \"%s\"\n", buffer );
}
```

Output:

```
The number of bytes that comprise the multibyte character is 1
And the converted string is "c"
```

Related information

- “stdlib.h” on page 70
- “mbtowc() — Convert multibyte character to wide character” on page 1056
- “wcrtomb() — Convert a wide character to a multibyte character” on page 2001
- “wcslen() — Calculate length of wide-character string” on page 2013
- “wcstombs() — Convert wide-character string to multibyte character string” on page 2041

wctrans(), towctrans() — Transliterate wide character

Standards

Standards / Extensions	C or C++	Dependencies
ISO C Amendment C99 Single UNIX Specification, Version 3	both	

Format

```
#include <wchar.h>

   wctrans_t  wctrans(const char * charclass);
   wint_t   towctrans(wint_t wc, wctrans_t desc);
```

General description

These two functions work together to provide transliteration of wide characters. For valid results, the setting of the LC_CTYPE category of the current locale must remain the same across the two calls. Character mapping rules are defined in the LC_CTYPE category.

The wctrans() function takes the character mapping name pointed to by *charclass* and returns a value of type `wctrans_t` for use as the second argument to the towctrans() function.

The towctrans() function applies the indicated mapping to wide-character code *wc* from the codeset identified in the current locale. The towctrans() function applies the mapping returned by wctrans() and passed as *desc*.

The following character mapping names are reserved by the standard and are defined in all locales: *tolower* and *toupper*.

Notes:

1. towctrans(wc, wctrans("tolower")) is equivalent to towlower(wc)
2. towctrans(wc, wctrans("toupper")) is equivalent to towupper(wc)

Returned value

If successful, wctrans() returns the non-zero value of type `wctrans_t` for use in calls to towctrans().

If unsuccessful, wctrans() returns 0 and sets errno to EINVAL if the mapping name pointed to by *charclass* is not valid for the current locale.

If successful, the towctrans() function returns the mapped value of *wc* using the mapping described by *desc*.

If unsuccessful, towctrans() returns *wc* unchanged. If the value of *desc* is invalid, towctrans() returns 0.

Related information

- "wctype.h" on page 86
- "tolower(), toupper() — Convert character case" on page 1897

wctype() — Obtain handle for character property classification

Standards

Standards / Extensions	C or C++	Dependencies
ISO C Amendment XPG4 XPG4.2 C99 Single UNIX Specification, Version 3	both	

Format

```
#include <wchar.h>

wctype_t wctype(const char *property);
```

SUSV3:

```
#define _POSIX_C_SOURCE 200112L
#include <wctype.h>
wctype_t wctype(const char *property);
```

General description

The wctype() function is defined for valid property names as defined in the current locale. The *property* is a string identifying a generic character class for which code-page-specific type information is required. The function returns a value of type *wctype_t*, which can be used as the second argument to a call of iswctype(). The wctype() function determines values of *wctype_t* according to rules of the coded character set defined by character type information in the program's locale (category LC_CTYPE). Values returned by wctype() are valid until a call to setlocale() that modifies the category LC_CTYPE.

The behavior of this wide-character function is affected by the LC_CTYPE category of the current locale. If you change the category, undefined results can occur.

Returned value

If successful, wctype() returns a value of type *wctype_t* that can be used in calls to iswctype().

If the given property name is not valid for the current locale (category LC_CTYPE), wctype() returns 0.

Related information

- "wchar.h" on page 85
- "wctype.h" on page 86

wcwidth() — Determine the display width of a wide character

Standards

Standards / Extensions	C or C++	Dependencies
XPG4 XPG4.2 Single UNIX Specification, Version 3	both	

Format

Non-XPG4:

```
#include <wchar.h>

int wcwidth(const wint_t wc);
```

XPG4:

```
#define _XOPEN_SOURCE
#include <wchar.h>

int wcwidth(const wchar_t wc);
```

General description

Determines the number of printing positions that a graphic representation of *wc* occupies on a display device. Each of the printing wide characters occupies its own number of printing positions on a display device. The number is independent of its location on the device.

Special behavior for XPG4: If you define any feature test macro specifying XPG4 behavior before the statement in your program source file to include the wchar header, then the compiler assumes that your program is using the XPG4 variety of the wcwidth() function. Please see Table 2 on page 4 for a list of XPG4 and other feature test macros.

The prototype for the XPG4 variety of the wcwidth() function is:

```
int wcwidth(const wchar_t wc);
```

The difference between this variety and the C/370, non-XPG4 variety of the wcwidth() function is that its parameter, `wc`, is a wchar_t rather than a wint_t type.

Returned value

If successful, wcwidth() returns the number of printing positions occupied by *wc*.

If *wc* is a NULL or non-spacing wide character, wcwidth() returns 0.

If *wc* is not a printing wide character, wcwidth() returns -1.

The behavior of wcwidth() is affected by the LC_CTYPE category.

Notes:

1. Under z/OS XL C/C++ applications, the width returned will be zero for a NULL or non-spacing character, 1 for a single-byte character, and 2 for a double-byte character.
2. A non-spacing character is a character belonging to the charclass named _zlc (zero length character class) in the LC_CTYPE category.

Example

CELEBW31

```
/* CELEBW31 */
#include <stdio.h>
#include <wchar.h>
```

```
int main(void)
{
   wint_t wc = L'A';

   printf("wc has a width of: %d\n", wcwidth(wc));
}
```

Output:

```
wc has a width of: 1
```

Related information

- “wchar.h” on page 85

w_getmntent() — Get information on mounted file systems

Standards

Standards / Extensions	C or C++	Dependencies
z/OS UNIX	both	

Format

```
#define _OPEN_SYS
#include <sys/mntent.h>

int w_getmntent(char *buffer, int size);
```

General description

Gets information about all currently mounted file systems.

buffer A pointer to storage that is to be filled with the retrieved information. The storage consists of a header and an entry table that correspond to one of the available mapping formats. The mappings available are defined in the <sys/mntent.h> header file. Each entry in the entry table will contain information for a single mounted file system.

size The length of the buffer. If *size* is zero, the total number of mount entries is returned. You can use this information to obtain a buffer large enough to hold all the information about all the entries. Use the new buffer on the subsequent call.

Three mapping formats available. Their corresponding headers are:

	Header	Eye Catcher	Entry
struct mnte3	MNTE3H (struct mnte3h)	MNTE3H_ID ("MNT3")	W_MNTENT3 (struct w_mntent3)
struct mnte2	MNTE2H (struct mnte2h)	MNTE2H_ID ("MNT2")	W_MNTENT2 (struct w_mntent2)
struct mnte	W_MNTH (struct w_mnth)	"MNTE"	W_MNTENT (struct w_mntent)

The buffer should be set to zero prior to setting of fields or the w_getmntent() call. The default mapping format is struct mnte when the eyecatcher field is unset.

Using structure w_mntent3: Prior to calling w_getmntent() with the w_mntent3 structure mapping, set the following fields in MNTE3H:

1. mnt3H_cbid eyecatcher to MNTE3H_ID
2. mnt3H_cblen to the size of struct mnte3
3. mnt3H_bodylen to the size of struct w_mntent3

In the header, the field mnt3H_cblen returns the number of bytes of data put in the buffer. The field mh3_cursor contains positioning information that w_getmntent() uses to store the information. If multiple calls are made, use the same buffer because the positioning information in mh3_cursor indicates where the function should continue with its list. The positioning information should not be changed between calls.

After a successful call to w_getmntent(), the following w_mntent3 structure fields will be available for reference:

mnt3_fstype
The file system type.

mnt3_mode
The mount mode of the file system. A flag field that specifies the mount mode and additional mount options: mntentfsaunmount.

mntentfsaunmount
If it is 1 after the file system is mounted, the file system will be unmounted when a system leaves the sysplex. If it is 0, then the setting of mntentfsnoautomove will be used. See mntentfsnoautomove below.

mntentfsclient
If it is 0, then the file system is a sysplex client. If it is 1, then the file system is not a sysplex client.

mntentfsnoautomove
If it is 0 after the file system is mounted, it can be moved automatically. If it is 1 after the file system is mounted, it will not be moved automatically.

mntentfsmodenosec
If it is 1, then the file system will not enforce security checks. If it is 0, then the file system will enforce security checks.

mntentfsmodeexport
If it is 0, then the file system has not been exported by DFS. If it is 1, then the file system has been exported by DFS.

mntentfsmoderdonly
If it is 0, then the file system is mounted as read/write. If it is 1, then the file system is mounted as read-only.

mntentfsmodenosuid
If it is 1, then the SETUID and SETGID mode flags will be ignored for programs that reside in this file system. If it is 0 then the SETUID and SETGID mode flags will be enforced for programs that reside in this file system.

mnt3_dev
Device number which stat() will return for all files in this file system.

mnt3_parentdev
st_dev of parent file system.

mnt3_rootino
The ino of the mount point.

mnt3_status
The status of the file system

mnt3_ddname
The ddname specified on mount.

mnt3_fstname
The name of the file system type specified by the FILESYS statement. The name for the file system that performed the mount.

mnt3_fsname
The name of the file system that was mounted.

mnt3_pathlen
The length of mount point path.

mnt3_mountpoint
The name of the directory where the file system is mounted.

mnt3_jobname
If this file system is quiesced, this is the job that made the request.

mnt3_PID
If this file system is quiesced, this is the PID that made the request.

mnt3_parmoffset
Offset of mount parameter parm_point, starting from the address of the mnt3_fstype member of the w_mnte3 struct.

mnt3_parmlen
The length of the mount parameter parm_point.

mnt3_sysname
The name of the target system.

mnt3_qsystem
The name of the quiesce system.

mnt3_fromsys
The name of the system from which the file system has moved.

mnt3_flags
The field containing the request flags.

mnt3_status2
The file system status extensions.

mnt3_readct
The number of reads done.

mnt3_writect
The number of writes done.

mnt3_diribc
The number of directory I/O blocks.

mnt3_readibc
The number of read I/O blocks.

mnt3_writeibc
The number of write I/O blocks.

mnt3_bytesreadhw
Total number of bytes read from high word.

mnt3_bytesreadlw
Total number of bytes read from low word.

mnt3_byteswrittenhw
Total number of bytes written to high word.

mnt3_byteswrittenlw
Total number of bytes written to low word.

mnt3_filetag
Mount tag.

mnt3_syslistoffset
Offset of system list.

mnt3_syslistlength
Length of system list.

mnt3_aggnamelength
Length of the aggregate name in mnt2_aggname. The length does not include the null terminating character, and is only valid if mnt3_aggnameoffset has a non-zero value.

mnt3_aggnameoffset
The offset of mnt2_aggname from w_mntent3. If the value is zero, then no aggregate name is returned.

mnt3_mntsec
The mount time in seconds when the file system was mounted.

mnt3_fstoken
This field is a virtual file system pointer for the file system.

mnt3_pfsnormalstatusoffset
This field provides the offset of the physical file system normal status string location.

mnt3_pfsnormalstatuslength
This field provides the length of the physical file system normal status string.

mnt3_pfsexcpstatuslength
This field provides the length of the exception status string of the physical file system.

mnt3_pfsexcptstatusoffset
This field provides the offset of the exception status string of the physical file system.

mnt3_fsusrmntUID
The effective UID (EUID) of the user that mounted the file system.

parm_point
This field contains the mount point parameters to be used when mounting a file system. It is a separate field in the mnte3 structure but contiguously allocated following the w_mnte3 body, its address is the sum of the address of w_mnte3 and mnt3_parmoffset.

If all entries do not fit in the buffer supplied, multiple calls are required. If an entry together with its mount parameter will not fit in the buffer, the entry is returned without the mount parameter. In this case, *mnt3_parmlen* contains the length of the mount parameter, and *mnt3_parmoffset* is zero.

To assure that at least one entry, including the mount parameter, is returned, it is advisable to allocate space for at least two entries.

When the final entry has been placed in the buffer, w_getmntent() returns no entries.

Using structure w_mntent2: Prior to calling w_getmntent() with the w_mntent2 structure mapping, set the following fields in MNTE2H:

1. mh2_hdr.mnth2_cbid eyecatcher to MNTE2H_ID
2. mh2_hdr.mnt2h_cblen to the size of struct mnte2
3. mh_bodylen to the size of struct w_mntent2

In the header, the field mnt2h_cblen returns the number of bytes of data put in the buffer. The field mh2_cursor contains positioning information that w_getmntent() uses to store the information. If multiple calls are made, use the same buffer because the positioning information in mh2_cursor indicates where the function should continue with its list. The positioning information should not be changed between calls.

After a successful call to w_getmntent(), the following w_mntent2 structure fields will be available for reference:

mnt2_fstype
The file system type.

mnt2_mode

The mount mode of the file system. A flag field that specifies the mount mode and additional mount options:

mntentfsaunmount
If it is 1 after the file system is mounted, the file system will be unmounted when a system leaves the sysplex. If it is 0, then the setting of mntentfsnoautomove will be used. See mntentfsnoautomove below.

mntentfsclient
If it is 0, then the file system is a sysplex client. If it is 1, then the file system is not a sysplex client.

mntentfsnoautomove
If it is 0 after the file system is mounted, it can be moved automatically. If it is 1 after the file system is mounted, it will not be moved automatically.

mntentfsmodenosec
If it is 1, then the file system will not enforce security checks. If it is 0, then the file system will enforce security checks.

mntentfsmodeexport
If it is 0, then the file system has not been exported by DFS. If it is 1, then the file system has been exported by DFS.

mntentfsmoderdonly
If it is 0, then the file system is mounted as read/write. If it is 1, then the file system is mounted as read-only.

mntentfsmodenosuid
If it is 1, then the SETUID and SETGID mode flags will be ignored for programs that reside in this file system. If it is 0 then the SETUID and SETGID mode flags will be enforced for programs that reside in this file system.

mnt2_dev
Device # which stat() will return for all files in this file system.

mnt2_parentdev
st_dev of parent file system.

mnt2_rootino
The ino of the mount point.

mnt2_status
status of the file system.

mnt2_ddname
The ddname specified on mount.

mnt2_fstname
The name of the file system type specified by the FILESYS statement. The name for the file system that performed the mount.

mnt2_fsname
The name of the file system that was mounted.

mnt2_pathlen
The length of mount point path.

mnt2_mountpoint
The name of the directory where the file system is mounted.

mnt2_jobname
If this file system is quiesced, this is the job that made the request.

mnt2_PID
If this file system is quiesced, this is the PID that made the request.

mnt2_parmoffset
Offset of mount parameter parm_point, starting from the address of the mnt2_fstype member of the w_mnte2 struct.

mnt2_parmlen
The length of the mount parameter parm_point.

mnt2_sysname
The name of the target system.

mnt2_qsystem
The name of the quiesce system.

mnt2_fromsys
The name of the system from which the file system has moved.

mnt2_flags
The field containing the request flags.

mnt2_status2
The file system status extensions.

mnt2_success
This field is used to return the number of successfully moved file systems when moving a collection of file systems.

mnt2_readct
The number of reads done.

mnt2_writect
The number of writes done.

mnt2_diribc
The number of directory I/O blocks.

mnt2_readibc
The number of read I/O blocks.

mnt2_writeibc
The number of write I/O blocks.

mnt2_bytesreadhw
Total number of bytes read from high word.

mnt2_bytesreadlw
Total number of bytes read from low word.

mnt2_byteswrittenhw
Total number of bytes written to high word.

mnt2_byteswrittenlw
Total number of bytes written to low word.

mnt2_filetag
Mount tag.

mnt2_syslistoffset
Offset of system list.

mnt2_syslistlength
Length of system list.

mnt2_aggnamelength
Length of the aggregate name in mnt2_aggname. The length does not include the null terminating character, and is only valid if mnt2_aggnameoffset has a non-zero value.

mnt2_aggnameoffset
The offset of mnt2_aggname from w_mntent2. If the value is zero, then no aggregate name is returned.

parm_point
This field contains the mountpoint parameters to be used when mounting a file system. It is a separate field in the mnte2 structure but contiguously allocated following the w_mnte2 body, its address is the sum of the address of w_mnte2 and mnt2_parmoffset.

If all entries do not fit in the buffer supplied, multiple calls are required. If an entry together with its mount parameter will not fit in the buffer, the entry is returned without the mount parameter. In this case, *mnt2_parmlen* contains the length of the mount parameter, and *mnt2_parmoffset* is zero.

To assure that at least one entry, including the mount parameter, is returned, it is advisable to allocate space for at least two entries.

When the final entry has been placed in the buffer, w_getmntent() returns no entries.

Using structure w_mntent: Prior to calling w_getmntent() with the w_mntent structure mapping, set the following fields in W_MNTH:

1. mnth_hid eyecather to "MNTE"
2. mnth_size to the size of struct mnte
3. mh_bodylen to the size of struct w_mntent

In the header, the field mnth_size returns the number of bytes of data put in the buffer. The fields mnth_cur1 and mnth_cur2 contain positioning information that w_getmntent() uses to store the information. If multiple calls are made, use the same buffer because the positioning information in mnth_cur1 and mnth_cur2 indicates where the function should continue with its list. The positioning information should not be changed between calls.

After a successful call to w_getmntent(), the following w_mntent structure fields will be available for reference:

mnt_fstype
The file system type.

mnt_mode
File system mount mode.

mnt_dev
Device # which stat() will return for all files in this file system.

mnt_parentdev
st_dev of parent file system.

mnt_rootino
The ino of the mount point.

mnt_status
The status of the file system.

mnt_ddname
The ddname specified on mount.

mnt_fstname
The name of the file system type specified by the FILESYS statement.

mnt_fsname
The file system name, which is up to 45 characters long and ends with a NULL.

mnt_pathlen
The length of mount point path.

mnt_mountpoint

The pathname of the directory where the file system is mounted. This field ends with a NULL.

If the caller of w_getmntent() lacks search authorization to one or more of the directories in the mount point pathname, *mnt_mountpoint* is returned empty. That is, *mnt_pathlen* is zero and *mnt_mountpoint* contains a NULL as the first character.

mnt_jobname
If the file system is quiesced, this is the job that made the request.

mnt_PID
If the file system is quiesced, this is the PID that made the request.

mnt_parmoffset
Offset of mount parameter from w_mntent.

mnt_parmlen
Length of mount parameter.

mnt_parm

The file-system-specific parameter specified on the mount() function when the file system was mounted. This field ends with a NULL.

If no parameter was specified, *mnt_parmlen* and *mnt_parmoffset* are each zero. If a parameter was specified, its address is the sum of the address of *w_mntent* and *mnt_parmoffset*.

If all entries do not fit in the buffer supplied, multiple calls are required. If an entry together with its mount parameter will not fit in the buffer, the entry is returned without the mount parameter. In this case, *mnt_parmlen* contains the length of the mount parameter, and *mnt_parmoffset* is zero.

To assure that at least one entry, including the mount parameter, is returned, it is advisable to allocate space for at least two entries.

When the final entry has been placed in the buffer, w_getmntent() returns no entries.

Returned value

If successful, w_getmntent() returns the number of entries in the buffer.

If unsuccessful, w_getmntent() returns -1 and sets errno to one of the following values:

Error Code
Description

EINVAL
A parameter was specified incorrectly.

ERANGE
The result is too large to fit in the available buffer space.

Example

CELEBW49

```
/* CELEBW49

   This example uses w_getmntent() to retrieve information
   about currently mounted systems in the MNTE3 mapping format.

   The MNTE3 mapping format is only available on z/OS V1.10 or higher,
   target value of 0x410A0000.

 */
#define _OPEN_SYS
#include <sys/mntent.h>
#include <stdio.h>

main() {
  int entries, entry;
  struct {
    MNTE3H   header;
    W_MNTENT3 mount_table[10];
  } work_area;

  memset(&work_area, 0x00, sizeof(work_area));

  /* 'header' initialization to specify MNTE3 mapping format */
  memcpy(work_area.header.mnt3H_cbid, MNTE3H_ID, 4);
  work_area.header.mnt3H_cblen = sizeof(struct mnte3);
  work_area.header.mnt3H_bodylen = sizeof(struct w_mntent3);

  do {
    if ((entries = w_getmntent((char *) &work_area,
                               sizeof(work_area))) == -1)
      perror("w_getmntent() error");

    else for (entry=0; entry<entries; entry++) {
      printf("filesystem %s is mounted at %s\n",
             work_area.mount_table[entry].mnt3_fsname,
             work_area.mount_table[entry].mnt3_mountpoint);
      printf("  MNTE2 and MNTE3 common: reads done is %i, writes done is %i\n",
             work_area.mount_table[entry].mnt3_readct,
             work_area.mount_table[entry].mnt3_writect);
      printf("  MNTE3 specific: mount time in seconds is %i\n",
             work_area.mount_table[entry].mnt3_mntsec);
    }
  } while (entries > 0);
}
```

Output:

```
filesystem ZOS110.LPP.HFS is mounted at /usr/lpp
  MNTE2 and MNTE3 common: reads done is 0, writes done is 0
  MNTE3 specific: mount time in seconds is 5833
filesystem ZOS110.NLS.HFS is mounted at /usr/lib/nls
  MNTE2 and MNTE3 common: reads done is 0, writes done is 0
  MNTE3 specific: mount time in seconds is 5833
filesystem ZOS110.MAN.HFS is mounted at /usr/man
  MNTE2 and MNTE3 common: reads done is 0, writes done is 0
  MNTE3 specific: mount time in seconds is 5833
filesystem ZOS110.VAR.HFS is mounted at /SYSTEM/var
  MNTE2 and MNTE3 common: reads done is 0, writes done is 0
  MNTE3 specific: mount time in seconds is 5833
filesystem ZOS110.ETC.HFS is mounted at /SYSTEM/etc
  MNTE2 and MNTE3 common: reads done is 0, writes done is 0
  MNTE3 specific: mount time in seconds is 5833
filesystem ZOS110.ROOT.HFS is mounted at /
  MNTE2 and MNTE3 common: reads done is 0, writes done is 0
  MNTE3 specific: mount time in seconds is 5833
```

CELEBW48

```
/* CELEBW48

   This example uses w_getmntent() to retrieve information
   about currently mounted systems in the MNTE2 mapping format.

 */
#define _OPEN_SYS
#include <sys/mntent.h>
#include <stdio.h>

main() {
  int entries, entry;
  struct {
    MNTE2H   header;
    W_MNTENT2 mount_table[10];
  } work_area;
  memset(&work_area, 0x00, sizeof(work_area));

  /* 'header' initialization to specify MNTE2 mapping format */
  memcpy(work_area.header.mh2_hdr.mnt2h_cbid, MNTE2H_ID, 4);
  work_area.header.mh2_hdr.mnt2h_cblen = sizeof(struct mnte2);
  work_area.header.mh_bodylen = sizeof(struct w_mntent2);

  do {
    if ((entries = w_getmntent((char *) &work_area,
                               sizeof(work_area))) == -1)
      perror("w_getmntent() error");

    else for (entry=0; entry<entries; entry++) {
      printf("filesystem %s is mounted at %s\n",
             work_area.mount_table[entry].mnt2_fsname,
             work_area.mount_table[entry].mnt2_mountpoint);
      printf("  MNTE2 specific: reads done is %i, writes done is %i\n",
             work_area.mount_table[entry].mnt2_readct,
             work_area.mount_table[entry].mnt2_writect);
    }
  } while (entries > 0);
}
```

Output:

```
filesystem ZOS110.LPP.HFS is mounted at /usr/lpp
  MNTE2 specific: reads done is 0, writes done is 0
filesystem ZOS110.NLS.HFS is mounted at /usr/lib/nls
  MNTE2 specific: reads done is 0, writes done is 0
filesystem ZOS110.MAN.HFS is mounted at /usr/man
  MNTE2 specific: reads done is 0, writes done is 0
filesystem ZOS110.VAR.HFS is mounted at /SYSTEM/var
  MNTE2 specific: reads done is 0, writes done is 0
filesystem ZOS110.ETC.HFS is mounted at /SYSTEM/etc
  MNTE2 specific: reads done is 0, writes done is 0
filesystem ZOS110.ROOT.HFS is mounted at /
  MNTE2 specific: reads done is 0, writes done is 0
```

CELEBW32

```
/* CELEBW32

   This example uses w_getmntent() to retrieve information
   about currently mounted systems in the MNTE mapping format.

 */
#define _OPEN_SYS
#include <sys/mntent.h>
#include <stdio.h>

main() {
  int entries, entry;
  struct {
    struct w_mnth   header;
    struct w_mntent mount_table[10];
  } work_area;

  memset(&work_area, 0x00, sizeof(work_area));
  do {
    if ((entries = w_getmntent((char *) &work_area,
                               sizeof(work_area))) == -1)
      perror("w_getmntent() error");

    else for (entry=0; entry<entries; entry++) {
      printf("filesystem %s is mounted at %s\n",
             work_area.mount_table[entry].mnt_fsname,
            work_area.mount_table[entry].mnt_mountpoint);
    }
  } while (entries > 0);
}
```

Output:

```
filesystem ZOS110.LPP.HFS is mounted at /usr/lpp
filesystem ZOS110.NLS.HFS is mounted at /usr/lib/nls
filesystem ZOS110.MAN.HFS is mounted at /usr/man
filesystem ZOS110.VAR.HFS is mounted at /SYSTEM/var
filesystem ZOS110.ETC.HFS is mounted at /SYSTEM/etc
filesystem ZOS110.ROOT.HFS is mounted at /
```

Related information

- "sys/mntent.h" on page 73
- "statvfs() — Get file system information" on page 1724
- "w_statfs() — Get the file system status" on page 2095

w_getpsent() — Get process data

Standards

Standards / Extensions	C or C++	Dependencies
z/OS UNIX	both	

Format

```
#include <sys/ps.h>

int w_getpsent(int token, W_PSPROC *buffptr, size_t length);
```

General description

Provides information about the status of any process that the calling process has access to.

token A relative number that identifies the relative position of a process in the system. Zero represents the first process in the system. On the first call to w_getpsent(), pass the token 0; the function then returns the token that identifies the next process to which the caller has access. Use that token on the next call.

buffptr The address of the buffer where the data is to be stored.

length The length of the buffer.

The data returned is described in the ps.h header file. See Table 61 for the format of the structure stored in the buffer.

Table 61. Variables Stored in Structure Returned by w_getpsent()

Variable Type	Variable Name	General Description
unsigned int	ps_state	Process state
pid_t	ps_pid	Process ID
pid_t	ps_ppid	Parent ID
pid_t	ps_sid	Session ID (leader)
pid_t	ps_pgpid	Process group ID

Table 61. Variables Stored in Structure Returned by w_getpsent() (continued)

Variable Type	Variable Name	General Description
pid_t	ps_fgpid	Foreground process group ID
uid_t	ps_euid	Effective user ID
uid_t	ps_ruid	Real user ID
uid_t	ps_suid	Saved set user ID
gid_t	ps_egid	Effective group ID
gid_t	ps_rgid	Real group ID
gid_t	ps_sgid	Saved set group ID
long	ps_size	Total size
time_t	ps_starttime	Starting time
clock_t	ps_usertime	User CPU time
clock_t	ps_systime	System CPU time
int	ps_conttylen	Length of ConTTY
char	*ps_conttyptr	Controlling terminal
int	ps_pathlen	Length of *arg0*
char	*ps_pathptr	File name
int	ps_cmdlen	Length of command
char	*ps_cmdptr	Command and arguments

Note: The ps_cmdlen and ps_cmdptr elements of W_PSPROC identify the length and location where w_getpsent() is to return the command and arguments used to start the process. As of z/OS V1R11, the maximum length in bytes, including the null terminator, that can be returned is defined by PS_CMDBLEN_LONG. The actual maximum length of the command and arguments returned is PS_CMDBLEN_LONG minus the length of the path. Prior to z/OS V1R11, the maximum length in bytes, including the null terminator, that can be returned is defined by PS_CMDBLEN. The actual maximum length of the command and arguments returned can be less than PS_CMDBLEN and depends on the number of arguments. The system will truncate the command and arguments as needed to allow for the null terminator. Both PS_CMDBLEN_LONG and PS_CMDBLEN are defined in <sys/ps.h>.

Usage notes

1. ps_usertime reports the user's CPU time consumed for the address space the process is running within. When only one process is running in the address space, this CPU time represents the accumulated user CPU time for that process. When more than one process is running in an address space, the information returned is actually the accumulated CPU time consumed by the address space. It is the sum of the CPU time used by all of the work running in that address space not including the system time.
2. ps_systime reports the system's CPU time consumed for the address space the process is running within. When only one process is running in the address space, this time represents the accumulated system CPU time for that process. However, when more than one process is running in an address space, the information returned is actually the accumulated system CPU time consumed by all of the work running in the address space.

Returned value

If successful, w_getpsent() returns the process token for the next process for which the caller has access. For the last active process to which the user has access, w_getpsent() returns 0, indicating there are no more processes to be accessed.

If unsuccessful, w_getpsent() returns -1 and sets errno to one of the following values:

Error Code
: **Description**

EINVAL
: An incorrect process token or NULL buffer address was specified.

E2BIG
: An insufficient buffer length was provided.

ESRCH
: No process can be accessed by the active user.

Example

CELEBW33

```
/* CELEBW33

   This example provides status information, using wgetpsent().

 */
#define _OPEN_SYS
#include <stdlib.h>
#include <stdio.h>
#include <sys/ps.h>
#include <sys/types.h>
#include <pwd.h>
#include <time.h>
#include <string.h>

int main(void) {
  int token;
  W_PSPROC buf;
  struct passwd *pw;

  token = 0;

  memset(&buf, 0x00, sizeof(buf));
  buf.ps_conttyptr = (char *) malloc(buf.ps_conttylen = PS_CONTTYBLEN);
  buf.ps_pathptr   = (char *) malloc(buf.ps_pathlen   = PS_PATHBLEN);
  buf.ps_cmdptr    = (char *) malloc(buf.ps_cmdlen    = PS_CMDBLEN_LONG);
  if ((buf.ps_conttyptr == NULL) ||
      (buf.ps_pathptr   == NULL) ||
      (buf.ps_cmdptr    == NULL))
    perror("buffer allocation error");

  else do {
    if ((token = w_getpsent(token, &buf, sizeof(buf))) == -1)
      perror("w_getpsent() error");
    else if (token > 0)
      if ((pw = getpwuid(buf.ps_ruid)) == NULL)
        perror("getpwuid() error");
      else printf("token %d: pid %10d, user %8s, started %s", token,
                  (int) buf.ps_pid, pw->pw_name,
                  ctime(&buf.ps_starttime));
  } while (token > 0);
return 0;
}
```

Output:

```
token 2: pid     131074, user   MVSUSR1, started Fri Jun 16 08:09:17 2006
token 3: pid      65539, user   MVSUSR1, started Fri Jun 16 08:09:41 2006
token 6: pid     589830, user   MVSUSR1, started Fri Jun 16 10:29:17 2006
token 7: pid     851975, user   MVSUSR1, started Fri Jun 16 10:30:04 2006
```

Related information

- “sys/ps.h” on page 74

w_ioctl(), __w_pioctl() — Control of devices

Standards

Standards / Extensions	C or C++	Dependencies
z/OS UNIX	both	OS/390 V2R8

General Format

```
#define _OPEN_SYS
#include <termios.h>

int w_ioctl(int fildes, int cmd, int arglen, void *arg);
int __w_pioctl(const char *pathname, int cmd, int arglen, void *arg);
```

ACL format:

```
#define _OPEN_SYS
#include <termios.h>
#include <sys>

int w_ioctl(int fildes, int cmd, int arglen, void arg);
int __w_pioctl(const char pathname, int cmd, int arglen, void arg);
```

General description

The w_ioctl() and __w_pioctl() functions are general entry points for device-specific commands. The specific actions specified by w_ioctl() and __w_pioctl() vary with the device, and they are defined by the device driver.

fildes A descriptor for an open character special file (used by w_ioctl).

pathname
The pathname of a file (used by __w_pioctl).

cmd The command to be passed to the device driver as an integer value.

Command
Description

SIOCGPARTNERINFO
Provides an interface for an application to retrieve security information about its partner. For more information, see *z/OS Communications Server: IP Programmer's Guide and Reference*.

arglen The length of the argument passed to the device driver.

arg The address of the buffer where the argument to be passed to the device driver is stored.

w_ioctl() and __w_pioctl() pass the *cmd*, *arglen*, and *arg* arguments to the device driver to be interpreted and processed. When w_ioctl() and __w_pioctl() complete successfully, the device driver returns *arglen* and *arg*, if appropriate.

Note: The __w_pioctl() function has a dependency on the level of the Enhanced ASCII Extensions. See “Enhanced ASCII support” on page 2113 for details.

Returned value

If successful, w_ioctl() returns 0.

If unsuccessful, w_ioctl() returns -1 and sets errno to one of the following values:

Error Code
Description

EACCES
Permission is denied.

EINVAL

One of the following situations occurred:

- An incorrect length was specified for *arglen*. The correct argument length range is 0 to 50,000.
- An invalid message queue was specified when command _IOCC_REGFILEINT was used.

ENAMETOOLONG
The length of the *pathname* argument exceeds **PATH_MAX**, or a *pathname* component is longer than **NAME_MAX** and {**_POSIX_NO_TRUNC**} is in effect for that file. For symbolic links, the length of the pathname string substituted for a symbolic link exceeds **PATH_MAX**. **PATH_MAX** and **NAME_MAX** values can be determined by using pathconf().

ENODEV
The device does not exist. The function is not supported by the device driver.

ENOENT
Either there is no file named *pathname* or *pathname* is an empty string.

ENOTDIR
A component of the *pathname* prefix is not a directory.

ENOTSUP

Operation not supported. The following occurred:

Command _IOCC_REGFILEINT was used and fildes (w_ioctl) or pathname (__w_pioctl) refers to a file residing in a R/W file system that is shared across a sysplex with zFS, or is accessed through an NFS Client. In these cases, the file is able to be changed on a remote system without the local system being aware. Because the program would not receive notification, this operation is rejected.

Use __errno2() to obtain more diagnostic information.

ENOTTY
An incorrect file descriptor was specified. *fildes* was not a character special file.

ACLs description

The w_ioctl() and __w_pioctl() functions are general entry points for SETFACL and GETFACL HFS commands. SETFACL is used to set information into an Access Control List. GETFACL is used to retrieve information from an Access Control List.

fildes A descriptor for an open character special file (used by w_ioctl).

pathname
The pathname of a file (used by __w_pioctl).

cmd The command to be passed to the device driver as an integer value, either SETFACL or GETFACL.

arglen The length of the user buffer passed to the HFS device driver as a value from 1 to 50,000 bytes. arglen is the combined size of the struct ACL_buf and the array of struct ACL_entrys.

arg arg specifies the user buffer which is mapped by struct ACL_buf followed immediately by an array of struct ACL_entrys. See *z/OS UNIX System Services Programming: Assembler Callable Services Reference* for more information about ACL_buf and the ACL_entrys.

w_ioctl() and __w_pioctl() pass the *cmd*, *arglen*, and *arg* arguments to the device driver to be interpreted and processed. When w_ioctl() and __w_pioctl() complete successfully, the device driver returns *arglen* and *arg*, if appropriate.

ACLs returned value

If successful, w_ioctl() returns 0.

If unsuccessful, w_ioctl() returns -1 and sets errno to one of the following values:

Error Code
Description

EBADF
The *fildes* parameter is not a valid file descriptor.

EINVAL
The request is invalid or not supported.

EMVSPARM
Incorrect parameters were passed to the service.

ENODEV
The device is incorrect. The function is not supported by the device driver.

Example

CELEBW34

```
/* CELEBW34

   This example shows a general entry point for device-specific commands.

 */
#include <termios.h>
#include <stdio.h>

main() {
  char buf[256];
  int  ret;

  memset(buf, 0x00, sizeof(buf));
  if ((ret = w_ioctl(0, 1, sizeof(buf), buf)) != 0)
    perror("w_ioctl() error");
  else
    printf("w_iotctl() returned '%s'\n", buf);
}
```

Output:

```
w_ioctl() error: Invalid argument
```

Note: w_ioctl() is dependent upon the file system device driver.

Related information

- “termios.h” on page 78
- “ioctl() — Control device” on page 869
- “tcdrain() — Wait until output has been transmitted” on page 1823
- “tcflow() — Suspend or resume data flow on a terminal” on page 1825
- “tcflush() — Flush input or output on a terminal” on page 1828
- “tcgetattr() — Get the attributes for a terminal” on page 1830
- “tcgetpgrp() — Get the foreground process group ID” on page 1835
- “tcsendbreak() — Send a break condition to a terminal” on page 1842
- “tcsetattr() — Set the attributes for a terminal” on page 1843
- “tcsetpgrp() — Set the foreground process group ID” on page 1858

wmemchr() — Locate wide character

Standards

Standards / Extensions	C or C++	Dependencies
ISO C Amendment C99 Single UNIX Specification, Version 3	both	

Format

Non-XPG4:

```
#include <wchar.h>

wchar_t *wmemchr(const wchar_t *s, wchar_t c, size_t n);
```

XPG4:

```
#define _XOPEN_SOURCE
#define _MSE_PROTOS
#include <wchar.h>

wchar_t *wmemchr(const wchar_t *s, wchar_t c, size_t n);
```

General description

Locates the first occurrence of the wide character *c* in the initial *n* wide chars of the object pointed to by *s*.

If *n* has the value 0, wmemchr() finds no occurrence of *c* and returns a NULL pointer.

Note: The function is now available as a built-in function under ARCH(7) when LP64 is not used.

Special behavior for XPG4: If you define any feature test macro specifying XPG4 behavior before the statement in your program source file to include the wchar header, then you must also define the _MSE_PROTOS feature test macro to make the declaration of the wmemchr() function in the wchar header available when you compile your program. See Table 2 on page 4 for a list of XPG4 and other feature test macros.

Returned value

If successful, wmemchr() returns a pointer to the first occurrence of the wide character *c* in object *s*.

If the wide character *c* does not occur within the first *n* wide characters of *s*, wmemchr() returns the NULL pointer.

Example

```
#include <stdio>
#include <wchar>

  main()
  {
   wchar_t  *in = L"1234ABCD";
   wchar_t  *ptr;
   wchar_t  fnd = L'A';

   printf("\nEXPECTED: ABCD");
   ptr = wmemchr(in, L'A', 6);
   if (ptr == NULL)
        printf("\n** ERROR ** ptr is NULL, char L'A' not found\n");
     else
        printf("\nRECEIVED: %ls \n",ptr);

  }
```

Related information

- "wchar.h" on page 85
- "wmemcmp() — Compare wide character"
- "wmemcpy() — Copy wide character" on page 2078
- "wmemmove() — Move wide character" on page 2079
- "wmemset() — Set wide character" on page 2080

wmemcmp() — Compare wide character

Standards

Standards / Extensions	C or C++	Dependencies
ISO C Amendment C99 Single UNIX Specification, Version 3	both	

Format

Non-XPG4:

```
#include <wchar.h>

int wmemcmp(const wchar_t * __restrict__s1,
          const wchar_t * __restrict__s2, size_t n);
```

XPG4:

```
#define _XOPEN_SOURCE
#define _MSE_PROTOS
#include <wchar.h>

int wmemcmp(const wchar_t *s1, const wchar_t *s2, size_t n);
```

General description

Compares the first *n* wide chars of the object pointed to by *s1* to the first *n* wide chars of the object pointed to by *s2*.

If *n* has the value 0, wmemcmp() returns 0.

Note: The function is now available as a built-in function under ARCH(7) when LP64 is not used.

Special behavior for XPG4: If you define any feature test macro specifying XPG4 behavior before the statement in your program source file to include the wchar header, then you must also define the _MSE_PROTOS feature test macro to make the declaration of the wmemcmp function in the wchar header available when you compile your program. Please see Table 2 on page 4 for a list of XPG4 and other feature test macros.

Returned value

wmemcmp() returns an integer that is:

< 0 If *s1* is less than *s2*.

= 0 If *s1* is equal to *s2*.

> 0 If *s1* is greater than *s2*.

Example

```
#include <wchar.h>
#include <stdio.h>

main()
  {
   int      ptr;
   wchar_t  *in  = L"12345678";
   wchar_t  *out = L"12AAAAAB";

   printf("\nGREATER is the expected result");
   ptr = wmemcmp(in, out, 3);
   if (ptr == 0)
        printf("\nArrays are EQUAL %ls %ls \n",in, out);
     else
       {
        if (ptr > 0)
            printf("\nArray %ls GREATER than %ls \n",in, out);
          else
            printf("\nArray %ls LESS than %ls \n",in, out);
       }

  }
```

Related information

- "wchar.h" on page 85
- "wmemchr() — Locate wide character" on page 2075
- "wmemcpy() — Copy wide character" on page 2078
- "wmemmove() — Move wide character" on page 2079

- “wmemset() — Set wide character” on page 2080

wmemcpy() — Copy wide character

Standards

Standards / Extensions	C or C++	Dependencies
ISO C Amendment C99 Single UNIX Specification, Version 3	both	

Format

Non-XPG4:

```
#include <wchar.h>

wchar_t *wmemcpy(wchar_t * __restrict__s1,
                 const wchar_t * __restrict__s2, size_t n);
```

XPG4:

```
#define _XOPEN_SOURCE
#define _MSE_PROTOS
#include <wchar.h>

wchar_t *wmemcpy(wchar_t *s1, const wchar_t *s2, size_t n);
```

General description

Copies *n* wide chars of the object pointed to by *s2* to the object pointed to by *s1*.

Result of the copy is unpredictable if *s1* and *s2* overlap. If *n* has the value 0, wmemcpy copies zero wide characters.

Note: The function is now available as a built-in function under ARCH(7) when LP64 is not used.

Special behavior for XPG4: If you define any feature test macro specifying XPG4 behavior before the statement in your program source file to include the wchar header, then you must also define the _MSE_PROTOS feature test macro to make the declaration of the wmemcpy function in the wchar header available when you compile your program. Please see Table 2 on page 4 for a list of XPG4 and other feature test macros.

Returned value

wmemcpy() returns the value of *s1*.

Example

```
#include <wchar.h>
#include <stdio.h>


main()
 {
  wchar_t  *in    = L"12345678";
  wchar_t  *out   = L"ABCDEFGH";
  wchar_t  *ptr;
  printf("-nExpected result: First 4 chars of in change");
  printf(" and are the same as first 4 chars of out");
  ptr = wmemcpy(in, out, 4);
  if (ptr == in)
       printf("-nArray in %ls array out %ls -n",in, out);
    else
      {
       printf("-n*** ERROR ***");
       printf(" returned pointer wrong");
      }
 }
```

Related information

- “wchar.h” on page 85
- “wmemchr() — Locate wide character” on page 2075
- “wmemcmp() — Compare wide character” on page 2076
- “wmemmove() — Move wide character”
- “wmemset() — Set wide character” on page 2080

wmemmove() — Move wide character

Standards

Standards / Extensions	C or C++	Dependencies
ISO C Amendment C99 Single UNIX Specification, Version 3	both	

Format

Non-XPG4:

```
#include <wchar.h>

wchar_t *wmemmove(wchar_t *s1, const wchar_t *s2, size_t n);
```

XPG4:

```
#define _XOPEN_SOURCE
#define _MSE_PROTOS
#include <wchar.h>

wchar_t *wmemmove(wchar_t *s1, const wchar_t *s2, size_t n);
```

General description

Copies *n* wide chars of the object pointed to by *s2* to the object pointed to by *s1*. Copying takes place as if the *n* wide characters from *s2* are first copied into a temporary array of n wide characters that does not overlay the objects pointed to by *s1* and *s2*, and then copied from the temporary array into the object pointed to by *s1*.

If *n* has the value 0, wmemmove() copies zero wide characters.

Special behavior for XPG4: If you define any feature test macro specifying XPG4 behavior before the statement in your program source file to include the wchar header, then you must also define the _MSE_PROTOS feature test macro to make the declaration of the wmemmove() function in the wchar header available when you compile your program. Please see Table 2 on page 4 for a list of XPG4 and other feature test macros.

Returned value

wmemmove() returns the value of *s1*.

Example

```
#include <wchar.h>
#include <stdio.h>


main()
  {
   wchar_t  *in    = L"12345678";
   wchar_t  *out   = L"ABCDEFGH";

   wchar_t  *ptr;

   printf("\nExpected result: First 4 chars of in and out the same");
   ptr = wmemmove(in, out, 4);
   if (ptr == in)
        printf("\nArray in %ls array out %ls \n",in, out);
     else
       {
        printf("\n*** ERROR ***");
        printf("  Returned pointer not correct.\n");
       }
  }
```

Related information

- "wchar.h" on page 85
- "wmemchr() — Locate wide character" on page 2075
- "wmemcmp() — Compare wide character" on page 2076
- "wmemcpy() — Copy wide character" on page 2078
- "wmemset() — Set wide character"

wmemset() — Set wide character

Standards

Standards / Extensions	C or C++	Dependencies
ISO C Amendment C99 Single UNIX Specification, Version 3	both	

Format

Non-XPG4:

```
#include <wchar.h>

wchar_t *wmemset(wchar_t *s, wchar_t c, size_t n);
```

XPG4:

```
#define _XOPEN_SOURCE
#define _MSE_PROTOS
#include <wchar.h>

wchar_t *wmemset(wchar_t *s, wchar_t c, size_t n);
```

General description

Copies the value of *c* into the first *n* wide chars of the object pointed to by *s*.

If *n* has the value 0, wmemset() copies zero wide characters.

Note: The function is now available as a built-in function under ARCH(7) when LP64 is not used.

Special behavior for XPG4: If you define any feature test macro specifying XPG4 behavior before the statement in your program source file to include the wchar header, then you must also define the _MSE_PROTOS feature test macro to make the declaration of the wmemset() function in the wchar header available when you compile your program. Please see Table 2 on page 4 for a list of XPG4 and other feature test macros.

Returned value

wmemset() returns the value of *s*.

Example

```
#include <wchar.h>
#include <stdio.h>


  void main()
  {
   wchar_t  *in = L"1234ABCD";
   wchar_t  *ptr;

   printf("\nEXPECTED: AAAAAACD");
   ptr = wmemset(in, L'A', 6);
   if (ptr == in)
        printf("\nResults returned - %ls \n",ptr);
     else
       {
        printf("\n** ERROR ** wrong pointer returned\n");
       }

  }
```

Related information

- “wchar.h” on page 85
- “wmemchr() — Locate wide character” on page 2075
- “wmemcmp() — Compare wide character” on page 2076
- “wmemcpy() — Copy wide character” on page 2078
- “wmemmove() — Move wide character” on page 2079

wordexp() — Perform shell word expansions

Standards

Standards / Extensions	C or C++	Dependencies
XPG4 XPG4.2 Single UNIX Specification, Version 3	both	POSIX(ON)

Format

```
#define _XOPEN_SOURCE
#include <wordexp.h>

int wordexp(const char *__restrict__ words,
            wordexp_t *__restrict__ pwordexp, int flags);

void wordfree(wordexp_t *pwordexp);
```

General description

The wordexp() function performs word expansions as described in *X/Open CAE Specification, Commands and Utilities, Issue 4, Version 2* (**XCU**) **Section 2.6 , Word Expansions**, subject to quoting as in the **XCU** specification, **Section 2.2 , Quoting**, and places the list of expanded words into the structure pointed to by *pwordexp*.

The *words* argument is a pointer to a string containing one or more words to be expanded. The expansions will be the same as would be performed by the shell described by the **XCU** specification if *words* were the part of a command line representing the arguments to a utility. Therefore, *words* must not contain an unquoted <newline> or any of the unquoted special shell characters:

```
   |    &    ;    <    >
```

except in the context of command substitution as specified in the **XCU** specification, **Section 2.6.3 , Command Substitution**. It also must not contain unquoted parentheses or braces, except in the context of command or variable substitution.

The structure type **wordexp_t** is defined in the header **<wordexp.h>** and includes the following members:

Member Type	Member Name	Description
size_t	we_wordc	Count of words matched by *words*.
char **	we_wordv	Pointer to list of expanded words.
size_t	we_offs	Slots to reserve at the beginning of *pwordexp*->we_wordv.

The wordexp() function stores the number of generated words into *pwordexp*->we_wordc and a pointer to a list of pointers to words in *pwordexp*->we_wordv. Each individual field created during field splitting (see the **XCU** specification, **Section 2.6.5 , Field Splitting**) or pathname expansion (see the **XCU** specification, **Section 2.6.6 , Pathname Expansion**) is a separate word in the *pwordexp*->we_wordv list. The words are in order as described in the **XCU** specification, **Section 2.6 , Word Expansions**. The first pointer after the last word pointer will be a NULL pointer. The expansion of special parameters described in the **XCU** specification, **Section 2.5.2 , Special Parameters** is unspecified.

It is the caller's responsibility to allocate the storage pointed to by *pwordexp*. The wordexp() function allocates the other space as needed, including memory pointed to by *pwordexp*->we_wordv. The wordfree() function frees any memory associated with *pwordexp* from a previous call to wordexp().

The *flags* argument is used to control the behavior of wordexp(). The value of *flags* is the bitwise inclusive-OR or zero or more of the following constants, which are defined in <wordexp.h>:

WRDE_APPEND
Append words generated to the ones from a previous call to wordexp().

WRDE_DOOFFS
Make use of *pwordexp*->we_offs. If this flag is set, *pwordexp*->we_offs is used to specify how many NULL pointers to add to the beginning of *pwordexp*->we_wordv. In other words, *pwordexp*->we_wordv will point to *pwordexp*->we_offs NULL pointers followed by *pwordexp*->we_wordc word pointers, followed by a NULL pointer.

WRDE_NOCMD
Fail if command substitution, as specified in the **XCU** specification, **Section 2.6.3 , Command Substitution**, is requested.

WRDE_REUSE
The *pwordexp* argument was passed to a previous successful call to wordexp(), and has not been passed to wordfree(). The result will be the same as if the application had called wordfree() and then called wordexp() without WRDE_REUSE.

WRDE_SHOWERR
Do not redirect stderr to /dev/null.

WRDE_UNDEF
Report error on an attempt to expand an undefined shell variable.

The WRDE_APPEND flag can be used to append a new set of words to those generated by a previous call to wordexp(). The following rules apply when two or more calls to wordexp() are made with the same value of *pwordexp* and without intervening calls to wordfree():

1. The first such call must not set WRDE_APPEND. All subsequent calls must set it.
2. All of the calls must set WRDE_DOOFFS, or all must not set it.
3. After the second and each subsequent call, *pwordexp*->we_wordv will point to a list containing the following:
   a. zero or more NULL pointers, as specified by WRDE_DOOFFS and *pwordexp*->we_offs
   b. pointers to the words that were in the *pwordexp*->we_wordv list before the call, in the same order as before
   c. pointers to the new words generated by the latest call, in the specified order
4. The count returned in *pwordexp*->we_wordc will be the total number of words from all of the calls.
5. The application can change any of the fields after a call to wordexp(), but if it does, it must reset them to the original value before a subsequent call, using the same *wordexp* value, to wordfree() or wordexp() with the WRDE_APPEND or WRDE_REUSE flag.

If *words* contains an unquoted

```
<newline> | & ; < > ( ) { }
```

in an inappropriate context, wordexp() will fail, and the number of expanded words will be 0.

Unless WRDE_SHOWERR is set in *flags*, wordexp() will redirect stderr to /dev/null for any utilities executed as a result of command substitution while expanding *words*. If WRDE_SHOWERR is set, wordexp() may write messages to stderr if syntax errors are detected while expanding *words*.

If WRDE_DOOFFS is set, then *pwordexp*->we_offs must have the same value for each wordexp() call and wordfree() call using a given *pwordexp*.

The following constants are defined in <wordexp.h> as error return values:

WRDE_BADCHAR
One of the unquoted characters:

```
<newline> | & ; < > ( ) { }
```

appears in *words* in an inappropriate context.

WRDE_BADVAL
Reference to undefined shell variable when WRDE_UNDEF is set in *flags*.

WRDE_CMDSUB
Command substitution requested when WRDE_NOCMD is set in *flags*.

WRDE_NOSPACE
Attempt to allocate memory failed.

WRDE_SYNTAX
Shell syntax error, such as unbalanced parentheses or unterminated string.

WRDE_NOSYS
POSIX shell not available.

WRDE_EPOPEN
wordexp() was invoked in an environment that does not supported a multi-threaded fork, or wordexp() was invoked from a multithreaded process in TSO or MVS batch.

WRDE_INTRUPT
wordexp() was interrupted by a signal, such as an alarm. In this event, the caller may re-issue wordexp().

For further information about the cause of a failure, please refer to __errno2() and errno in addition to the above error return values.

Returned value

If successful, wordexp() returns 0.

If unsuccessful, wordexp() returns a nonzero value, as defined in <wordexp.h> and described above, to indicate an error. If wordexp() returns the value WRDE_NOSPACE, then *pwordexp*->**we_wordc**, and *pwordexp*->**we_wordv** will be updated to reflect any words that were successfully expanded. In other cases, they will not be modified.

Related information

- “wordexp.h” on page 87

wordfree() — Free shell word expansion memory

Standards

Standards / Extensions	C or C++	Dependencies
XPG4 XPG4.2 Single UNIX Specification, Version 3	both	

Format

```
#define _XOPEN_SOURCE
#include <wordexp.h>

int wordexp(const char *words, wordexp_t *pwordexp, int flags);

void wordfree(wordexp_t *pwordexp);
```

General description

The wordfree() function frees any memory associated with *pwordexp* from a previous call to wordexp(). Please refer to the description of wordexp() for the rules governing use of wordexp() and wordfree().

Returned value

wordfree() returns no values.

Related information

- "wordexp.h" on page 87

__w_pioctl() — Control of devices

The information for this function is included in "w_ioctl(), __w_pioctl() — Control of devices" on page 2072.

wprintf() — Format and write wide characters

The information for this function is included in "fwprintf(), swprintf(), wprintf() — Format and write wide characters" on page 672.

write() — Write data on a file or socket

Standards

Standards / Extensions	C or C++	Dependencies
POSIX.1 XPG4 XPG4.2 Single UNIX Specification, Version 3	both	

Format

```
#define _POSIX_SOURCE
#include <unistd.h>

ssize_t write(int fs, const void *buf, size_t N);
```

X/Open:

```
#define _XOPEN_SOURCE_EXTENDED 1
#include <unistd.h>

ssize_t write(int fs, const void *buf, ssize_t N);
```

Berkeley sockets:

```
#define _OE_SOCKETS
#include <unistd.h>

ssize_t write(int fs, const void *buf, ssize_t N);
```

General description

Writes *N* bytes from *buf* to the file or socket associated with *fs*. *N* should not be greater than INT_MAX (defined in the limits.h header file). If *N* is zero, write() simply returns 0 without attempting any other action.

If *fs* refers to a regular file or any other type of file on which a process can seek, write() begins writing at the file offset associated with *fs*. A successful write() increments the file offset by the number of bytes written. If the incremented file offset is greater than the previous length of the file, the length of the file is set to the new file offset.

If *fs* refers to a file on which a process cannot seek, write() begins writing at the current position. There is no file offset associated with such a file.

If O_APPEND (defined in the fcntl.h header file) is set for the file, write() sets the file offset to the end of the file before writing the output.

If there is not enough room to write the requested number of bytes (for example, because there is not enough room on the disk), write() outputs as many bytes as the remaining space can hold.

If write() is interrupted by a signal, the effect is one of the following:
- If write() has not written any data yet, it returns -1 and sets errno to EINTR.
- If write() has successfully written some data, it returns the number of bytes it wrote before it was interrupted.

Write operations on pipes or FIFO special files are handled in the same way a write operation on a regular file, with the following exceptions:
- A pipe has no associated file offset, so every write appends to the end of the pipe.
- If *N* is less than or equal to PIPE_BUF, the output is not interleaved with data written by other processes that are writing to the same pipe. If *N* is greater than PIPE_BUF bytes, the output can be interleaved with other data (regardless of the setting of O_NONBLOCK, which is defined in the fcntl.h header file). A write to a pipe never returns with errno set to EINTR if it has transferred any data.
- If O_NONBLOCK (defined in the fcntl.h header file) is not set, write() may block process execution until normal completion.
- If O_NONBLOCK is set, write() does not block process execution. If *N* is less than or equal to PIPE_BUF, write() succeeds completely and returns the value of *N*, or else it writes nothing, sets errno to EAGAIN, and returns -1. If *N* is greater than PIPE_BUF, write() writes as many bytes as it can and returns this number as its result, or else it writes nothing, sets errno to EAGAIN, and returns -1.

With other files that support nonblocking writes and cannot accept data immediately, the effect is one of the following:
- If O_NONBLOCK is not set, write() blocks until the data can be written.
- If O_NONBLOCK is set, write() does not block the process. If some data can be written without blocking the process, write() writes what it can and returns the number of bytes written. Otherwise, it sets errno to EAGAIN and returns -1.

write() causes the signal SIGTTOU to be sent if all of these conditions are true:

- The process is attempting to write to its controlling terminal and TOSTOP is set as a terminal attribute.
- The process is running in a background process group and the SIGTTOU signal is not blocked or ignored.
- The process is not an orphan.

A successful write() updates the change and modification times for the file.

If *fs* refers to a socket, write() is equivalent to send() with no flags set.

Behavior for sockets: The write() function writes data from a buffer on a socket with descriptor *fs*. The socket must be a connected socket. This call writes up to N bytes of data.

Parameter	Description
fs	The file or socket descriptor.
buf	The pointer to the buffer holding the data to be written.
N	The length in bytes of the buffer pointed to by the *buf* parameter.

If there is not enough available buffer space to hold the socket data to be transmitted, and the socket is in blocking mode, write() blocks the caller until additional buffer space becomes available. If the socket is in nonblocking mode, write() returns -1 and sets the error code to EWOULDBLOCK. See "fcntl() — Control open file descriptors" on page 474 or "ioctl() — Control device" on page 869 for a description of how to set the nonblocking mode.

When the socket is not ready to accept data and the process is trying to write data to the socket:

- Unless FNDELAY or O_NONBLOCK is set, write() blocks until the socket is ready to accept data.
- If FNDELAY is set, write() returns 0.
- If O_NONBLOCK is set, write() does not block the process. If some data can be written without blocking the process, write() writes what it can and returns the number of bytes written. Otherwise, it sets the error code to EAGAIN and returns -1.

For datagram sockets, this call sends the entire datagram, provided that the datagram fits into the TCP/IP buffers. Stream sockets act like streams of information with no boundaries separating data. For example, if an application program wishes to send 1000 bytes, each call to this function can send 1 byte or 10 bytes or the entire 1000 bytes. Therefore, application programs using stream sockets should place this call in a loop, calling this function until all data has been sent.

Special behavior for C++ and sockets: To use this function with C++, you must use the _XOPEN_SOURCE_EXTENDED 1 feature test macro.

Large file support for z/OS UNIX files: Large z/OS UNIX files are supported automatically for AMODE 64 C/C++ applications. AMODE 31 C/C++ applications must be compiled with the option LANGLVL(LONGLONG) and define the _LARGE_FILES feature test macro before any headers are included to enable this function to operate on z/OS UNIX files that are larger than 2 GB in size. File size and offset fields are enlarged to 63 bits in width. Therefore, any other function operating on the file is required to define the _LARGE_FILES feature test macro as well.

Returned value

If successful, write() returns the number of bytes actually written, less than or equal to *N*.

A value of 0 or greater indicates the number of bytes sent. However, this does not assure that data delivery was complete. A connection can be dropped by a peer socket and a SIGPIPE signal generated at a later time if data delivery is not complete.

If unsuccessful, write() returns -1 and sets errno to one of the following values:

Error Code
  Description

EAGAIN
  Resources temporarily unavailable. Subsequent requests may complete normally.

EBADF
  fs is not a valid file or socket descriptor.

ECONNRESET
  A connection was forcibly closed by a peer.

EDESTADDRREQ
  The socket is not connection-oriented and no peer address is set.

EFAULT
  Using the *buf* and N parameters would result in an attempt to access storage outside the caller's address space.

EFBIG
  Writing to the output file would exceed the maximum file size supported by the implementation.

  An attempt was made to write a file that exceeds the system established maximum file size or the process's file size limit.

  The file is a regular file, nbyte is greater than 0 and the starting position is greater than or equal to the offset maximum established in the open file description associated with fields.

EINTR
  write() was interrupted by a signal before it had written any output.

EINVAL
  The request is invalid or not supported. The STREAM or multiplexer referenced by *fs* is linked (directly or indirectly) downstream from a multiplexer.

EIO The process is in a background process group and is attempting to write to its controlling terminal, but TOSTOP (defined in the termios.h header file) is set, the process is neither ignoring nor blocking SIGTTOU signals, and the process group of the process is orphaned. An I/O error occurred.

EMSGSIZE
The message was too big to be sent as a single datagram.

ENOBUFS
Buffer space is not available to send the message.

ENOSPC
There is no available space left on the output device.

ENOTCONN
The socket is not connected.

ENXIO
A hang-up occurred on the STREAM being written to.

EPIPE write() is trying to write to a pipe that is not open for reading by any other process. This error also generates a SIGPIPE signal. For a connected stream socket the connection to the peer socket has been lost.

ERANGE
The transfer request size was outside the range supported by the STREAMS file associated with *fs*.

EWOULDBLOCK
socket is in nonblocking mode and no data buffers are available or the SO_SNDTIMEO timeout value was reached before buffers became available.

A write to a STREAMS file may fail if an error message has been received at the STREAM head. In this case, errno is set to the value included in the error message.

Note: z/OS UNIX System Services does not supply any STREAMS devices or pseudodevices. It is impossible for write() to write any data on a STREAM. None of the STREAMS errnos will be visible to the invoker. See "open() — Open a file" on page 1152

Example

CELEBW35

```
/* CELEBW35

   This example writes a certain amount of bytes to a file, using write().

 */
#define _POSIX_SOURCE
#include <fcntl.h>
#include <sys/stat.h>
#include <sys/types.h>
#include <unistd.h>
#undef _POSIX_SOURCE
#include <stdlib.h>
#include <stdio.h>

#define mega_string_len 1000000

main() {
  char *mega_string;
  int  fd, ret;
  char fn[]="write.file";

  if ((mega_string = (char*) malloc(mega_string_len)) == NULL)
    perror("malloc() error");
  else if ((fd = creat(fn, S_IWUSR)) < 0)
```

```
    perror("creat() error");
  else {
    memset(mega_string, '0', mega_string_len);
    if ((ret = write(fd, mega_string, mega_string_len)) == -1)
      perror("write() error");
    else printf("write() wrote %d bytes\n", ret);
    close(fd);
    unlink(fn);
  }
}
```

Output:

```
write() wrote 1000000 bytes
```

Related information

- "fcntl.h" on page 27
- "termios.h" on page 78
- "unistd.h" on page 82
- "connect() — Connect a socket" on page 308
- "creat() — Create a new file or rewrite an existing one" on page 340
- "dup() — Duplicate an open file descriptor" on page 404
- "fcntl() — Control open file descriptors" on page 474
- "fwrite() — Write items" on page 676
- "getsockopt() — Get the options associated with a socket" on page 777
- "ioctl() — Control device" on page 869
- "lseek() — Change the offset of a file" on page 1020
- "open() — Open a file" on page 1152
- "pipe() — Create an unnamed pipe" on page 1179
- "pwrite() — Write data on a file or socket without file pointer change" on page 1361
- "read() — Read from a file or socket" on page 1376
- "readv() — Read data on a file or socket and store in a set of buffers" on page 1389
- "recv() — Receive data on a socket" on page 1398
- "recvfrom() — Receive messages on a socket" on page 1401
- "recvmsg() — Receive messages on a socket and store in an array of message headers" on page 1405
- "select(), pselect() — Monitor activity on files or sockets and message queues" on page 1470
- "selectex() — Monitor activity on files or sockets and message queues" on page 1479
- "send() — Send data on a socket" on page 1494
- "sendmsg() — Send messages on a socket" on page 1500
- "sendto() — Send data on a socket" on page 1506
- "setsockopt() — Set options associated with a socket" on page 1578
- "socket() — Create a socket" on page 1688
- "writev() — Write data on a file or socket from an array" on page 2092

__writedown() — Query or change the setting of the write-down privilege of an ACEE.

Standards

Standards / Extensions	C or C++	Dependencies
z/OS UNIX	both	z/OS V1R5

Format

```
#define _OPEN_SYS
#include <sys/stat.h>
int __writedown ( int writedownop, int writedownscope);
```

General description

The __writedown() function will enable callers to query or change the setting of the write-down privilege of an ACEE (access control environment element) at the address space level or task level. User's having write-down privilege can write data to a resource protected by a seclabel of lower authority then that of the seclabel represented in the address space level ACEE.

To activate the write-down privilege the userid in the target ACEE must be permitted to the the IRR.WRITEDOWN.BYUSER profile in the FACILITY class. The FACILITY class must be active and RACLISTed, and the SETROPTS MLS option must be active.

See *z/OS V1R5 Planning for Multilevel Security* for more details on the usage of this function.

writedownop
The operation to be performed

__WD_QUERY
Query the current setting of the write-down privelege

__WD_ACTIVATE
Activate the write-down privilege

__WD_INACTIVATE
Inactivate the write-down privilage

__WD_RESET
Reset the write-down privilage to the users original default value.

writedownscope
Scope of the write-down operation.

__WD_SCOPE_AS
Perform write-down operation on the address space level ACEE.

__WD_SCOPE_THD
Perform write-down operation on the task level ACEE.

Returned value

For the __writedown() activate, inactivate, and reset operations:

If successful, __writedown() returns 0.

For the __writedown() query operation:

If successful, __writedown() returns one of the following values indicating the state of the current setting of the writedown privelege.

__WD_IS_ACTIVE
The write-down privelege is active for the ACEE.

__WD_IS_INACTIVE
The write-down privelege is inactive for the ACEE.

For all __writedown() operations:

If unsuccessful, all __writedown() operations return -1 and sets errno to EINVAL.

Related information

writev() — Write data on a file or socket from an array

Standards

Standards / Extensions	C or C++	Dependencies
XPG4.2 Single UNIX Specification, Version 3	both	

Format

X/Open:

```
#define _XOPEN_SOURCE_EXTENDED 1
#include <sys/uio.h>

ssize_t writev(int fs, const struct iovec *iov, int iovcnt);
```

Berkeley sockets:

```
#define _OE_SOCKETS
#include <sys/uio.h>

int writev(int fs, struct iovec *iov, int iovcnt);
```

General description

The writev() function writes data to a file or socket with descriptor `fs` from a set of buffers. The data is gathered from the buffers specified by iov[0]...iov[iovcnt-1]. When the descriptor refers to a socket, it must be a connected socket.

Parameter
Description

fs The file or socket descriptor.

iov A pointer to an array of **iovec** buffers.

iovcnt The number of buffers pointed to by the *iov* parameter.

The **iovec** structure is defined in **uio.h** and contains the following fields:

Element
Description

iov_base
Pointer to the buffer.

iov_len Length of the buffer.

This call writes the sum of the *iov_len* bytes of data.

If there is not enough available buffer space to hold the socket data to be transmitted, and the socket is in blocking mode, writev() blocks the caller until additional buffer space becomes available. If the socket is in a nonblocking mode, writev() returns -1 and sets the error code to EWOULDBLOCK. See “fcntl() —

Control open file descriptors” on page 474 or “ioctl() — Control device” on page 869 for a description of how to set nonblocking mode.

When the socket is not ready to accept data and the process is trying to write data to the socket:

- Unless FNDELAY or O_NONBLOCK is set, writev() blocks until the socket is ready to accept data.
- If FNDELAY is set, writev() returns a 0.
- If O_NONBLOCK is set, writev() does not block the process. If some data can be written without blocking the process, writev() writes what it can and returns the number of bytes written. Otherwise, it sets the error code to EAGAIN and returns -1.

For datagram sockets, this call sends the entire datagram, provided that the datagram fits into the TCP/IP buffers. Stream sockets act like streams of information with no boundaries separating data. For example, if an application program wishes to send 1000 bytes, each call to this function can send 1 byte, or 10 bytes, or the entire 1000 bytes. Therefore, application programs using stream sockets should place this call in a loop, calling this function until all data has been sent.

Special behavior for C++: To use this function with C++, you must use the _XOPEN_SOURCE_EXTENDED 1 feature test macro.

Returned value

If successful, writev() returns the number of bytes written from the buffer.

A value of 0 or greater indicates the number of bytes sent, however, this does not assure that data delivery was complete. A connection can be dropped by a peer socket and a SIGPIPE signal generated at a later time if data delivery is not complete.

If unsuccessful, writev() returns -1 and sets errno to one of the following values:

Error Code
Description

EAGAIN
Resources temporarily unavailable. Subsequent requests may complete normally.

EBADF
fs is not a valid file or socket descriptor.

ECONNRESET
A connection was forcibly closed by a peer.

EDESTADDRREQ
The socket is not connection-oriented and no peer address is set.

EFAULT
Using the *iov* and *iovcnt* parameters would result in an attempt to access storage outside the caller's address space.

EINTR
A signal interrupted writev() before any data was transmitted.

EINVAL
An incorrect value for *iocvnt* was detected.

EMSGSIZE
The message was too big to be sent as a single datagram.

ENOBUFS
Buffer space is not available to send the message.

ENOTCONN
The socket is not connected.

EPIPE For a connected stream socket the connection to the peer socket has been lost. A SIGPIPE signal is sent to the calling process.

EPROTOTYPE
The protocol is the wrong type for this socket. A SIGPIPE signal is sent to the calling process.

EWOULDBLOCK
socket is in nonblocking mode and no data buffers are available or the SO_SNDTIMEO timeout value was reached before buffers became available.

Related information

- "sys/uio.h" on page 76
- "connect() — Connect a socket" on page 308
- "fcntl() — Control open file descriptors" on page 474
- "getsockopt() — Get the options associated with a socket" on page 777
- "ioctl() — Control device" on page 869
- "read() — Read from a file or socket" on page 1376
- "readv() — Read data on a file or socket and store in a set of buffers" on page 1389
- "recv() — Receive data on a socket" on page 1398
- "recvfrom() — Receive messages on a socket" on page 1401
- "recvmsg() — Receive messages on a socket and store in an array of message headers" on page 1405
- "select(), pselect() — Monitor activity on files or sockets and message queues" on page 1470
- "selectex() — Monitor activity on files or sockets and message queues" on page 1479
- "send() — Send data on a socket" on page 1494
- "sendmsg() — Send messages on a socket" on page 1500
- "sendto() — Send data on a socket" on page 1506
- "setsockopt() — Set options associated with a socket" on page 1578
- "socket() — Create a socket" on page 1688
- "write() — Write data on a file or socket" on page 2085

__wsinit() — Reinitialize writable static

Standards

Standards / Extensions	C or C++	Dependencies
C Library	both	

Format

```
#include <unistd.h>

int __wsinit(void (*func_ptr)());
```

General description

The __wsinit() function reinitializes the writable static area of a module that was loaded by the fetch() library call. *func_ptr* must be a valid fetch pointer returned by fetch() or fetchep(). If the module contains C++, __wsinit() first runs any C++ static destructors, then __wsinit() runs the static constructors that are present in the load module.

Program variables with the static or extern storage class and writable strings receive the initial value defined in the program, if any initial value was assigned. The C header files contain external variable declarations for those variables defined by the POSIX, XPG4 and XPG4.2 standards. If the fetched module contains these variables, __wsinit() reinitializes them as described in *z/OS XL C/C++ Programming Guide*.

Returned value

If successful, __wsinit() returns 0 .

If unsuccessful, __wsinit() returns -1 and sets errno to one of the following values:

Error Code
Description

EINVAL
func_ptr is not a valid fetch pointer.

Related information

- “unistd.h” on page 82
- “fetch() — Get a load module” on page 507

w_statfs() — Get the file system status

Standards

Standards / Extensions	C or C++	Dependencies
z/OS UNIX	both	

Format

```
#include <sys/statfs.h>

int w_statfs(const char *filesystem, struct w_statfs *statbuf, size_t length);
```

General description

Gets status about a specific file system.

filesystem
The name of the file system for which status is being retrieved. This file system name can be one of the following:

- The name specified in the FILESYSTEM parameter of the ROOT or MOUNT statements in the BPXPRMxx parmlib member.
- The name specified in a TSO/E MOUNT command.
- The name returned on a previous call to w_getmntent().

statbuf A buffer that the status information is put into. The status information is mapped by the sys/statfs.h header file.

int statfs_len
Length of *statfs*

int statfs_blksize
Block size

unsigned int statfs_total_space
Total space in block size units

unsigned int statfs_used_space
Allocated space in block size units

unsigned int statfs_free_space
Space available to unprivileged users in block size units

length The length of the buffer. The length of the buffer and the length of the structure are compared, and the shorter of the two is used to determine how much information to return in the buffer.

If the buffer length is zero, only the return value is returned. A process can use a length of zero to detect if a file system exists or not.

Special behavior for XPG4.2: w_statfs() is replaced by w_statvfs().

Returned value

If successful, w_statfs() returns the length of the data in the buffer.

If unsuccessful, w_statfs() returns -1 and sets errno to one of the following values:

Error Code
Description

EINVAL
A parameter was incorrectly specified.

Example

CELEBW36

```
/* CELEBW36 */
#define _OPEN_SYS
#include <sys/statfs.h>
#include <stdio.h>

main() {
  char fs[]="POSIX.ROOT.FS";
  struct w_statfs buf;

  if (w_statfs(fs, &buf, sizeof(buf)) == -1)
    perror("w_statfs() error");
  else {
    printf("each block in %s is %d bytes big\n", fs,
           buf.statfs_blksize);
    printf("there are %d blocks in use out of a total of %d\n",
           buf.statfs_used_space, buf.statfs_total_space);
    printf("in bytes, that's %.0f bytes used out of a total of %.0f\n",
           ((double)buf.statfs_used_space  * buf.statfs_blksize),
           ((double)buf.statfs_total_space * buf.statfs_blksize));
  }
}
```

Output:

```
each block in POSIX.ROOT.FS is 4096 bytes big
there are 2089 blocks in use out of a total of 2400
in bytes, that's 8556544 bytes used out of a total of 9830400
```

Related information

- “sys/statfs.h” on page 75
- “mount() — Make a file system available” on page 1092
- “w_getmntent() — Get information on mounted file systems” on page 2058

w_statvfs() — Get the file system status

Standards

Standards / Extensions	C or C++	Dependencies
z/OS UNIX	both	

Format

```
#define _OPEN_SOURCE 2
#include <sys/statvfs.h>

int w_statvfs(const char *filesystem, struct statvfs *buffer, size_t buflen);
```

General description

The w_statvfs() function gets status about a specific file system.

filesystem
The name of the file system for which status is being retrieved. This file system name can be one of the following:

- The name specified in the FILESYSTEM parameter of the ROOT or MOUNT statements in the BPXPRMxx parmlib member.
- The name specified in a TSO/E MOUNT command.
- The name returned on a previous call to w_getmntent().

buffer
A buffer that the status information is put into. The information is returned in a `statvfs` structure, as defined in the sys/statvfs.h header file. The elements of this structure are described in “statvfs() — Get file system information” on page 1724.

buflen
The length of the buffer. The length of the buffer and the length of the structure are compared, and the shorter of the two is used to determine how much information to return in the buffer.

If the buffer length is zero, only the return value is returned. A process can use a length of zero to detect if a file system exists or not.

Returned value

If successful, w_statvfs() returns the length of the data in the buffer.

If unsuccessful, w_statvfs() returns -1 and sets errno to one of the following values:

Error Code
Description

EINVAL
A parameter was specified incorrectly. For example, the file system name (*filesystem*) was not found.

Example

```
#define _OPEN_SOURCE 2
#include <sys/statvfs.h>
#include <stdio.h>

main() {
  char fs[]="POSIX.ROOT.FS";
  struct statvfs buf;

  if (w_statvfs(fs, &buf, sizeof(buf)) == -1)
    perror("w_statvfs() error");
  else {
    printf("each block in %s is %d bytes big\n", fs,
           buf.f_bsize);
    printf("there are %d blocks available out of a total of %d\n",
           buf.f_bavail, buf.f_blocks);
    printf("in bytes, that's %.0f bytes free out of a total of %.0f\n",
           ((double)buf.f_bavail * buf.f_bsize),
           ((double)buf.f_blocks * buf.f_bsize));
  }
}
```

Output:

```
each block in POSIX.ROOT.FS is 4096 bytes big
there are 2089 blocks available out of a total of 2400
in bytes, that's 8556544 bytes free out of a total of 9830400
```

Related information

- “sys/statvfs.h” on page 75
- “fstatvfs() — Get file system information” on page 653
- “mount() — Make a file system available” on page 1092
- “statvfs() — Get file system information” on page 1724
- “w_getmntent() — Get information on mounted file systems” on page 2058

y0(), y1(), yn() — Bessel functions of the second kind

Standards

Standards / Extensions	C or C++	Dependencies
SAA XPG4 XPG4.2 Single UNIX Specification, Version 3	both	

Format

```
#include <math.h>

double y0(double x);
double y1(double x);
double yn(int n, double x);
```

Compiler option: LANGLVL(SAA), LANGLVL(SAA2), or LANGLVL(EXTENDED)

General description

Bessel functions are solutions to certain types of differential equations.

The y0(), y1(), and yn() functions are Bessel functions of the *second kind*, for orders 0, 1, and *n*, respectively. The argument *x* must be positive. The argument *n* should be greater than or equal to zero. If *n* is less than zero, there will be a negative exponent in the result.

Note: This function works in both IEEE Binary Floating-Point and hexadecimal floating-point formats. See "IEEE binary floating-point" on page 94 for more information about IEEE Binary Floating-Point.

Returned value

If successful, the function returns the calculated value.

For y0(), y1(), or yn(), if *x* is negative, the function sets errno to EDOM and returns -HUGE_VAL.

For y0(), y1(), or yn(), if *x* causes overflow, the function sets errno to ERANGE and returns -HUGE_VAL.

Special behavior for IEEE: If *x* is negative, y0(), y1(), and yn() return the value NaNQ. If *x* is 0, y0(), y1(), and yn() return the value -HUGE_VAL. In all cases, errno remains unchanged.

Example

CELEBY01

```
/* CELEBY01

   This example computes y to be the order 0 Bessel function of the first
   kind for x and z to be the order 3 Bessel function of the second kind for x.

 */
#include <math.h>
#include <stdio.h>

int main(void)
{
    double x, y, z;
    x = 4.27;

    y = y0(x);       /* y = -0.3660 is the order 0 bessel */
                     /* function of the first kind for x  */
    z = yn(3,x);     /* z = -0.0875 is the order 3 bessel */
                     /* function of the second kind for x */
    printf("x = %f\n  y = %f\n  z = %f\n", x, y, z);
}
```

Related information

- "math.h" on page 44
- "erf(), erfc(), erff(), erfl(), erfcf(), erfcl() — Calculate error and complementary error functions" on page 428
- "gamma() — Calculate gamma function" on page 682
- "j0(), j1(), jn() — Bessel functions of the first kind" on page 930

Library functions for the system programming C (SPC) facilities

Restriction: The SPC facility is not supported in AMODE 64.

The library functions specific to the System Programming C (SPC) environment are described in *z/OS XL C/C++ Programming Guide*. These system programming functions are as follows:

- __xhotc()
- __xhotl()
- __xhott()
- __xhotu()
- __xregs()
- __xsacc()
- __xsrvc()
- __xusr()
- __xusr2()
- __24malc()
- __4kmalc()

Appendix A. XL C/C++ Macros

You can use the macros listed in Table 62 to write programs that use built-in services of the z/OS XL C/C++ product. The general purpose macros, labelled *General*, are predefined macros that are either ANSI-standard macros or extensions to the Systems Application Architecture (SAA) definition. The internal-use-only macros, labelled *Internal*, are developed by IBM to provide you with additional z/OS XL C/C++ functionality. Internal macros are defined in corresponding include files.

Table 62. XL C/C++ Macros

Macro	Include File	General or Internal
_A_	dynit.h	General
ABDAY_1	nl_langinfo.h	General
ABDAY_2	nl_langinfo.h	General
ABDAY_3	nl_langinfo.h	General
ABDAY_4	nl_langinfo.h	General
ABDAY_5	nl_langinfo.h	General
ABDAY_6	nl_langinfo.h	General
ABDAY_7	nl_langinfo.h	General
__abendcode	signal.h, stdlib.h	General
ABMON_1	nl_langinfo.h	General
ABMON_10	nl_langinfo.h	General
ABMON_11	nl_langinfo.h	General
ABMON_12	nl_langinfo.h	General
ABMON_2	nl_langinfo.h	General
ABMON_3	nl_langinfo.h	General
ABMON_4	nl_langinfo.h	General
ABMON_5	nl_langinfo.h	General
ABMON_6	nl_langinfo.h	General
ABMON_7	nl_langinfo.h	General
ABMON_8	nl_langinfo.h	General
ABMON_9	nl_langinfo.h	General
__abnd_i	signal.h	General
_ABS(x)	math.h	Internal
abs(x)	stdlib.h	General
acos(x)	math.h	General
alignas	stdalign.h	General
alignof	stdalign.h	General
__alignas_is_defined	stdalign.h	General
__alignof_is_defined	stdalign.h	General
AM_STR	nl_langinfo.h	General
__amrc	stdio.h	General
__amrc_i	stdio.h	Internal
__APPEND	stdio.h	General
asin(x)	math.h	General
__assert	assert.h	Internal
assert(expr)	assert.h	General
assert(ignore)	assert.h	General
atan(x)	math.h	General
atan2(x,y)	math.h	General
_B_	dynit.h	General

Table 62. XL C/C++ Macros (continued)

Macro	Include File	General or Internal
__BINARY	stdio.h	General
__BLOCKED	stdio.h	General
__BSAM_CLOSE	stdio.h	General
__BSAM_CLOSE_T	stdio.h	General
__BSAM_NOTE	stdio.h	General
__BSAM_OPEN	stdio.h	General
__BSAM_POINT	stdio.h	General
__BSAM_READ	stdio.h	General
__BSAM_WRITE	stdio.h	General
BUFSIZ	stdio.h	General
cdump(x)	ctest.h	General
__CELMSGF_WRITE	stdio.h	General
CHAR_BIT	limits.h	General
CHAR_MAX	limits.h	General
CHAR_MIN	limits.h	General
__cics	cics.h	Internal
__cics_CD	cics.h	Internal
CLK_TCK	time.h	General
CLOCKS_PER_SEC	time.h	General
__CLOSE	dynit.h	General
clrmemf(x)	stdio.h	General
__CMS_CLOSE	stdio.h	General
__CMS_OPEN	stdio.h	General
__CMS_READ	stdio.h	General
__CMS_STATE	stdio.h	General
__CMS_WRITE	stdio.h	General
__cntrl(c)	stdio.h	Internal
CODESET	nl_langinfo.h	General
__CONTIG	dynit.h	General
cos(x)	math.h	General
cosh(x)	math.h	General
CRNCYSTR	nl_langinfo.h	General
csnap(x)	ctest.h	General
ctdli	ims.h	General
ctest(x)	ctest.h	General
__ctest__	ctest.h	Internal
ctime(t)	time.h	General
ctrace(x)	ctest.h	General
__ctype	ctype.h	Internal
__ctype_i	ctype.h	Internal
__ctypec	ctype.h	Internal
__ctypec_i	ctype.h	Internal
__CURR	stdarg.h	Internal
__CURRENT	stdio.h	General
__CURRENT_LOWER	stdio.h	General
__cust_def	ctype.h	Internal
__CYL	dynit.h	General
_D_	dynit.h	General
D_FMT	nl_langinfo.h	General
D_T_FMT	nl_langinfo.h	General
DAY_1	nl_langinfo.h	General
DAY_2	nl_langinfo.h	General
DAY_3	nl_langinfo.h	General

Table 62. XL C/C++ Macros (continued)

Macro	Include File	General or Internal
DAY_4	nl_langinfo.h	General
DAY_5	nl_langinfo.h	General
DAY_6	nl_langinfo.h	General
DAY_7	nl_langinfo.h	General
DBL_DECIMAL_DIG	float.h	General
DBL_DIG	float.h	General
__dbl_eps	float.h	Internal
DBL_EPSILON	float.h	General
__dbl_flts_i	float.h	Internal
DBL_MANT_DIG	float.h	General
DBL_MAX	float.h	General
DBL_MAX_EXP	float.h	General
DBL_MAX_10_EXP	float.h	General
DBL_MIN	float.h	General
DBL_MIN_EXP	float.h	General
DBL_MIN_10_EXP	float.h	General
DBL_SUBNORM	float.h	General
DBL_TRUE_MIN	float.h	General
__DEF_CLASS	dynit.h	General
difftime(t1, t0)	time.h	General
__DISK	stdio.h	General
__DISP_CATLG	dynit.h	General
__DISP_DELETE	dynit.h	General
__DISP_KEEP	dynit.h	General
__DISP_MOD	dynit.h	General
__DISP_NEW	dynit.h	General
__DISP_OLD	dynit.h	General
__DISP_SHR	dynit.h	General
__DISP_UNCATLG	dynit.h	General
DOMAIN	math.h	General
__DSNT_BASIC,	dynit.h	General
__DSNT_EXTPREF	dynit.h	General
__DSNT_EXTREQ	dynit.h	General
__DSNT_HFS	dynit.h	General
__DSNT_LARGE	dynit.h	General
__DSNT_LIBRARY	dynit.h	General
__DSNT_PDS	dynit.h	General
__DSNT_PIPE	dynit.h	General
__DSORG_DA	dynit.h	General
__DSORG_DAU	dynit.h	General
__DSORG_GS	dynit.h	General
__DSORG_IS	dynit.h	General
__DSORG_ISU	dynit.h	General
__DSORG_PO	dynit.h	General
__DSORG_POU	dynit.h	General
__DSORG_PS	dynit.h	General
__DSORG_PSU	dynit.h	General
__DSORG_unknown	dynit.h	General
__DSORG_VSAM	dynit.h	General
__DUMMY	stdio.h	General
__DUMMY_DSN	dynit.h	General
dyninit(__dynp)	dynit.h	General
__dynit	dynit.h	Internal

Table 62. XL C/C++ Macros (continued)

Macro	Include File	General or Internal
EACTIVE	mtf.h	General
EAUTOALC	mtf.h	General
EBADLNKG	mtf.h	General
EDOM	errno.h	General
EENTRY	mtf.h	General
EINACTIVE	mtf.h	General
EMODFIND	mtf.h	General
EMODFMT	mtf.h	General
EMODREAD	mtf.h	General
ENAME2LNG	mtf.h	General
ENOMEM	mtf.h	General
EOF	stdio.h	General
ERANGE	errno.h	General
erf(x)	math.h	General
erfc(x)	math.h	General
errno	errno.h	General
__errno_a	errno.h	Internal
__errno_i	errno.h	Internal
__errnoflg	errno.h	Internal
__errnoh	errno.h	Internal
__ES	dynit.h	General
__ESDS	stdio.h	General
__ESDS_PATH	stdio.h	General
ESUBCALL	mtf.h	General
ETASKABND	mtf.h	General
ETASKFAIL	mtf.h	General
ETASKID	mtf.h	General
ETASKNUM	mtf.h	General
EWRONGOS	mtf.h	General
EXIT_FAILURE	stdlib.h	General
EXIT_SUCCESS	stdlib.h	General
exp(x)	math.h	General
_F_	dynit.h	General
fabs(x)	math.h	General
_FB_	dynit.h	General
FBS	dynit.h	General
fdelrec(x)	stdio.h	General
fetch(x)	stdlib.h	General
fetchep(x)	stdlib.h	General
__FILE	assert.h	Internal
FILENAME_MAX	stdio.h	General
fldata(x,y,z)	stdio.h	General
__floath	float.h	Internal
flocate(w,x,y,z)	stdio.h	General
FLT_DECIMAL_DIG	float.h	General
FLT_DIG	float.h	General
__flt_eps	float.h	Internal
__flt_eps_i	float.h	Internal
FLT_EPSILON	float.h	General
FLT_MANT_DIG	float.h	General
FLT_MAX	float.h	General
FLT_MAX_EXP	float.h	General
FLT_MAX_10_EXP	float.h	General

Table 62. XL C/C++ Macros (continued)

Macro	Include File	General or Internal
FLT_MIN	float.h	General
FLT_MIN_EXP	float.h	General
FLT_MIN_10_EXP	float.h	General
FLT_RADIX	float.h	General
FLT_ROUNDS	float.h	General
FLT_SUBNORM	float.h	General
FLT_TRUE_MIN	float.h	General
FOPEN_MAX	stdio.h	General
FSETLOCKING_BYCALLER	stdio_ext.h	General
FSETLOCKING_INTERNAL	stdio_ext.h	General
FSETLOCKING_QUERY	stdio_ext.h	General
fupdate(x,y,z)	stdio.h	General
gamma(x)	math.h	General
getc(p)	stdio.h	General
__getc(p)	stdio.h	Internal
getchar(c)	stdio.h	General
__gtab(x)	stdio.h	Internal
__gtca()	assert.h	Internal
__HFS	stdio.h	General
__HIPERSPACE	stdio.h	General
__HOLDQ	dynit.h	General
__HSP_CREATE	stdio.h	General
__HSP_DELETE	stdio.h	General
__HSP_EXTEND	stdio.h	General
__HSP_READ	stdio.h	General
__HSP_WRITE	stdio.h	General
HUGE	math.h	General
HUGE_VAL	math.h	General
iconv	iconv.h	General
__ims	ims.h	Internal
__imspcb_a	ims.h	Internal
INT_MAX	limits.h	General
INT_MIN	limits.h	General
__INTERCEPT_READ	stdio.h	General
__INTERCEPT_WRITE	stdio.h	General
__IO_DEVTYPE	stdio.h	General
__IO_INIT	stdio.h	General
__IO_RDJFCB	stdio.h	General
_IOFBF	stdio.h	General
_IOLBF	stdio.h	General
__ISALNUM	ctype. and wchar.h	Internal
isalnum(c)	ctype.h	General
__isalnum(c)	ctype.h	Internal
__ISALPHA	ctype. and wchar.h	Internal
isalpha(c)	ctype.h	General
__isalpha(c)	ctype.h	Internal
isblank	ctype.h	General
__ISBLANK	ctype. and wchar.h	Internal
iscics	cics.h	General
__ISCNTRL	ctype. and wchar.h	Internal
iscntrl(c)	ctype.h	General
__iscntrl(c)	ctype.h	Internal
__ISDIGIT	ctype. and wchar.h	Internal

Table 62. XL C/C++ Macros (continued)

Macro	Include File	General or Internal
isdigit(c)	ctype.h	General
__isdigit(c)	ctype.h	Internal
__ISGRAPH	ctype. and wchar.h	Internal
isgraph(c)	ctype.h	General
__isgraph(c)	ctype.h	Internal
islower(c)	ctype.h	General
__islower(c)	ctype.h	Internal
__ISPRINT	ctype. and wchar.h	Internal
isprint(c)	ctype.h	General
__isprint(c)	ctype.h	Internal
__ISPUNCT	ctype. and wchar.h	Internal
ispunct(c)	ctype.h	General
__ispunct(c)	ctype.h	Internal
__ISSPACE	ctype. and wchar.h	Internal
isspace(c)	ctype.h	General
__isspace(c)	ctype.h	Internal
__ISUPPER	ctype. and wchar.h	Internal
isupper(c)	ctype.h	General
__isupper(c)	ctype.h	Internal
iswalnum	wchar.h	General
iswalpha	wchar.h	General
iswcntrl	wchar.h	General
iswctype	wchar.h	General
iswdigit	wchar.h	General
iswgraph	wchar.h	General
iswlower	wchar.h	General
iswprint	wchar.h	General
iswpunct	wchar.h	General
iswspace	wchar.h	General
iswupper	wchar.h	General
iswxdigit	wchar.h	General
isxdigit(c)	ctype.h	General
__isxdigit(c)	ctype.h	Internal
__KEY_EQ	stdio.h	General
__KEY_EQ_BWD	stdio.h	General
__KEY_FIRST	stdio.h	General
__KEY_GE	stdio.h	General
__KEY_LAST	stdio.h	General
__KS	dynit.h	General
__KSDS	stdio.h	General
__KSDS_PATH	stdio.h	General
L_tmpnam	stdio.h	General
LC_ALL	locale.h	General
LC_C	locale.h	General
LC_C_FRANCE	locale.h	General
LC_C_GERMANY	locale.h	General
LC_C_ITALY	locale.h	General
LC_C_SPAIN	locale.h	General
LC_C_UK	locale.h	General
LC_C_USA	locale.h	General
LC_COLLATE	locale.h	General
LC_CTYPE	locale.h	General
LC_MONETARY	locale.h	General

Table 62. XL C/C++ Macros (continued)

Macro	Include File	General or Internal
LC_NUMERIC	locale.h	General
LC_SYNTAX	locale.h	General
LC_TIME	locale.h	General
LC_TOD	locale.h	General
LDBL_DECIMAL_DIG	float.h	General
LDBL_DIG	float.h	General
__ldbl_eps	float.h	Internal
LDBL_EPSILON	float.h	General
__ldbl_flts_i	float.h	Internal
LDBL_MANT_DIG	float.h	General
LDBL_MAX	float.h	General
LDBL_MAX_EXP	float.h	General
LDBL_MAX_10_EXP	float.h	General
LDBL_MIN	float.h	General
LDBL_MIN_EXP	float.h	General
LDBL_MIN_10_EXP	float.h	General
LDBL_SUBNORM	float.h	General
LDBL_TRUE_MIN	float.h	General
LEAWI_INCLUDED	leawi.h	Internal
__limits	limits.h	Internal
__locale	locale.h	Internal
log(x)	math.h	General
log10(x)	math.h	General
__LOWER	stdio.h	General
__LS	dynit.h	General
_M_	dynit.h	General
M_E	math.h	General
M_LN10	math.h	General
M_LN2E	math.h	General
M_LOG	math.h	General
M_LOG10E	math.h	General
M_PI	math.h	General
M_PI_2	math.h	General
M_PI_4	math.h	General
M_SQRT1_2	math.h	General
M_SQRT2	math.h	General
M_1_PI	math.h	General
M_2_PI	math.h	General
M_2_SQRTPI	math.h	General
M_2PI	math.h	General
__math	math.h	Internal
__max_flt	math.h	Internal
__max_flts_i	float.h	Internal
MAXTASK	mtf.h	General
MB_CUR_MAX	stdlib.h	General
MB_LEN_MAX	limits.h	General
memchr(x,y,z)	string.h	General
memcmp(x,y,z)	string.h	General
memcpy(x,y,z)	string.h	General
memset(x,y,z)	string.h	General
__min_flts	float.h	Internal
__min_flts_i	float.h	Internal
MON_1	nl_langinfo.h	General

Table 62. XL C/C++ Macros (continued)

Macro	Include File	General or Internal
MON_2	nl_langinfo.h	General
MON_3	nl_langinfo.h	General
MON_4	nl_langinfo.h	General
MON_5	nl_langinfo.h	General
MON_6	nl_langinfo.h	General
MON_7	nl_langinfo.h	General
MON_8	nl_langinfo.h	General
MON_9	nl_langinfo.h	General
MON_10	nl_langinfo.h	General
MON_11	nl_langinfo.h	General
MON_12	nl_langinfo.h	General
__MSGFILE	stdio.h	General
__MSGRTN	stdio.h	General
MTF_ALL	mtf.h	General
MTF_ANY	mtf.h	General
MTF_OK	mtf.h	General
__mtfh	mtf.h	Internal
__NEXT	stdarg.h	Internal
__nextword(base)	stdarg.h	Internal
__NL_NUM_ITEMS	nl_langinfo.h	Internal
NOEXPR	nl_langinfo.h	General
__NOTVSAM	stdio.h	General
NULL	stddef.h	General
offsetof(x,y)	stddef.h	General
OMIT_FC	leawi.h	General
__osplist	stdlib.h	General
__OTHER	stdio.h	General
OVERFLOW	math.h	General
__PATH_OAPPEND	dynit.h	General
__PATH_OCREAT	dynit.h	General
__PATH_OEXCL	dynit.h	General
__PATH_ONOCTTY	dynit.h	General
__PATH_ONONBLOCK	dynit.h	General
__PATH_ORDONLY	dynit.h	General
__PATH_ORDWR	dynit.h	General
__PATH_OTRUNC	dynit.h	General
__PATH_OWRONLY	dynit.h	General
__PATH_SIRGRP	dynit.h	General
__PATH_SIROTH	dynit.h	General
__PATH_SIRUSR	dynit.h	General
__PATH_SIRWXG	dynit.h	General
__PATH_SIRWXO	dynit.h	General
__PATH_SIRWXU	dynit.h	General
__PATH_SISGID	dynit.h	General
__PATH_SISUID	dynit.h	General
__PATH_SIWGRP	dynit.h	General
__PATH_SIWOTH	dynit.h	General
__PATH_SIWUSR	dynit.h	General
__PATH_SIXGRP	dynit.h	General
__PATH_SIXOTH	dynit.h	General
__PATH_SIXUSR	dynit.h	General
PCB_STRUCT(key_len)	ims.h	General
__pcblist	ims.h	General

Table 62. XL C/C++ Macros (continued)

Macro	Include File	General or Internal
__PCBLIST_INDEX	ims.h	Internal
__PERM	dynit.h	General
PLOSS	math.h	General
PM_STR	nl_langinfo.h	General
pow(x,y)	math.h	General
__PRINTER	stdio.h	General
__psizeof(type)	stdarg.h	Internal
putc(c, p)	stdio.h	General
__putc(c, p)	stdio.h	Internal
putchar(c)	stdio.h	General
RADIXCHAR	nl_langinfo.h	General
RAND_MAX	stdlib.h	General
__RBA_EQ	stdio.h	General
__RBA_EQ_BWD	stdio.h	General
__READ	stdio.h	General
__RECORD	stdio.h	General
REG_BADBR	regex.h	General
REG_BADPT	regex.h	General
REG_BADRPT	regex.h	General
REG_EBOL	regex.h	General
REG_EBRACE	regex.h	General
REG_EBRACK	regex.h	General
REG_ECHAR	regex.h	General
REG_ECOLLATE	regex.h	General
REG_ECTYPE	regex.h	General
REG_EEOL	regex.h	General
REG_EESCAPE	regex.h	General
REG_EPAREN	regex.h	General
REG_ERANGE	regex.h	General
REG_ESPACE	regex.h	General
REG_NOMATCH	regex.h	General
__RELEASE	dynit.h	General
release(x)	stdlib.h	General
__RLS_CR	dynit.h	General
__RLS_CRE	dynit.h	General
__RLS_NRI	dynit.h	General
__ROUND	dynit.h	General
__RR	dynit.h	General
__RRDS	stdio.h	General
__rsn_i	signal.h	Internal
__rsncode	signal.h, stdlib.h	General
__R1	stdlib.h	General
_S_	dynit.h	General
SCHAR_MAX	limits.h	General
SCHAR_MIN	limits.h	General
SEEK_CUR	stdio.h	General
SEEK_END	stdio.h	General
SEEK_SET	stdio.h	General
__setjmp	setjmp.h	Internal
setjmp(x)	setjmp.h	General
SHRT_MAX	limits.h	General
SHRT_MIN	limits.h	General
SIG_DFL	signal.h	General

Table 62. XL C/C++ Macros (continued)

Macro	Include File	General or Internal
SIG_ERR	signal.h	General
SIG_IGN	signal.h	General
SIG_PROMOTE	signal.h	General
SIGABND	signal.h	General
SIGABRT	signal.h	General
SIGFPE	signal.h	General
SIGILL	signal.h	General
SIGINT	signal.h	General
__signal	signal.h	Internal
SIGSEGV	signal.h	General
SIGTERM	signal.h	General
SIGUSR1	signal.h	General
SIGUSR2	signal.h	General
sin(x)	math.h	General
SING	math.h	General
sinh(x)	math.h	General
__size_t	time.h	Internal
__spc	spc.h	Internal
sqrt(x)	math.h	General
__stdarg	stdarg.h	Internal
__stddef	stddef.h	Internal
stderr	assert.h	General
__stderr_i	assert.h	Internal
stdin	stdio.h	General
__stdin_i	stdio.h	Internal
__stdio	stdio.h	Internal
__stdlib	stdlib.h	Internal
stdout	stdio.h	General
__stdout_i	stdio.h	Internal
strcat(x,y)	string.h	General
strchr(x,y)	string.h	General
strcmp(x,y)	string.h	General
strcpy(x,y)	string.h	General
__string	string.h	Internal
strlen(x)	string.h	General
strrchr(x,y)	string.h	General
__SVC99	stdio.h	Internal
svc99(x)	stdio.h	General
__SVC99_ALLOC	stdio.h	General
__SVC99_ALLOC_NEW	stdio.h	General
__sysplist	stdlib.h	General
__sysplist_i	stdlib.h	Internal
__SYSPLIST_INDEX	stdlib.h	Internal
T_FMT	nl_langinfo.h	General
T_FMT_AMPM	nl_langinfo.h	General
tan(x)	math.h	General
tanh(x)	math.h	General
__TAPE	stdio.h	General
__TDQ	stdio.h	General
__temp	ctype.h	Internal
__temp_a	stdio.h	Internal
__temp_i	ctype.h	Internal
__TERM	dynit.h	General

Table 62. XL C/C++ Macros (continued)

Macro	Include File	General or Internal
__TERMINAL	stdio.h	General
__TEXT	stdio.h	General
__TGET_READ	stdio.h	General
THOUSEP	nl_langinfo.h	General
__time	time.h	Internal
tinit(x,y)	mtf.h	General
TLOSS	math.h	General
TMP_MAX	stdio.h	General
tolower(c)	ctype.h	General
__tolower(c)	ctype.h	Internal
__TOLOWER_INDEX	ctype.h	Internal
toupper(c)	ctype.h	General
__toupper(c)	ctype.h	Internal
__TOUPPER_INDEX	ctype.h	Internal
towlower	wchar.h	General
towupper	wchar.h	General
__TPUT_WRITE	stdio.h	General
__TRK	dynit.h	General
tsched	mtf.h	General
tsetsubt(x,y)	mtf.h	General
tsyncro(x)	mtf.h	General
tterm(x)	mtf.h	General
_U_	dynit.h	General
UCHAR_MAX	limits.h	General
UINT_MAX	limits.h	General
ULONG_MAX	limits.h	General
UNDERFLOW	math.h	General
__UNKN_ERROR	stdio.h	General
__UPDATE	stdio.h	General
USHRT_MAX	limits.h	General
_V_	dynit.h	General
va_arg(ap, type)	stdarg.h	General
va_end(ap)	stdarg.h	General
va_list	stdarg.h	General
va_start(ap, arg)	stdarg.h	General
__valist	stdarg.h	Internal
__valist_	stdio.h	Internal
_VB_	dynit.h	General
VBS	dynit.h	General
__VSAM_CLOSE	stdio.h	General
__VSAM_ENDREQ	stdio.h	General
__VSAM_ERASE	stdio.h	General
__VSAM_GENCB	stdio.h	General
__VSAM_GET	stdio.h	General
__VSAM_MODCB	stdio.h	General
__VSAM_OPEN_ESDS	stdio.h	General
__VSAM_OPEN_ESDS_PAT	stdio.h	General
__VSAM_OPEN_FAIL	stdio.h	General
__VSAM_OPEN_KSDS	stdio.h	General
__VSAM_OPEN_KSDS_PAT	stdio.h	General
__VSAM_OPEN_RRDS	stdio.h	General
__VSAM_POINT	stdio.h	General
__VSAM_PUT	stdio.h	General

Table 62. XL C/C++ Macros (continued)

Macro	Include File	General or Internal
__VSAM_SHOWCB	stdio.h	General
__VSAM_TESTCB	stdio.h	General
__wchar_t	stddef.h	Internal
__wcstr	wcstr.h	Internal
WEOF	wchar.h	General
wmemchr(x,y,z)	wchar.h	General
wmemcmp(x,y,z)	wchar.h	General
wmemcpy(x,y,z)	wchar.h	General
wmemset(x,y,z)	wchar.h	General
__WRITE	stdio.h	General
YESEXPR	nl_langinfo.h	General

Appendix B. Function support table

Preinitialized environments for authorized programs

Preinitialized Environments for Authorized Programs is a new feature of z/OS Language Environment. It is intended to provide support for authorized components or products to create preinitialized environments that are capable of executing C, C++, and Language Environment-conforming assembler code in supervisor state, on a TCB or an SRB, or in cross-memory mode (with some restrictions).

Table 64 on page 2116 lists all the functions in the z/OS XL C/C++ Runtime Library and their support of Preinitialized Environments for Authorized Programs.

Note: The function table does not include compiler built-in functions (builtin.h).

Enhanced ASCII support

Restriction: Enhanced ASCII and __LIBASCII are independent, and should not be used together. Using Enhanced ASCII and __LIBASCII together is not supported.

Enhanced ASCII support provides the means to write z/OS XL C/C++ applications which will execute with ASCII data representation.

Enhanced ASCII support makes it easier to port internationalized C/C++ applications developed on or for ASCII platforms to z/OS platforms by providing conversion from ASCII to EBCDIC and EBCDIC to ASCII.

In order to use Enhanced ASCII support, a C or C++ module must be compiled specifying ASCII as the data representation. Application compile units will be bound to ASCII versions of external variables and interfaces at compile time if they:

1. Use headers to declare external variables and interfaces used in compile unit source.
2. Are compiled with the ASCII option.

New ASCII function-versions and other support functions have been added to the z/OS XL C/C++ Runtime Library to handle ASCII data in files, data manipulations, and conversions between ASCII and EBCDIC. The compile-time binding will determine which ASCII or EBCDIC function version is called during runtime execution.

Table 64 on page 2116 lists all the functions in the z/OS XL C/C++ Runtime Library and their support of Enhanced ASCII processing.

Note: The function table does not include compiler built-in functions (builtin.h).

Table 63 on page 2114 lists all the External Variables and their support status in Enhanced ASCII. For other information about these variables, see “External variables” on page 96.

Table 63. Status of External Variables in Enhanced ASCII

External Variable	Enhanced ASCII Support Level
daylight	Neutral
environ	Yes
errno	Neutral
getdate_err	Neutral
h_errno	Neutral
locs	No
loc1	No
__loc1	Neutral
loc2	No
optarg	Neutral
opterr	Neutral
optind	Neutral
optopt	Neutral
signgam	Neutral
stderr	Neutral
stdin	Neutral
stdout	Neutral
t_errno	Neutral
timezone	Neutral
tzname	Yes

For a description of Limitations to the use of Enhanced ASCII, see *z/OS XL C/C++ Programming Guide*.

Enhanced ASCII extensions

Applications compiled ASCII were permitted to use library functions that did not have ASCII support for character data with the understanding that all ASCII to EBCDIC conversion was the responsibility of the application.

As service updates or new releases extend Enhanced ASCII to previously unsupported functions, it is necessary to protect the ASCII applications that have been calling those functions (with user-supplied character conversions).

A new feature test macro is defined to control the exposure of extensions to Enhanced ASCII. This feature test macro is _ENHANCED_ASCII_EXT and should be set to a non-zero, numeric value indicating the level of Enhanced ASCII support desired. Otherwise, the default value of 0x41020000 is assumed, which will limit exposure to only the Enhanced ASCII support provided in the z/OS V1R2 base. Numeric values less than the default will behave as if the default were specified. The special value 0xFFFFFFFF provides exposure to all Enhanced ASCII support, regardless of the service level, available for the TARGET release.

Functions whose Enhanced ASCII support exposure is controlled by values of this feature test macro, other than the default value or the special 0xFFFFFFFF value, will have the specific value documented in Table 64 on page 2116.

Note: Functions which are introduced in a release, and include Enhanced ASCII support, are not controlled using this macro. For example, getnameinfo() first appears in z/OS V1R4 base and has Enhanced ASCII support. It is always available in its ASCII form since there was no prior release containing the function.

Table 64 on page 2116 lists all the functions in the z/OS XL C/C++ Runtime Library and their support of Enhanced ASCII processing.

Library function support

Table 64 on page 2116 shows all the z/OS XL C/C++ Runtime Library functions in alphabetical order, their support of Enhanced ASCII processing and their support of Preinitialized Environments for Authorized Programs.

Enhanced ASCII

Each function is identified by the extent to which it supports Enhanced ASCII processing:

- Yes = supports Enhanced ASCII.
- No = does not support Enhanced ASCII.
- Neutral = not sensitive to the issue of ASCII/EBCDIC character encoding.

Following is a list of values for feature test macro _ENHANCED_ASCII_EXT:

0x41020000
z/OS V1.2 base support plus all functions first introduced in subsequent releases. This is the default.

0x41020010
Includes support added with APAR PQ63405

0x41060000
Includes support added in z/OS V1.6.

0x41070000
Includes support added in z/OS V1.7

0x410A0000
Includes support added in z/OS V1.10

0xFFFFFFFF
Exposes all Enhanced ASCII support regardless of service level for the TARGET release.

Each higher value includes all support exposed with the lesser values.

Functions which are used for conversion to ASCII or EBCDIC are labelled with those data types.

Preinitialized Environments for Authorized Programs

Each function is identified by the extent to which it supports Preinitialized Environments for Authorized Programs:

Yes the function supports Preinitialized Environments for Authorized Programs.

No the function does not support Preinitialized Environments for Authorized Programs.

Table 64. Library function support table

Function	Enhanced ASCII Support Level	Minimum Value for _ENHANCED_ASCII_EXT Feature Test Macro	Preinitialized Environments for Authorized Programs Support Level	Notes
abort()	Neutral		Yes	4
abs()	Neutral		Yes	
absf()	Neutral		Yes	
absl()	Neutral		Yes	
accept()	Yes	0x41020010	Yes	2,3,8
accept_and_recv()	Yes	0x41020010	No	3
access()	Yes		No	
acl_create_entry()	Neutral		No	
acl_delete_entry()	Neutral		No	
acl_delete_fd()	Neutral		No	
acl_delete_file()	Neutral		No	
acl_first_entry()	Neutral		No	
acl_free()	Neutral		No	
acl_from_text()	Neutral		No	
acl_get_entry()	Neutral		No	
acl_get_fd()	Neutral		No	
acl_get_file()	Neutral		No	
acl_init()	Neutral		No	
acl_set_fd()	Neutral		No	
acl_set_file()	Neutral		No	
acl_sort()	Neutral		No	
acl_to_text()	Neutral		No	
acl_update_entry()	Neutral		No	
acl_valid()	Neutral		No	
acos()	Neutral		Yes	
acosd32(), acosd64(), acosd128()	Neutral		No	
acosf()	Neutral		Yes	
acosh()	Neutral		Yes	
acoshd32(), acoshd64(), acoshd128()	Neutral		No	
acoshf()	Neutral		Yes	
acoshl()	Neutral		Yes	
acosl()	Neutral		Yes	

Table 64. Library function support table (continued)

Function	Enhanced ASCII Support Level	Minimum Value for _ENHANCED_ASCII_EXT Feature Test Macro	Preinitialized Environments for Authorized Programs Support Level	Notes
advance()	No		No	
__ae_correstbl_query()	Yes		No	
aio_cancel()	No		No	
aio_error()	No		No	
aio_read()	No		No	
aio_return()	No		No	
aio_suspend()	No		No	
aio_write()	No		No	
alarm()	Neutral		No	
alloca()	Neutral		Yes	
asctime(), asctime64()	Yes		Yes	
asctime_r(), asctime64_r()	Yes		Yes	
asin()	Neutral		Yes	
asind32(), asind64(), asind128()	Neutral		No	
asinf()	Neutral		Yes	
asinh()	Neutral		Yes	
asinhd32(), asinhd64(), asinhd128()	Neutral		No	
asinhf()	Neutral		Yes	
asinhl()	Neutral		Yes	
asinl()	Neutral		Yes	
assert()	Neutral		No	
atan()	Neutral		Yes	
atand32(), atand64(), atand128(),atan2d32(), atan2d64(), atan2d128()	Neutral		No	
atanf()	Neutral		Yes	
atanh()	Neutral		Yes	
atanhd32(), atanhd64(), atanhd128()	Neutral		No	
atanhf()	Neutral		Yes	
atanhl()	Neutral		Yes	
atanl()	Neutral		Yes	

Table 64. Library function support table (continued)

Function	Enhanced ASCII Support Level	Minimum Value for _ENHANCED_ASCII_EXT Feature Test Macro	Preinitialized Environments for Authorized Programs Support Level	Notes
__atanpid32(), __atanpid64(), __atanpid128()	Neutral		No	
atan2()	Neutral		Yes	
atan2f()	Neutral		Yes	
atan2l()	Neutral		Yes	
atexit()	Neutral		Yes	
__atoe()	ASCII		No	
__atoe_l()	ASCII		No	
atof()	Yes		Yes	
atoi()	Yes		Yes	
atol()	Yes		Yes	
atoll()	Yes		No	
__a2e_l()	ASCII		No	
__a2e_s()	ASCII		No	
a64l()	Yes		Yes	
basename()	Yes		Yes	
bcmp()	Neutral		Yes	
bcopy()	Neutral		Yes	
bind()	Yes	0x41020010	Yes	2,3,8
bind2addrsel()	Neutral		Yes	8
brk()	Neutral		No	
bsd_signal()	Neutral		No	
bsearch()	Neutral		Yes	
btowc()	Yes		Yes	
bzero()	Neutral		Yes	
c16rtomb()	Neutral		No	
c32rtomb()	Neutral		No	
__cabend()	No		No	
calloc()	Neutral		Yes	
catclose()	Neutral		No	
catgets()	Yes		No	
catopen()	Yes		No	
cbrt()	Neutral		Yes	

Table 64. Library function support table (continued)

Function	Enhanced ASCII Support Level	Minimum Value for _ENHANCED_ASCII_EXT Feature Test Macro	Preinitialized Environments for Authorized Programs Support Level	Notes
cbrtd32() cbrtd64() cbrtd128()	Neutral		No	
cbrtf()	Neutral		Yes	
cbrtl()	Neutral		Yes	
cclass()	No		Yes	
__CcsidType()	Yes		No	
cds()	Neutral		Yes	
cdump()	Yes		No	
ceil()	Neutral		Yes	
ceild32(), ceild64(), ceild128()	Neutral		No	
ceilf()	Neutral		Yes	
ceill()	Neutral		Yes	
__certificate()	No		No	
cfgetispeed()	No		No	
cfgetospeed()	No		No	
cfsetispeed()	No		No	
cfsetospeed()	No		No	
__chattr()	Yes		No	
chaudit()	Yes		No	
chdir()	Yes		No	
__check_resource_auth_np()	Yes		No	
CheckSchEnv()	Yes		No	
chmod()	Yes		No	
chown()	Yes		No	
chpriority()	Neutral		No	
chroot()	Yes		No	
clearenv()	Neutral		Yes	
clearerr(), clearerr_unlocked()	Neutral		Yes	
clock()	Neutral		No	
close()	Neutral		Yes	8
closedir()	Neutral		No	
closelog()	Neutral		No	
clrmemf()	Neutral		Yes	

Table 64. Library function support table (continued)

Function	Enhanced ASCII Support Level	Minimum Value for _ENHANCED_ASCII_EXT Feature Test Macro	Preinitialized Environments for Authorized Programs Support Level	Notes
__cnvblk()	Neutral		No	
collequiv()	No		Yes	
collorder()	No		Yes	
collrange()	No		Yes	
colltostr()	No		Yes	
compile()	No		No	
confstr()	Yes		Yes	
connect()	Yes	0x41020010	Yes	2,3,8
ConnectExportImport()	No		No	
ConnectServer()	Yes		No	
ConnectWorkMgr()	Yes		No	
__console()	Yes		No	
__console2()	No		No	
ContinueWorkUnit()	Neutral		No	
__convert_id_np()	No		No	
copysign()	Neutral		Yes	
copysignd32(), copysingd64() copysingd128()	Neutral		No	
copysignf()	Neutral		Yes	
copysignl()	Neutral		Yes	
cos()	Neutral		Yes	
cosd32(), cosd64(), cosd128()	Neutral		No	
cosf()	Neutral		Yes	
cosh()	Neutral		Yes	
coshd32(), coshd64(), coshd128()	Neutral		No	
coshf()	Neutral		Yes	
coshl()	Neutral		Yes	
cosl()	Neutral		Yes	
__cospid32(), __cospid64(), __cospid128()	Neutral		No	
__cotan()	Neutral		No	
__cotanf()	Neutral		No	
__cotanl()	Neutral		No	

Table 64. Library function support table (continued)

Function	Enhanced ASCII Support Level	Minimum Value for _ENHANCED_ASCII_EXT Feature Test Macro	Preinitialized Environments for Authorized Programs Support Level	Notes
__cpl()	Neutral		No	
creat()	Yes		No	
CreateWorkUnit()	Yes		No	
crypt()	Yes		Yes	
cs()	Neutral		Yes	
csid()	Yes		Yes	
__CSNameType()	Yes		No	
csnap()	Yes		No	
__csplist	No		No	
ctdli()	Neutral		No	
ctermid()	Yes		No	
ctest()	Yes		Yes	
ctime(), ctime64()	Yes		Yes	
ctime_r(), ctime64_r()	Yes		Yes	
ctrace()	Yes		No	
cuserid()	Yes		No	
dbm_clearerr()	Neutral		No	
dbm_close()	Neutral		No	
dbm_delete()	Neutral		No	
dbm_error()	Neutral		No	
dbm_fetch()	Neutral		No	
dbm_firstkey()	Neutral		No	
dbm_nextkey()	Neutral		No	
dbm_open()	Yes		No	
dbm_store()	Neutral		No	
decabs()	Neutral		Yes	
decchk()	Neutral		Yes	
decfix()	Neutral		Yes	
DeleteWorkUnit()	Neutral		No	
difftime(), difftime64()	Neutral		Yes	
dirname()	Yes		No	
__discarddata()	Neutral		No	

Table 64. Library function support table (continued)

Function	Enhanced ASCII Support Level	Minimum Value for _ENHANCED_ASCII_EXT Feature Test Macro	Preinitialized Environments for Authorized Programs Support Level	Notes
DisconnectServer()	Neutral		No	
div()	Neutral		Yes	
dlclose()	Neutral		No	
dlerror()	Yes		No	
dllfree()	Neutral		Yes	
dllload()	Yes		Yes	7
dllqueryfn()	Yes		Yes	
dllqueryvar()	Yes		Yes	
dlopen()	Yes		No	7
dlsym()	Yes		No	
dn_comp()	Yes	0x41020010	No	
dn_expand()	Yes	0x41020010	No	
dn_find()	Yes	0x41020010	No	
dn_skipname()	Yes	0x41020010	No	
drand48()	Neutral		Yes	
dup()	Neutral		No	
dup2()	Neutral		No	
dynalloc()	Yes	0x41020010	No	
dynfree()	Yes	0x41020010	No	
dyninit()	No		No	
ecvt()	Yes		Yes	
encrypt()	Neutral		Yes	
endgrent()	Neutral		No	
endhostent()	Neutral		No	
endnetent()	Neutral		No	
endprotoent()	Neutral		No	
endpwent()	Neutral		No	
endservent()	Neutral		No	
endutxent()	Neutral		No	
erand48()	Neutral		Yes	
erf()	Neutral		Yes	
erfc()	Neutral		Yes	

Table 64. Library function support table (continued)

Function	Enhanced ASCII Support Level	Minimum Value for _ENHANCED_ASCII_EXT Feature Test Macro	Preinitialized Environments for Authorized Programs Support Level	Notes
erfcf()	Neutral		Yes	
erfcl()	Neutral		Yes	
erff()	Neutral		Yes	
erfl()	Neutral		Yes	
erfd32(), erfd64(), erfd128(), erfcd32(), erfcd64(), erfcd128()	Neutral		No	
__errno2()	Neutral		Yes	
__err2ad()	Neutral		Yes	
__etoa()	EBCDIC		No	
__etoa_l()	EBCDIC		No	
execl()	Yes		No	
execle()	Yes		No	
execlp()	Yes		No	
execv()	Yes		No	
execve()	Yes		No	
execvp()	Yes		No	
exit()	Neutral		Yes	
_exit()	Neutral		No	
_Exit()	Neutral		No	
exp()	Neutral		Yes	
expd32(), expd64(), expd128()	Neutral		No	
expm1d32() expm1d64() expm1d128()	Neutral		No	
exp2()	Neutral		Yes	
exp2d32() exp2d64() exp2d128()	Neutral		No	
exp2f()	Neutral		Yes	
exp2l()	Neutral		Yes	
expf()	Neutral		Yes	
expl()	Neutral		Yes	
expm1()	Neutral		Yes	
expm1f()	Neutral		Yes	
expm1l()	Neutral		Yes	
ExportWorkUnit()	Neutral		No	

Table 64. Library function support table (continued)

Function	Enhanced ASCII Support Level	Minimum Value for _ENHANCED_ASCII_EXT Feature Test Macro	Preinitialized Environments for Authorized Programs Support Level	Notes
extlink_np()	Yes		No	
ExtractWorkUnit()	Neutral		No	
__e2a_l()	EBCDIC		No	
__e2a_s()	EBCDIC		No	
fabs()	Neutral		Yes	
fabsd32(), fabsd64(), fabsd128()	Neutral		No	
fabsf()	Neutral		Yes	
fabsl()	Neutral		Yes	
fattach()	No		No	
__fbufsize()	Neutral		No	
__fchattr()	Neutral		No	
fchaudit()	Neutral		No	
fchdir()	Neutral		No	
fchmod()	Neutral		No	
fchown()	Neutral		No	
fclose()	Neutral		No	
fcntl()	Neutral		No	
fcvt()	Yes		Yes	
fdelrec(), fdelrec_unlocked()	Neutral		No	
fdetach()	No		No	
fdim()	Neutral		Yes	
fdimd32(), fdimd64(), fdimd128()	Neutral		No	
fdimf()	Neutral		Yes	
fdiml()	Neutral		Yes	
fdopen()	Yes		No	
fe_dec_getround()	Neutral		No	
fe_dec_setround()	Neutral		No	
fegetround()	Neutral		No	
feof(), feof_unlocked()	Neutral		No	
ferror(), ferror_unlocked()	Neutral		No	
fesetenv()	Neutral		No	
fetch()	Yes		Yes	

Table 64. Library function support table (continued)

Function	Enhanced ASCII Support Level	Minimum Value for _ENHANCED_ASCII_EXT Feature Test Macro	Preinitialized Environments for Authorized Programs Support Level	Notes
fetchep()	Neutral		Yes	
fetestexcept()	Neutral		No	
fflush(), fflush_unlocked()	Neutral		No	
ffs()	Neutral		Yes	
fgetc(), fgetc_unlocked()	Neutral		No	
fgetpos(), fgetpos_unlocked()	Neutral		No	
fgets(), fgets_unlocked()	Yes		No	
fgetwc(), fgetwc_unlocked()	Yes		No	
fgetws(), fgetws_unlocked()	Yes		No	
fileno(), fileno_unlocked()	Neutral		No	
finite()	Neutral		Yes	
__flbf()	Neutral		No	
fldata(), fldata_unlocked()	Yes		No	
flocate(), flocate_unlocked()	Neutral		No	
flockfile()	Neutral		No	
floor()	Neutral		Yes	
floord32(), floord64(), floord128()	Neutral		No	
floorf()	Neutral		Yes	
floorl()	Neutral		Yes	
_flushlbf()	Neutral		No	
fmad32() fmad64() fmad128()	Neutral		No	
fmaxd32(), fmaxd64(), fmaxd128()	Neutral		No	
fmod()	Neutral		Yes	
fmodd32() fmodd64() fmodd128()	Neutral		No	
fmodf()	Neutral		Yes	
fmodl()	Neutral		Yes	
fmtmsg()	Yes		No	
fnmatch()	Yes		Yes	
fopen()	Yes		No	
fork()	Neutral		No	
fortrc()	Neutral		No	
fp_clr_flag()	Neutral		Yes	

Table 64. Library function support table (continued)

Function	Enhanced ASCII Support Level	Minimum Value for _ENHANCED_ASCII_EXT Feature Test Macro	Preinitialized Environments for Authorized Programs Support Level	Notes
fp_raise_xcp()	Neutral		Yes	
fp_read_flag()	Neutral		Yes	
fp_read_rnd()	Neutral		Yes	
fp_swap_rnd()	Neutral		Yes	
fpathconf()	Neutral		No	
__fpending()	Neutral		No	
fprintf(), fprintf_unlocked()	Yes		No	
__fpurge()	Neutral		No	
fputc(), fputc_unlocked()	Yes		No	
fputs(), fputs_unlocked()	Yes		No	
fputwc(), fputwc_unlocked()	Yes		No	
fputws(), fputws_unlocked()	Yes		No	
fread(), fread_unlocked()	Yes		No	
__freadable()	Neutral		No	
__freadahead	Neutral		No	
__freading()	Neutral		No	
free()	Neutral		Yes	
freeaddrinfo()	Neutral		No	
freopen()	Yes		No	
frexp()	Neutral		Yes	
frexpd32(), frexpd64(), frexpd128()	Neutral		No	
frexpf()	Neutral		Yes	
frexpl()	Neutral		Yes	
fscanf(), fscanf_unlocked()	Yes		No	
fseek(), fseek_unlocked()	Neutral		No	
fseeko(), fseeko_unlocked()	Neutral		No	
__fseterr()	Neutral		No	
__fsetlocking()	Neutral		No	
fsetpos(), fsetpos_unlocked()	Neutral		No	
fstat()	Neutral		No	
fstatvfs()	Neutral		No	
fsync()	Neutral		No	

Table 64. Library function support table (continued)

Function	Enhanced ASCII Support Level	Minimum Value for _ENHANCED_ASCII_EXT Feature Test Macro	Preinitialized Environments for Authorized Programs Support Level	Notes
ftell(), ftell_unlocked()	Neutral		No	
ftello(), ftello_unlocked()	Neutral		No	
ftime()	Neutral		Yes	
ftok()	Yes		No	
ftruncate()	Neutral		No	
ftrylockfile()	Neutral		No	
ftw()	Yes		No	
funlockfile()	Neutral		No	
fupdate(), fupdate_unlocked()	Neutral		No	
fwide(), fwide_unlocked()	Neutral		No	
fwprintf(), fwprintf_unlocked()	Yes	0x41070000	No	
__fwritable()	Neutral		No	
fwrite(), fwrite_unlocked()	Yes		No	
__fwriting()	Neutral		No	
fwscanf(), fwscanf_unlocked()	Yes		No	
gai_strerror	Yes		No	
gamma()	Neutral		Yes	
gcvt()	Yes		Yes	
getaddrinfo()	Yes		No	
getc()	Neutral		No	
getc_unlocked()	Neutral		No	
getchar()	Neutral		No	
getchar_unlocked()	Neutral		No	
getclientid()	No		No	
__getclientid()	No		No	
getcontext()	Neutral		No	
__get_cpuid()	No		Yes	
getcwd()	Yes		No	
getdate(), getdate64()	Yes		Yes	
getdtablesize()	Neutral		No	
getegid()	Neutral		No	
getenv()	Yes		Yes	

Table 64. Library function support table (continued)

Function	Enhanced ASCII Support Level	Minimum Value for _ENHANCED_ASCII_EXT Feature Test Macro	Preinitialized Environments for Authorized Programs Support Level	Notes
__getenv()	Yes	0x41060000	Yes	
geteuid()	Neutral		No	
getgid()	Neutral		No	
getgrent()	Yes		No	
getgrgid()	Yes		No	
__getgrgid1	Yes		No	
getgrgid_r()	Yes		No	
getgrnam()	Yes		No	
__getgrnam1	Yes		No	
getgrnam_r()	Yes		No	
getgroups()	Neutral		No	
getgroupsbyname()	Yes		No	
gethostbyaddr()	Yes		No	2
gethostbyname()	Yes		No	2
gethostent()	Yes		No	2
gethostid()	Neutral		No	
gethostname()	Yes		No	
getibmopt()	No		No	
getibmsockopt()	No		No	
__getipc()	No		No	
getipv4sourcefilter()	Neutral		No	
getitimer()	Neutral		No	
getlogin()	Yes		No	1
getlogin_r()	Yes		No	
__getlogin1()	No		No	
getmccoll()	No		Yes	
getmsg()	No		No	
getnameinfo()	Yes		No	
getnetbyaddr()	Yes		No	2
getnetbyname()	Yes		No	2
getnetent()	Yes		No	2
getopt()	Yes		Yes	

Table 64. Library function support table (continued)

Function	Enhanced ASCII Support Level	Minimum Value for _ENHANCED_ASCII_EXT Feature Test Macro	Preinitialized Environments for Authorized Programs Support Level	Notes
getpagesize()	Neutral		Yes	
getpass()	Yes		No	
getpeername()	Yes	0x41020010	No	2,3
getpgid()	Neutral		No	
getpgrp()	Neutral		No	
getpid()	Neutral		No	
getpmsg()	No		No	
getppid()	Neutral		No	
getpriority()	Neutral		No	
getprotobyname()	Yes		No	2
getprotobynumber()	Yes		No	2
getprotoent()	Yes		No	2
getpwent()	Yes		No	
getpwnam()	Yes		No	
getpwnam_r()	Yes		No	
getpwuid()	Yes		No	
getpwuid_r()	Yes		No	
getrlimit()	Neutral		No	
getrusage()	Neutral		No	
gets(), gets_unlocked()	Yes		No	
getservbyname()	Yes		No	2
getservbyport()	Yes		No	2
getservent()	Yes		No	2
getsid()	Neutral		No	
getsockname()	Yes	0x41020010	Yes	2,3,8
getsockopt()	No		Yes	8
getsourcefilter()	Neutral		No	
getstablesize()	No		No	
getsubopt()	No		Yes	
getsyntx()	No		Yes	
__get_system_settings()	No		No	
gettimeofday(), gettimeofday64()	Yes		Yes	

Table 64. Library function support table (continued)

Function	Enhanced ASCII Support Level	Minimum Value for _ENHANCED_ASCII_EXT Feature Test Macro	Preinitialized Environments for Authorized Programs Support Level	Notes
getuid()	Neutral		No	
__getuserid()	No		No	
getutxent()	Neutral		No	
getutxid()	No		No	
getutxline()	No		No	
getw()	Neutral		No	
getwc(), getwc_unlocked()	Yes		No	
getwchar(), getwchar_unlocked()	Yes		No	
getwd()	Yes		No	
getwmccoll()	No		Yes	
givesocket()	No		No	
glob()	Yes		No	
globfree()	Neutral		No	
gmtime(), gmtime64()	Yes		Yes	
gmtime_r(), gmtime64_r()	Yes		Yes	
grantpt()	Neutral		No	
hcreate()	Neutral		Yes	
hdestroy()	Neutral		Yes	
__heaprpt()	Neutral		No	
hsearch()	Neutral		Yes	
htonl()	Neutral		No	
htons()	Neutral		No	
hypot()	Neutral		Yes	
hypotd32(), hypotd64(), hypotd128()	Neutral		No	
hypotf()	Neutral		Yes	
hypotl()	Neutral		Yes	
ibmsflush()	Neutral		No	
iconv()	Neutral		No	
iconv_close()	Neutral		No	
iconv_open()	Yes		No	
if_freenameindex()	Neutral		No	
if_indextoname()	Yes		No	

Table 64. Library function support table (continued)

Function	Enhanced ASCII Support Level	Minimum Value for _ENHANCED_ASCII_EXT Feature Test Macro	Preinitialized Environments for Authorized Programs Support Level	Notes
if_nameindex()	Yes		No	
if_nametoindex()	Yes		No	
ilogb()	Neutral		Yes	
ilogbd32(), ilogbd64(), ilogbd128()	Neutral		No	
imaxabs()	Neutral		No	
imaxdiv()	Neutral		No	
ImportWorkUnit()	Neutral		No	
index()	Neutral		Yes	
inet6_is_srcaddr()	Neutral		No	
inet6_opt_append()	Neutral		No	
inet6_opt_find()	Neutral		No	
inet6_opt_finish()	Neutral		No	
inet6_opt_get_val()	Neutral		No	
inet6_opt_init()	Neutral		No	
inet6_opt_next()	Neutral		No	
inet6_opt_set_val()	Neutral		No	
inet6_rth_add()	Neutral		No	
inet6_rth_getaddr()	Neutral		No	
inet6_opt_init()	Neutral		No	
inet6_rth_reverse()	Neutral		No	
inet6_rth_segments()	Neutral		No	
inet6_rth_space()	Neutral		No	
inet_addr()	Yes	0x41020010	No	2
inet_lnaof()	Neutral		No	
inet_makeaddr()	No		No	
inet_netof()	Neutral		No	
inet_network()	Yes	0x41020010	No	2
inet_ntoa()	Yes	0x41020010	No	2
inet_ntop()	Yes	0x41020010	Yes	8
inet_pton()	Yes	0x41020010	Yes	8
initgroups()	Yes		No	
initstate()	No		Yes	

Table 64. Library function support table (continued)

Function	Enhanced ASCII Support Level	Minimum Value for _ENHANCED_ASCII_EXT Feature Test Macro	Preinitialized Environments for Authorized Programs Support Level	Notes
insque()	Neutral		Yes	
ioctl()	No		Yes	8
__ipdbcs()	No		No	
__ipDomainName()	No		No	
__ipdspx()	No		No	
__iphost()	No		No	
__ipmsgc()	No		No	
__ipnode()	No		No	
__iptcpn()	No		No	
isalnum()	Yes		Yes	
isalpha()	Yes		Yes	
isascii()	Neutral		Yes	
isastream()	Neutral		No	
isatty()	Neutral		No	
__isBFP()	Neutral		No	
isblank()	Yes	0x41070000	Yes	
iscics()	Neutral		No	
iscntrl()	Yes		Yes	
isdigit()	Yes		Yes	
isgraph()	Yes		Yes	
islower()	Yes		Yes	
ismccollel()	No		Yes	
isnan()	Neutral		Yes	
__isPosixOn()	Neutral		No	
isprint()	Yes		Yes	
ispunct()	Yes		Yes	
isspace()	Yes		Yes	
isupper()	Yes		Yes	
iswalnum()	Yes		Yes	
iswalpha()	Yes		Yes	
iswblank()	Yes	0x41070000	Yes	
iswcntrl()	Yes		Yes	

Table 64. Library function support table (continued)

Function	Enhanced ASCII Support Level	Minimum Value for _ENHANCED_ASCII_EXT Feature Test Macro	Preinitialized Environments for Authorized Programs Support Level	Notes
iswctype()	Yes		Yes	
iswdigit()	Yes		Yes	
iswgraph()	Yes		Yes	
iswlower()	Yes		Yes	
iswprint()	Yes		Yes	
iswpunct()	Yes		Yes	
iswspace()	Yes		Yes	
iswupper()	Yes		Yes	
iswxdigit()	Yes		Yes	
isxdigit()	Yes		Yes	
itoa()	Yes		Yes	
jn()	Neutral		Yes	
JoinWorkUnit()	Neutral		No	
jrand48()	Neutral		Yes	
j0()	Neutral		Yes	
j1()	Neutral		Yes	
kill()	Neutral		No	
killpg()	Neutral		No	
labs()	Neutral		Yes	
__lchattr()	Yes		No	
lchown()	Yes		No	
lcong48()	Neutral		Yes	
ldexp()	Neutral		Yes	
ldexpd32(), ldexpd64(), ldexpd128()	Neutral		No	
ldexpf()	Neutral		Yes	
ldexpl()	Neutral		Yes	
ldiv()	Neutral		Yes	
LeaveWorkUnit()	Neutral		No	
__le_cib_get()	No		No	
__le_ceegtjs	No		No	
__le_condition_token_build()	No		No	
__le_msg_add_insert()	No		No	

Table 64. Library function support table (continued)

Function	Enhanced ASCII Support Level	Minimum Value for _ENHANCED_ASCII_EXT Feature Test Macro	Preinitialized Environments for Authorized Programs Support Level	Notes
__le_msg_get()	No		No	
__le_msg_get_and_write()	No		No	
__le_msg_write()	No		No	
__le_traceback()	Neutral		No	
lfind()	Neutral		Yes	
lgamma()	Neutral		Yes	
lgammad32(), lgammad64(), lgammad128()	Neutral		No	
lgammaf()	Neutral		Yes	
lgammal()	Neutral		Yes	
__librel()	Neutral		Yes	
link()	Yes		No	
listen()	Neutral		Yes	8
llabs()	Neutral		Yes	
lldiv()	Neutral		Yes	
llroundd32(), llroundd64(), llroundd128()	Neutral		No	
lltoa()	Yes		Yes	
localdtconv()	No		No	
localeconv()	Yes		No	
localtime(), localtime64()	Yes		Yes	
localtime_r(), localtime64_r()	Yes		Yes	
lockf()	Neutral		No	
log()	Neutral		Yes	
logb()	Neutral		Yes	
logbd32(), logbd64(), logbd128()	Neutral		No	
logd32(), logd64(), logd128()	Neutral		No	
log1pd32() log1pd64() log1pd128()	Neutral		No	
log2d32() log2d64() log2d128()	Neutral		No	
log10d32(), log10d64(), log10d128()	Neutral		No	
logf()	Neutral		Yes	
__login(), __login_applid()	Yes		No	

Table 64. Library function support table (continued)

Function	Enhanced ASCII Support Level	Minimum Value for _ENHANCED_ASCII_EXT Feature Test Macro	Preinitialized Environments for Authorized Programs Support Level	Notes
logl()	Neutral		Yes	
log1p()	Neutral		Yes	
log1pf()	Neutral		Yes	
log1pl()	Neutral		Yes	
log10()	Neutral		Yes	
log10f()	Neutral		Yes	
log10l()	Neutral		Yes	
log2()	Neutral		Yes	
log2f()	Neutral		Yes	
log2l()	Neutral		Yes	
longjmp()	Neutral		Yes	
_longjmp()	Neutral		No	
lrand48()	Neutral		Yes	
lrintd32(), lrintd64(), lrintd128() and llrintd32(), llrintd64(), llrintd128()	Neutral		No	
lround()	Neutral		Yes	
lroundd32(), lroundd64(), lroundd128()	Neutral		No	
lroundf()	Neutral		Yes	
lsearch()	Neutral		Yes	
lseek()	Neutral		No	
lstat()	Yes		No	
l64a()	Yes		Yes	
ltoa()	Yes		Yes	
m_create_layout()	No		Yes	5
m_destroy_layout()	No		Yes	
m_getvalues_layout()	No		Yes	
m_setvalues_layout()	No		Yes	
m_transform_layout()	No		Yes	
m_wtransform_layout()	No		Yes	
makecontext()	Neutral		No	
malloc()	Neutral		Yes	
__malloc24()	Neutral		Yes	

Table 64. Library function support table (continued)

Function	Enhanced ASCII Support Level	Minimum Value for _ENHANCED_ASCII_EXT Feature Test Macro	Preinitialized Environments for Authorized Programs Support Level	Notes
__malloc31()	Neutral		Yes	
__map_init()	Neutral		No	
__map_service()	Neutral		No	
maxcoll()	No		Yes	
maxdesc()	Neutral		No	
mblen()	Yes		Yes	
mbrlen()	No		Yes	
mbrtoc16()	Neutral		No	
mbrtoc32()	Neutral		No	
mbrtowc()	No		Yes	
mbsinit()	No		Yes	
mbsrtowcs()	No		Yes	
mbstowcs()	Yes		Yes	
mbtowc()	Yes		Yes	
memccpy()	Neutral		Yes	
memchr()	Neutral		Yes	
memcmp()	Neutral		Yes	
memcpy()	Neutral		Yes	
memmove()	Neutral		Yes	
memset()	Neutral		Yes	
mkdir()	Yes		No	
mkfifo()	Yes		No	
mknod()	Yes		No	
mkstemp()	Yes		No	
mktemp()	Yes		No	
mktime(), mktime64()	Yes		Yes	
__mlockall()	Neutral		No	
mmap()	Neutral		No	
modf()	Neutral		Yes	
modfd32(), modfd64(), modfd128()	Neutral		No	
modff()	Neutral		Yes	
modfl()	Neutral		Yes	

Table 64. Library function support table (continued)

Function	Enhanced ASCII Support Level	Minimum Value for _ENHANCED_ASCII_EXT Feature Test Macro	Preinitialized Environments for Authorized Programs Support Level	Notes
moservices()	Neutral		Yes	
mount()	Yes		No	
__mount()	No		No	
mprotect()	Neutral		No	
mrand48()	Neutral		Yes	
msgctl()	Neutral		No	
msgget()	Neutral		No	
msgrcv()	Yes		No	
__msgrcv_timed()	No		No	
msgsnd()	Yes		No	
msgxrcv()	Yes		No	
msync()	Neutral		No	
munmap()	Neutral		No	
__must_stay_clean()	Neutral		No	
nan(), nanf(), nanl()	Yes	0x410A0000	No	
nand32(), nand64(), nand128()	Yes	0x410A0000	No	
nearbyintd32(), nearbyintd64(), nearbyintd128()	Neutral		No	
nextafter()	Neutral		Yes	
nextafterd32(), nextafterd64(), nextafterd128()	Neutral		No	
nexttowardd32(), nexttowardd64(), nexttowardd128()	Neutral		No	
nftw()	Yes		No	
nice()	Neutral		No	
nl_langinfo()	Yes		Yes	
nlist()	Yes		No	
nrand48()	Neutral		Yes	
ntohl()	Neutral		No	
ntohs()	Neutral		No	
open()	Yes		No	
__open_stat()	Yes		No	
opendir()	Yes		No	

Table 64. Library function support table (continued)

Function	Enhanced ASCII Support Level	Minimum Value for _ENHANCED_ASCII_EXT Feature Test Macro	Preinitialized Environments for Authorized Programs Support Level	Notes
__opendir2()	Yes		No	
openlog()	Neutral		No	
__osname()	Yes		Yes	
__passwd(), __passwd_applid()	Yes		No	
pathconf()	Yes		No	
pause()	Neutral		No	
pclose()	Neutral		No	
perror(), perror_unlocked()	Yes		No	
__pid_affinity()	Neutral		No	
pipe()	Neutral		No	
__poe	Neutral		No	
poll()	Neutral		No	
popen()	Yes		No	
posix_openpt()	Neutral		No	
pow()	Neutral		Yes	
powd32(), powd64(), powd128()	Neutral		No	
powf()	Neutral		Yes	
powl()	Neutral		Yes	
pread()	Neutral		No	
printf(), printf_unlocked()	Yes		No	
pselect()	Neutral		No	
pthread_atfork()	Neutral		No	
pthread_attr_destroy()	Neutral		No	
pthread_attr_getdetachstate()	Neutral		No	
pthread_attr_getguardsize()	Neutral		No	
pthread_attr_getschedparam()	Neutral		No	
pthread_attr_getstack()	Neutral		No	
pthread_attr_getstackaddr()	Neutral		No	
pthread_attr_getstacksize()	Neutral		No	
pthread_attr_getsynctype_np()	Neutral		No	
pthread_attr_getweight_np()	Neutral		No	
pthread_attr_init()	Neutral		No	

Table 64. Library function support table (continued)

Function	Enhanced ASCII Support Level	Minimum Value for _ENHANCED_ASCII_EXT Feature Test Macro	Preinitialized Environments for Authorized Programs Support Level	Notes
pthread_attr_setdetachstate()	Neutral		No	
pthread_attr_setguardsize()	Neutral		No	
pthread_attr_setschedparam()	Neutral		No	
pthread_attr_setstack()	Neutral		No	
pthread_attr_setstackaddr()	Neutral		No	
pthread_attr_setstacksize()	Neutral		No	
pthread_attr_setsynctype_np()	Neutral		No	
pthread_attr_setweight_np()	Neutral		No	
pthread_cancel()	Neutral		No	
pthread_cleanup_pop()	Neutral		No	
pthread_cleanup_push()	Neutral		No	
pthread_cond_broadcast()	Neutral		No	
pthread_cond_destroy()	Neutral		No	
pthread_cond_init()	Neutral		No	
pthread_cond_signal()	Neutral		No	
pthread_cond_timedwait()	Neutral		No	
pthread_cond_wait()	Neutral		No	
pthread_condattr_destroy()	Neutral		No	
pthread_condattr_getkind_np()	Neutral		No	
pthread_condattr_getpshared()	Neutral		No	
pthread_condattr_init()	Neutral		No	
pthread_condattr_setkind_np()	Neutral		No	
pthread_condattr_setpshared()	Neutral		No	
pthread_create()	Neutral		No	
pthread_detach()	Neutral		No	
pthread_equal()	Neutral		No	
pthread_exit()	Neutral		No	
pthread_getconcurrency()	Neutral		No	
pthread_getspecific()	Neutral		No	
pthread_getspecific_d8_np()	Neutral		No	
pthread_join()	Neutral		No	
pthread_join_d4_np()	Neutral		No	

Table 64. Library function support table (continued)

Function	Enhanced ASCII Support Level	Minimum Value for _ENHANCED_ASCII_EXT Feature Test Macro	Preinitialized Environments for Authorized Programs Support Level	Notes
pthread_key_create()	Neutral		No	
pthread_key_delete()	Neutral		No	
pthread_kill()	Neutral		No	
pthread_mutex_destroy()	Neutral		No	
pthread_mutex_init()	Neutral		No	
pthread_mutex_lock()	Neutral		No	
pthread_mutex_trylock()	Neutral		No	
pthread_mutex_unlock()	Neutral		No	
pthread_mutexattr_destroy()	Neutral		No	
pthread_mutexattr_getkind_np()	Neutral		No	
pthread_mutexattr_getpshared()	Neutral		No	
pthread_mutexattr_gettype()	Neutral		No	
pthread_mutexattr_init()	Neutral		No	
pthread_mutexattr_setkind_np()	Neutral		No	
pthread_mutexattr_setpshared()	Neutral		No	
pthread_mutexattr_settype()	Neutral		No	
pthread_once()	Neutral		No	
pthread_rwlock_destroy()	Neutral		No	
pthread_rwlock_init()	Neutral		No	
pthread_rwlock_rdlock()	Neutral		No	
pthread_rwlock_tryrdlock()	Neutral		No	
pthread_rwlock_trywrlock()	Neutral		No	
pthread_rwlock_unlock()	Neutral		No	
pthread_rwlock_wrlock()	Neutral		No	
pthread_rwlockattr_destroy()	Neutral		No	
pthread_rwlockattr_getpshared()	Neutral		No	
pthread_rwlockattr_init()	Neutral		No	
pthread_rwlockattr_setpshared()	Neutral		No	
pthread_security_np(), pthread_security_applid_np()	Yes		No	
pthread_self()	Neutral		No	
pthread_setcancelstate()	Neutral		No	
pthread_setcanceltype()	Neutral		No	

Table 64. Library function support table (continued)

Function	Enhanced ASCII Support Level	Minimum Value for _ENHANCED_ASCII_EXT Feature Test Macro	Preinitialized Environments for Authorized Programs Support Level	Notes
pthread_setconcurrency()	Neutral		No	
pthread_set_limit_np()	Neutral		No	
pthread_setintr()	Neutral		No	
pthread_setintrtype()	Neutral		No	
pthread_setspecific()	Neutral		No	
pthread_sigmask()	Neutral		No	
pthread_tag_np()	No		No	
pthread_testcancel()	Neutral		No	
pthread_testintr()	Neutral		No	
pthread_yield()	Neutral		No	
ptsname()	Yes		No	
putc()	Yes		No	
putc_unlocked()	Yes		No	
putchar()	Yes		No	
putchar_unlocked()	Yes		No	
putenv()	Yes		Yes	
putmsg()	No		No	
putpmsg()	No		No	
puts(), puts_unlocked()	Yes		No	
pututxline()	No		No	
putw()	Neutral		No	
putwc(), putwc_unlocked()	Yes		No	
putwchar(), putwchar_unlocked()	Yes		No	
pwrite()	Neutral		No	
qsort()	Neutral		Yes	
quantexpd32() quantexpd64() quantexpd128()	Neutral		No	
quantized32(), quantized64(), quantized128()	Neutral		No	
QueryMetrics()	Yes		No	
QuerySchEnv()	Yes		No	
QueryWorkUnitClassification()	No		No	
read()	Neutral		No	

Table 64. Library function support table (continued)

Function	Enhanced ASCII Support Level	Minimum Value for _ENHANCED_ASCII_EXT Feature Test Macro	Preinitialized Environments for Authorized Programs Support Level	Notes
raise()	Neutral		Yes	6
rand()	Neutral		Yes	
rand_r()	Neutral		Yes	
random()	Neutral		Yes	
readdir()	Yes		No	
readdir_r()	Yes		No	
__readdir2()	Yes		No	
readlink()	Yes		No	
readv()	Neutral		No	
realloc()	Neutral		Yes	
realpath()	Yes		No	
re_comp()	No		No	
recv()	No		Yes	8
recvfrom()	Yes	0x41020010	No	2,3
recvmsg()	Yes	0x41020010	No	2,3
re_exec()	No		No	
regcmp()	No		No	
regcomp()	Yes		Yes	
regerror()	Yes		Yes	
regex()	No		No	
regexec()	Yes		Yes	
regfree()	Yes		Yes	
release()	Neutral		Yes	
remainder()	Neutral		Yes	
remainderd32(), remainderd64(), remainderd128()	Neutral		No	
remainderf()	Neutral		Yes	
remainderl()	Neutral		Yes	
remove()	Yes		No	
remque()	Neutral		Yes	
remquo()	Neutral		Yes	
__remquod32() __remquod64() __remquod128()	Neutral		No	

Table 64. Library function support table (continued)

Function	Enhanced ASCII Support Level	Minimum Value for _ENHANCED_ASCII_EXT Feature Test Macro	Preinitialized Environments for Authorized Programs Support Level	Notes
remquof()	Neutral		Yes	
remquol()	Neutral		Yes	
rename()	Yes		No	
__reset_exception_handler()	No		No	
res_init()	Yes		No	
res_mkquery()	Yes	0x41020010	No	
res_query()	Yes	0x41020010	No	
res_querydomain()	Yes	0x41020010	No	
res_search()	Yes	0x41020010	No	
res_send()	Yes	0x41020010	No	
rewind(), rewind_unlocked()	Neutral		No	
rewinddir()	Neutral		No	
rexec()	Yes	0x41060000	No	
rexec_af()	Yes	0x41060000	No	
rindex()	Neutral		Yes	
rint()	Neutral		Yes	
rintd32(), rintd64(), rintd128()	Neutral		No	
rmdir()	Yes		No	
roundd32(), roundd64(), roundd128()	Neutral		No	
rpmatch()	Yes		No	
samequantumd32(), samequantumd64(), samequantumd128()	Neutral		No	
sbrk()	Neutral		No	
scalb()	Neutral		Yes	
scalbn()	Neutral		Yes	
scalbnd32(), scalbnd64(), scalbnd128() and scalblnd32(), scalblnd64(), scalblnd128()	Neutral		No	
scanf(), scanf_unlocked()	Yes		No	
sched_yield()	Neutral		No	
seed48()	Neutral		Yes	
seekdir()	Neutral		No	

Table 64. Library function support table (continued)

Function	Enhanced ASCII Support Level	Minimum Value for _ENHANCED_ASCII_EXT Feature Test Macro	Preinitialized Environments for Authorized Programs Support Level	Notes
select()	No		Yes	8
selectex()	No		No	
semctl()	Neutral		No	
semget()	Neutral		No	
semop()	Neutral		No	
__semop_timed()	Neutral		No	
send()	No		Yes	8
send_file()	No		No	
sendmsg()	Yes	0x41020010	No	2,3
sendto()	Yes	0x41020010	No	2,3
__server_classify()	Yes		No	
__server_classify_create()	Neutral		No	
__server_classify_destroy()	Neutral		No	
__server_classify_reset()	Neutral		No	
__server_init()	Yes		No	
__server_pwu()	Yes		No	
__server_threads_query()	Neutral		No	
setbuf()	Neutral		No	
setcontext()	Neutral		No	
setegid()	Neutral		No	
setenv()	Yes		Yes	
seteuid()	Neutral		No	
__set_exception_handler()	Neutral		No	
setgid()	Neutral		No	
setgrent()	Neutral		No	
setgroups()	Neutral		No	
sethostent()	Neutral		No	
setibmopt()	No		No	
setibmsockopt()	No		No	
setipv4sourcefilter()	Neutral		No	
setitimer()	Neutral		No	
setjmp()	Neutral		Yes	

Table 64. Library function support table (continued)

Function	Enhanced ASCII Support Level	Minimum Value for _ENHANCED_ASCII_EXT Feature Test Macro	Preinitialized Environments for Authorized Programs Support Level	Notes
_setjmp()	Neutral		No	
setkey()	No		Yes	
setlocale()	Yes		Yes	5
setlogmask()	Neutral		Yes	
setnetent()	Neutral		No	
set_new_handler()	Neutral		Yes	
setpeer()	No		No	
setpgid()	Neutral		No	
setpgrp()	Neutral		No	
setpriority()	Neutral		No	
setprotoent()	Neutral		No	
setpwent()	Neutral		No	
setregid()	Neutral		No	
setreuid()	Neutral		No	
setrlimit()	Neutral		No	
setservent()	Neutral		No	
setsid()	Neutral		No	
setsockopt()	No		Yes	8
setsourcefilter()	Neutral		No	
setstate()	No		Yes	
set_terminate()	Neutral		Yes	
_SET_THLIIPADDR()	Neutral		No	
setuid()	Neutral		No	
set_unexpected()	Neutral		Yes	
setutxent()	Neutral		No	
setvbuf()	Neutral		No	
shmat()	Neutral		No	
shmctl()	Neutral		No	
shmdt()	Neutral		No	
shmget()	Neutral		No	
shutdown()	Neutral		No	
__shutdown_registration()	Neutral		No	

Table 64. Library function support table (continued)

Function	Enhanced ASCII Support Level	Minimum Value for _ENHANCED_ASCII_EXT Feature Test Macro	Preinitialized Environments for Authorized Programs Support Level	Notes
sigaction()	Neutral		No	
__sigactionset()	Neutral		No	
sigaddset()	Neutral		No	
sigaltstack()	Neutral		No	
sigdelset()	Neutral		No	
sigemptyset()	Neutral		No	
sigfillset()	Neutral		No	
sighold()	Neutral		No	
sigignore()	Neutral		No	
siginterrupt()	Neutral		No	
sigismember()	Neutral		No	
siglongjmp()	Neutral		No	
signal()	Neutral		No	
__signgam()	Neutral		No	
sigpause()	Neutral		No	
sigpending()	Neutral		No	
sigprocmask()	Neutral		No	
sigqueue()	Neutral		No	
sigrelse()	Neutral		No	
sigset()	Neutral		No	
sigsetjmp()	Neutral		No	
sigstack()	Neutral		No	
sigsuspend()	Neutral		No	
sigtimedwait()	Neutral		No	
sigwait()	Neutral		No	
sigwaitinfo()	Neutral		No	
sin()	Neutral		Yes	
sind32(), sind64(), sind128()	Neutral		No	
sinf()	Neutral		Yes	
sinh()	Neutral		Yes	
sinhd32(), sinhd64(), sinhd128()	Neutral		No	
sinhf()	Neutral		Yes	

Table 64. Library function support table (continued)

Function	Enhanced ASCII Support Level	Minimum Value for _ENHANCED_ASCII_EXT Feature Test Macro	Preinitialized Environments for Authorized Programs Support Level	Notes
sinhl()	Neutral		Yes	
sinl()	Neutral		Yes	
__sinpid32(), __sinpid64(), __sinpid128()	Neutral		No	
sleep()	Neutral		No	
__smf_record()	No		No	
snprintf()	Yes		Yes	
sockatmark()	Neutal		No	
sock_debug()	Neutral		No	
sock_debug_bulk_perf0()	Neutral		No	
sock_do_bulkmode()	Neutral		No	
sock_do_teststor()	Neutral		No	
socket()	Neutral		Yes	8
socketpair()	Neutral		No	
spawn()	Yes		No	
spawnp()	Yes		No	
__spawnp2()	Yes		No	
__spawn2()	Yes		No	
sprintf()	Yes		Yes	
sqrt()	Neutral		Yes	
sqrtd32(), sqrtd64(), sqrtd128()	Neutral		No	
sqrtf()	Neutral		Yes	
sqrtl()	Neutral		Yes	
srand()	Neutral		Yes	
srandom()	Neutral		Yes	
srand48()	Neutral		Yes	
sscanf()	Yes		Yes	
stat()	Yes		No	
statvfs()	Yes		No	
step()	No		No	
strcasecmp()	Yes		Yes	
strcat()	Neutral		Yes	
strchr()	Neutral		Yes	

Table 64. Library function support table (continued)

Function	Enhanced ASCII Support Level	Minimum Value for _ENHANCED_ASCII_EXT Feature Test Macro	Preinitialized Environments for Authorized Programs Support Level	Notes
strcmp()	Neutral		Yes	
strcoll()	Yes		Yes	
strcpy()	Neutral		Yes	
strcspn()	Neutral		Yes	
strdup()	Neutral		Yes	
strerror()	Yes		Yes	
strerror_r()	Yes		No	
strfmon()	Yes		Yes	
strftime()	Yes		Yes	
strlen()	Neutral		Yes	
strncasecmp()	Yes		Yes	
strncat()	Neutral		Yes	
strncmp()	Neutral		Yes	
strncpy()	Neutral		Yes	
strpbrk()	Neutral		Yes	
strptime()	Yes		Yes	
strrchr()	Neutral		Yes	
strspn()	Neutral		Yes	
strstr()	Neutral		Yes	
strtocoll()	No		Yes	
strtod()	Yes		Yes	
strtof()	Yes		No	
strtoimax()	Yes		No	
strtok()	Neutral		Yes	
strtok_r()	Neutral		Yes	
strtod32(), strtod64(), strtod128()	Yes		No	
strtol()	Yes		Yes	
strtold()	Yes		No	
strtoll()	Yes		Yes	
strtoul()	Yes		Yes	
strtoull()	Yes		Yes	
strtoumax()	Yes		No	

Table 64. Library function support table (continued)

Function	Enhanced ASCII Support Level	Minimum Value for _ENHANCED_ASCII_EXT Feature Test Macro	Preinitialized Environments for Authorized Programs Support Level	Notes
strxfrm()	Yes		Yes	
__superkill()	Neutral		No	
svc99()	No		No	
swab()	Neutral		Yes	
swapcontext()	Neutral		No	
swprintf()	Yes		Yes	
swscanf()	Yes		Yes	
symlink()	Yes		No	
sync()	Neutral		No	
sysconf()	Neutral		No	
syslog()	Yes		No	
system()	Yes		No	
t_accept()	No		No	
t_alloc()	No		No	
t_bind()	No		No	
tcgetattr()	Yes	0x41020010	No	
__tcgetcp()	Yes	0x41020010	No	
t_close()	Neutral		No	
t_connect()	No		No	
tcsetattr()	Yes	0x41020010	No	
__tcsetcp()	Yes	0x41020010	No	
t_error()	No		No	
t_free()	No		No	
t_getinfo()	No		No	
t_getprotaddr()	No		No	
t_getstate()	Neutral		No	
t_listen()	No		No	
t_look()	Neutral		No	
t_open()	No		No	
t_optmgmt()	No		No	
t_rcv()	No		No	
t_rcvconnect()	No		No	

Table 64. Library function support table (continued)

Function	Enhanced ASCII Support Level	Minimum Value for _ENHANCED_ASCII_EXT Feature Test Macro	Preinitialized Environments for Authorized Programs Support Level	Notes
t_rcvdis()	No		No	
t_rcvrel()	Neutral		No	
t_rcvudata()	No		No	
t_rcvuderr()	No		No	
t_snd()	No		No	
t_snddis()	No		No	
t_sndrel()	Neutral		No	
t_sndudata()	No		No	
t_strerror()	No		No	
t_sync()	Neutral		No	
t_unbind()	Neutral		No	
takesocket()	No		No	
tan()	Neutral		Yes	
tand32(), tand64(), tand128()	Neutral		No	
tanf()	Neutral		Yes	
tanh()	Neutral		Yes	
tanhd32(), tanhd64(), tanhd128()	Neutral		No	
tanhf()	Neutral		Yes	
tanhl()	Neutral		Yes	
tanl()	Neutral		Yes	
tcdrain()	Neutral		No	
tcflow()	Neutral		No	
tcflush()	Neutral		No	
tcgetattr()	Yes		No	
__tcgetcp()	Yes		No	
tcgetpgrp()	Neutral		No	
tcgetsid()	Neutral		No	
tcperror()	No		No	
tcsendbreak()	Neutral		No	
tcsetattr()	Yes		No	
__tcsetcp()	Yes		No	
tcsetpgrp()	Neutral		No	

Table 64. Library function support table (continued)

Function	Enhanced ASCII Support Level	Minimum Value for _ENHANCED_ASCII_EXT Feature Test Macro	Preinitialized Environments for Authorized Programs Support Level	Notes
__tcsettables()	No		No	
tdelete()	Neutral		Yes	
telldir()	Neutral		No	
tempnam()	Yes		No	
terminate()	Neutral		No	
tfind()	Neutral		Yes	
tgamma()	Neutral		Yes	
tgammad32(), tgammad64(), tgammad128()	Neutral		No	
tgammaf()	Neutral		Yes	
tgammal()	Neutral		Yes	
time(), time64()	Neutral		Yes	
times()	Neutral		No	
tinit()	No		No	
tmpfile()	Neutral		No	
tmpnam()	Yes		No	
toascii()	Neutral		Yes	
__toCcsid()	Yes		No	
__toCSName()	Yes		No	
tolower()	Yes		Yes	
_tolower()	Yes		No	
toupper()	Yes		Yes	
_toupper()	Yes		No	
towlower()	Yes		Yes	
towupper()	Yes		Yes	
trunc()	Neutral		Yes	
truncate()	Yes		No	
truncd32(), truncd64(), truncd128()	Neutral		No	
truncf()	Neutral		Yes	
truncl()	Neutral		Yes	
tsched()	No		No	
tsearch()	Neutral		Yes	

Table 64. Library function support table (continued)

Function	Enhanced ASCII Support Level	Minimum Value for _ENHANCED_ASCII_EXT Feature Test Macro	Preinitialized Environments for Authorized Programs Support Level	Notes
tsyncro()	Neutral		No	
tterm()	Neutral		No	
ttyname()	Yes		No	
ttyname_r()	Yes		No	
ttyslot()	Neutral		No	
twalk()	Neutral		Yes	
tzset()	Yes		No	
ualarm()	Neutral		No	
__ucreate()	Neutral		No	
__ufree()	Neutral		No	
__uheapreport()	Neutral		No	
ulimit()	Neutral		No	
ulltoa()	Yes		No	
ultoa()	Yes		No	
__umalloc()	Neutral		No	
umask()	Neutral		No	
umount()	Yes		No	
uname()	Yes		No	
uncaught_exception()	Neutral		Yes	
UnDoExportWorkUnit()	Neutral		No	
UnDoImportWorkUnit()	Neutral		No	
unexpected()	Neutral		Yes	
ungetc(), ungetc_unlocked()	Neutral		No	
ungetwc(), ungetwc_unlocked()	Yes		No	
unlink()	Yes		No	
unlockpt()	Neutral		No	
unsetenv()	Yes		No	
usleep()	Neutral		No	
utime()	Yes		No	
utimes()	Yes		No	
__utmpxname()	Neutral		No	
utoa()	Yes		Yes	

Table 64. Library function support table (continued)

Function	Enhanced ASCII Support Level	Minimum Value for _ENHANCED_ASCII_EXT Feature Test Macro	Preinitialized Environments for Authorized Programs Support Level	Notes
va_arg()	Neutral		Yes	
va_copy()	Neutral		Yes	
va_end()	Neutral		Yes	
va_start()	Neutral		Yes	
valloc()	Neutral		No	
vfork()	Neutral		No	
vfprintf(), vfprintf_unlocked()	Yes		No	
vfscanf(), vfscanf_unlocked()	Yes		No	
vfwprintf(), vfwprintf_unlocked()	Yes	0x41070000	No	
vfwscanf(), vfwscanf_unlocked()	Yes		No	
vprintf(), vprintf_unlocked()	Yes		No	
vsnprintf()	Yes		Yes	
vsprintf()	Yes		Yes	
vscanf(), vscanf_unlocked()	Yes		No	
vsscanf()	Yes		Yes	
vswprintf()	Yes		Yes	
vswscanf()	Yes		No	
vwprintf(), vwprintf_unlocked()	Yes	0x41070000	No	
vwscanf(), vwscanf_unlocked()	Yes		No	
w_getmntent()	Yes		No	
w_getpsent()	Yes		No	
w_ioctl()	No		No	
w_statfs()	Yes		No	
w_statvfs()	Yes		No	
wait()	Neutral		No	
waitid()	Neutral		No	
waitpid()	Neutral		No	
wait3()	Neutral		No	
wcrtomb()	No		No	
wcscat()	Neutral		No	
wcschr()	Neutral		No	
wcscmp()	Neutral		No	

Table 64. Library function support table (continued)

Function	Enhanced ASCII Support Level	Minimum Value for _ENHANCED_ASCII_EXT Feature Test Macro	Preinitialized Environments for Authorized Programs Support Level	Notes
wcscoll()	Yes		No	
wcscpy()	Neutral		No	
wcscspn()	Neutral		No	
wcsftime()	Yes		No	
wcsid()	Yes		No	
wcslen()	Neutral		No	
wcsncat()	Neutral		No	
wcsncmp()	Neutral		No	
wcsncpy()	Neutral		No	
wcspbrk()	Neutral		No	
wcsrchr()	Neutral		No	
wcsrtombs()	No		No	
wcsspn()	Neutral		No	
wcsstr()	Neutral		No	
wcstod()	Yes		No	
wcstod32(), wcstod64(), wcstod128()	Yes		No	
wcstof()	Yes		No	
wcstoimax()	Yes		No	
wcstok()	Neutral		No	
wcstol()	Yes		No	
wcstold()	Yes		No	
wcstoll()	Yes	0x41070000	No	
wcstombs()	Yes		No	
wcstoul()	Yes		No	
wcstoull()	Yes	0x41070000	No	
wcstoumax()	Yes		No	
wcswcs()	Neutral		No	
wcswidth()	Yes		No	
wcsxfrm()	Yes		No	
wctob()	Yes		No	
wctomb()	Yes		No	
wctype()	Yes		No	

Table 64. Library function support table (continued)

Function	Enhanced ASCII Support Level	Minimum Value for _ENHANCED_ASCII_EXT Feature Test Macro	Preinitialized Environments for Authorized Programs Support Level	Notes
wcwidth()	Yes		No	
wmemchr()	Neutral		No	
wmemcmp()	Neutral		No	
wmemcpy()	Neutral		No	
wmemmove()	Neutral		No	
wmemset()	Neutral		No	
wordexp()	No		No	
wordfree()	No		No	
__w_pioctl()	Yes	0x41020010	No	
wprintf()	Yes	0x41070000	No	
write()	Neutral		No	
__writedown()	Neutral		No	
writev()	Neutral		No	
wscanf()	Yes		No	
__wsinit()	Neutral		No	
yn()	Neutral		Yes	
y0()	Neutral		Yes	
y1()	Neutral		Yes	

Notes:

1. ASCII support provided only for XPG4 (or higher) interface.
2. ASCII support provided only for X/Open Sockets interface.
3. ASCII support is for the sun_path element of struct sockaddr_un when working with the AF_UNIX address family.
4. Preinitialized Environments for Authorized Programs support provided only for non-posix signals.
5. Preinitialized Environments for Authorized Programs support provided only if locale file resides in a dataset, not in a z/OS UNIX file system.
6. Preinitialized Environments for Authorized Programs support provided only for non-posix form of the function.
7. Preinitialized Environments for Authorized Programs support provided only for non z/OS UNIX file system DLLs.
8. Preinitialized Environments for Authorized Programs support provided only when dispatchable unit mode is task and cross memory mode is PASN=HASN=SASN. In addition, the RECOVERY=ESTAE parameter must be used on the CELAAUTH macro invocation.
9. This function table does not include compiler built-in functions (builtin.h).

Appendix C. Accessibility

Accessible publications for this product are offered through IBM Knowledge Center (http://www.ibm.com/support/knowledgecenter/SSLTBW/welcome).

If you experience difficulty with the accessibility of any z/OS information, send a detailed message to the "Contact us" web page for z/OS (http://www.ibm.com/systems/z/os/zos/webqs.html) or use the following mailing address.

IBM Corporation
Attention: MHVRCFS Reader Comments
Department H6MA, Building 707
2455 South Road
Poughkeepsie, NY 12601-5400
United States

Accessibility features

Accessibility features help users who have physical disabilities such as restricted mobility or limited vision use software products successfully. The accessibility features in z/OS can help users do the following tasks:

- Run assistive technology such as screen readers and screen magnifier software.
- Operate specific or equivalent features by using the keyboard.
- Customize display attributes such as color, contrast, and font size.

Consult assistive technologies

Assistive technology products such as screen readers function with the user interfaces found in z/OS. Consult the product information for the specific assistive technology product that is used to access z/OS interfaces.

Keyboard navigation of the user interface

You can access z/OS user interfaces with TSO/E or ISPF. The following information describes how to use TSO/E and ISPF, including the use of keyboard shortcuts and function keys (PF keys). Each guide includes the default settings for the PF keys.

- *z/OS TSO/E Primer*
- *z/OS TSO/E User's Guide*
- *z/OS V2R2 ISPF User's Guide Vol I*

Dotted decimal syntax diagrams

Syntax diagrams are provided in dotted decimal format for users who access IBM Knowledge Center with a screen reader. In dotted decimal format, each syntax element is written on a separate line. If two or more syntax elements are always present together (or always absent together), they can appear on the same line because they are considered a single compound syntax element.

Each line starts with a dotted decimal number; for example, `3` or `3.1` or `3.1.1`. To hear these numbers correctly, make sure that the screen reader is set to read out punctuation. All the syntax elements that have the same dotted decimal number (for example, all the syntax elements that have the number 3.1) are mutually exclusive alternatives. If you hear the lines 3.1 USERID and 3.1 SYSTEMID, your syntax can include either USERID or SYSTEMID, but not both.

The dotted decimal numbering level denotes the level of nesting. For example, if a syntax element with dotted decimal number 3 is followed by a series of syntax elements with dotted decimal number 3.1, all the syntax elements numbered 3.1 are subordinate to the syntax element numbered 3.

Certain words and symbols are used next to the dotted decimal numbers to add information about the syntax elements. Occasionally, these words and symbols might occur at the beginning of the element itself. For ease of identification, if the word or symbol is a part of the syntax element, it is preceded by the backslash (\) character. The * symbol is placed next to a dotted decimal number to indicate that the syntax element repeats. For example, syntax element *FILE with dotted decimal number 3 is given the format 3 * FILE. Format 3* FILE indicates that syntax element FILE repeats. Format 3* * FILE indicates that syntax element * FILE repeats.

Characters such as commas, which are used to separate a string of syntax elements, are shown in the syntax just before the items they separate. These characters can appear on the same line as each item, or on a separate line with the same dotted decimal number as the relevant items. The line can also show another symbol to provide information about the syntax elements. For example, the lines 5.1*, 5.1 LASTRUN, and 5.1 DELETE mean that if you use more than one of the LASTRUN and DELETE syntax elements, the elements must be separated by a comma. If no separator is given, assume that you use a blank to separate each syntax element.

If a syntax element is preceded by the % symbol, it indicates a reference that is defined elsewhere. The string that follows the % symbol is the name of a syntax fragment rather than a literal. For example, the line 2.1 %OP1 means that you must refer to separate syntax fragment OP1.

The following symbols are used next to the dotted decimal numbers.

? indicates an optional syntax element

The question mark (?) symbol indicates an optional syntax element. A dotted decimal number followed by the question mark symbol (?) indicates that all the syntax elements with a corresponding dotted decimal number, and any subordinate syntax elements, are optional. If there is only one syntax element with a dotted decimal number, the ? symbol is displayed on the same line as the syntax element, (for example 5? NOTIFY). If there is more than one syntax element with a dotted decimal number, the ? symbol is displayed on a line by itself, followed by the syntax elements that are optional. For example, if you hear the lines 5 ?, 5 NOTIFY, and 5 UPDATE, you know that the syntax elements NOTIFY and UPDATE are optional. That is, you can choose one or none of them. The ? symbol is equivalent to a bypass line in a railroad diagram.

! indicates a default syntax element

The exclamation mark (!) symbol indicates a default syntax element. A dotted decimal number followed by the ! symbol and a syntax element indicate that the syntax element is the default option for all syntax elements that share the same dotted decimal number. Only one of the syntax elements that share the dotted decimal number can specify the ! symbol. For example, if you hear the lines 2? FILE, 2.1! (KEEP), and 2.1 (DELETE), you know that (KEEP) is the default option for the FILE keyword. In the example, if you include the FILE keyword, but do not specify an option, the default option KEEP is applied. A default option also applies to the next higher dotted decimal number. In this example, if the FILE keyword is omitted, the default FILE(KEEP) is used. However, if you hear the lines 2? FILE, 2.1, 2.1.1! (KEEP), and 2.1.1 (DELETE), the default option KEEP applies only to the next higher dotted decimal number, 2.1 (which does not have an associated keyword), and does not apply to 2? FILE. Nothing is used if the keyword FILE is omitted.

*** indicates an optional syntax element that is repeatable**
The asterisk or glyph (*) symbol indicates a syntax element that can be repeated zero or more times. A dotted decimal number followed by the * symbol indicates that this syntax element can be used zero or more times; that is, it is optional and can be repeated. For example, if you hear the line 5.1* data area, you know that you can include one data area, more than one data area, or no data area. If you hear the lines 3* , 3 HOST, 3 STATE, you know that you can include HOST, STATE, both together, or nothing.

Notes:

1. If a dotted decimal number has an asterisk (*) next to it and there is only one item with that dotted decimal number, you can repeat that same item more than once.
2. If a dotted decimal number has an asterisk next to it and several items have that dotted decimal number, you can use more than one item from the list, but you cannot use the items more than once each. In the previous example, you can write HOST STATE, but you cannot write HOST HOST.
3. The * symbol is equivalent to a loopback line in a railroad syntax diagram.

+ indicates a syntax element that must be included
The plus (+) symbol indicates a syntax element that must be included at least once. A dotted decimal number followed by the + symbol indicates that the syntax element must be included one or more times. That is, it must be included at least once and can be repeated. For example, if you hear the line 6.1+ data area, you must include at least one data area. If you hear the lines 2+, 2 HOST, and 2 STATE, you know that you must include HOST, STATE, or both. Similar to the * symbol, the + symbol can repeat a particular item if it is the only item with that dotted decimal number. The + symbol, like the * symbol, is equivalent to a loopback line in a railroad syntax diagram.

Notices

This information was developed for products and services offered in the U.S.A. or elsewhere.

IBM may not offer the products, services, or features discussed in this document in other countries. Consult your local IBM representative for information on the products and services currently available in your area. Any reference to an IBM product, program, or service is not intended to state or imply that only that IBM product, program, or service may be used. Any functionally equivalent product, program, or service that does not infringe any IBM intellectual property right may be used instead. However, it is the user's responsibility to evaluate and verify the operation of any non-IBM product, program, or service.

IBM may have patents or pending patent applications covering subject matter described in this document. The furnishing of this document does not give you any license to these patents. You can send license inquiries, in writing, to:

IBM Director of Licensing
IBM Corporation
North Castle Drive
Armonk, NY 10504-1785
U.S.A

For license inquiries regarding double-byte character set (DBCS) information, contact the IBM Intellectual Property Department in your country or send inquiries, in writing, to:

Intellectual Property Licensing
Legal and Intellectual Property Law
IBM Japan, Ltd.
19-21, Nihonbashi-Hakozakicho, Chuo-ku
Tokyo 103-8510, Japan

The following paragraph does not apply to the United Kingdom or any other country where such provisions are inconsistent with local law: INTERNATIONAL BUSINESS MACHINES CORPORATION PROVIDES THIS PUBLICATION “AS IS” WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF NON-INFRINGEMENT, MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. Some states do not allow disclaimer of express or implied warranties in certain transactions, therefore, this statement may not apply to you.

This information could include technical inaccuracies or typographical errors. Changes are periodically made to the information herein; these changes will be incorporated in new editions of the publication. IBM may make improvements and/or changes in the product(s) and/or the program(s) described in this publication at any time without notice.

Any references in this information to non-IBM Web sites are provided for convenience only and do not in any manner serve as an endorsement of those Web sites. The materials at those Web sites are not part of the materials for this IBM product and use of those Web sites is at your own risk.

IBM may use or distribute any of the information you supply in any way it believes appropriate without incurring any obligation to you.

Licensees of this program who wish to have information about it for the purpose of enabling: (i) the exchange of information between independently created programs and other programs (including this one) and (ii) the mutual use of the information which has been exchanged, should contact:

Site Counsel
IBM Corporation
2455 South Road
Poughkeepsie, NY 12601-5400
USA

Such information may be available, subject to appropriate terms and conditions, including in some cases, payment of a fee.

The licensed program described in this information and all licensed material available for it are provided by IBM under terms of the IBM Customer Agreement, IBM International Program License Agreement, or any equivalent agreement between us.

Information concerning non-IBM products was obtained from the suppliers of those products, their published announcements or other publicly available sources. IBM has not tested those products and cannot confirm the accuracy of performance, compatibility or any other claims related to non-IBM products. Questions on the capabilities of non-IBM products should be addressed to the suppliers of those products.

All statements regarding IBM's future direction or intent are subject to change or withdrawal without notice, and represent goals and objectives only.

If you are viewing this information softcopy, the photographs and color illustrations may not appear.

COPYRIGHT LICENSE:

This information might contain sample application programs in source language, which illustrate programming techniques on various operating platforms. You may copy, modify, and distribute these sample programs in any form without payment to IBM, for the purposes of developing, using, marketing or distributing application programs conforming to the application programming interface for the operating platform for which the sample programs are written. These examples have not been thoroughly tested under all conditions. IBM, therefore, cannot guarantee or imply reliability, serviceability, or function of these programs. The sample programs are provided "AS IS", without warranty of any kind. IBM shall not be liable for any damages arising out of your use of the sample programs.

Policy for unsupported hardware

Various z/OS elements, such as DFSMS, HCD, JES2, JES3, and MVS, contain code that supports specific hardware servers or devices. In some cases, this device-related element support remains in the product even after the hardware devices pass their announced End of Service date. z/OS may continue to service element code; however, it will not provide service related to unsupported hardware devices. Software problems related to these devices will not be accepted for service, and current service activity will cease if a problem is determined to be associated with out-of-support devices. In such cases, fixes will not be issued.

Minimum supported hardware

The minimum supported hardware for z/OS releases identified in z/OS announcements can subsequently change when service for particular servers or devices is withdrawn. Likewise, the levels of other software products supported on a particular release of z/OS are subject to the service support lifecycle of those products. Therefore, z/OS and its product publications (for example, panels, samples, messages, and product documentation) can include references to hardware and software that is no longer supported.

- For information about software support lifecycle, see: IBM Lifecycle Support for z/OS (http://www.ibm.com/software/support/systemsz/lifecycle/)
- For information about currently-supported IBM hardware, contact your IBM representative.

Programming interface information

This publication documents *intended* Programming Interfaces that allow the customer to write z/OS XL C/C++ programs.

Standards

The following standards are supported in combination with the Language Environment element:

- The C language is consistent with *Programming languages - C (ISO/IEC 9899:1999)* and a subset of *Programming languages - C (ISO/IEC 9899:2011)*. For more information on ISO, visit their website at http://www.iso.org.
- The C++ language is consistent with *Programming languages - C++ (ISO/IEC 14882:1998)*, *Programming languages - C++ (ISO/IEC 14882:2003(E))*, and a subset of *Programming languages - C++ (ISO/IEC 14882:2011)*.

The following standards are supported in combination with the Language Environment and z/OS UNIX System Services elements:

- A subset of *IEEE Std. 1003.1-2001 (Single UNIX Specification, Version 3)*. For more information on IEEE, visit their website at http://www.iso.org.
- *IEEE Std 1003.1—1990, IEEE Standard Information Technology—Portable Operating System Interface (POSIX)—Part 1: System Application Program Interface (API) [C language]*, copyright 1990 by the Institute of Electrical and Electronic Engineers, Inc.
- The core features of *IEEE P1003.1a Draft 6 July 1991, Draft Revision to Information Technology—Portable Operating System Interface (POSIX), Part 1: System Application Program Interface (API) [C Language]*, copyright 1992 by the Institute of Electrical and Electronic Engineers, Inc.
- *IEEE Std 1003.2—1992, IEEE Standard Information Technology—Portable Operating System Interface (POSIX)—Part 2: Shells and Utilities*, copyright 1990 by the Institute of Electrical and Electronic Engineers, Inc.
- The core features of *IEEE Std P1003.4a/D6—1992, IEEE Draft Standard Information Technology—Portable Operating System Interface (POSIX)—Part 1: System Application Program Interface (API)—Amendment 2: Threads Extension [C language]*, copyright 1990 by the Institute of Electrical and Electronic Engineers, Inc.

- The core features of *IEEE 754-1985 (R1990) IEEE Standard for Binary Floating-Point Arithmetic (ANSI)*, copyright 1985 by the Institute of Electrical and Electronic Engineers, Inc.
- *X/Open CAE Specification, System Interfaces and Headers, Issue 4 Version 2*, copyright 1994 by The Open Group
- *X/Open CAE Specification, Networking Services, Issue 4*, copyright 1994 by The Open Group
- *X/Open Specification Programming Languages, Issue 3, Common Usage C*, copyright 1988, 1989, and 1992 by The Open Group
- United States Government's *Federal Information Processing Standard (FIPS) publication for the programming language C, FIPS-160*, issued by National Institute of Standards and Technology, 1991

Trademarks

IBM, the IBM logo, and ibm.com are trademarks or registered trademarks of International Business Machines Corp., registered in many jurisdictions worldwide. Other product and service names might be trademarks of IBM or other companies. A current list of IBM trademarks is available on the Web at “Copyright and trademark information” at www.ibm.com/legal/copytrade.shtml.

Adobe, Acrobat, PostScript and all Adobe-based trademarks are either registered trademarks or trademarks of Adobe Systems Incorporated in the United States, other countries, or both.

Microsoft, Windows, Windows NT, and the Windows logo are trademarks of Microsoft Corporation in the United States, other countries, or both.

UNIX is a registered trademark of The Open Group in the United States and other countries.

Java and all Java-based trademarks and logos are trademarks or registered trademarks of Oracle and/or its affiliates.

Linux is a trademark of Linus Torvalds in the United States, other countries, or both.

Other company, product, or service names may be trademarks or service marks of others.

Index

Product Number: 5650-ZOS

Printed in USA

SC14-7314-07