IBM i
Version 7.2

Programming
ILE C/C++ Runtime Library Functions

IBM
Note

Before using this information and the product it supports, read the information in “Notices” on page 617.

This edition applies to version IBM i 7.2 (product number 5770-SS1) and to all subsequent releases and modifications until otherwise indicated in new editions. This version does not run on all reduced instruction set computer (RISC) models nor does it run on CISC models.

This document may contain references to Licensed Internal Code. Licensed Internal Code is Machine Code and is licensed to you under the terms of the IBM License Agreement for Machine Code.

US Government Users Restricted Rights – Use, duplication or disclosure restricted by GSA ADP Schedule Contract with IBM Corp.
About ILE C/C++ Runtime Library Functions

This information provides an overview of the C runtime behavior.

The information is intended for programmers who are familiar with the C/C++ programming language and who want to write or maintain ILE C/C++ applications. You must have experience in using applicable IBM® i menus, displays, or control language (CL) commands. You also need knowledge of Integrated Language Environment® as explained in the ILE Concepts manual.

This information does not describe how to program in the C or C++ programming languages, nor does it explain the concepts of ILE. Companion publications for this reference are:

• C/C++ Legacy Class Libraries Reference, SC09-7652-00
• ILE Concepts
• ILE C/C++ for AS/400 MI Library Reference, SC09-2418-00
• Standard C/C++ Library Reference, SC09-4949-01
• ILE C/C++ Compiler Reference
• ILE C/C++ Language Reference
• ILE C/C++ Programmer's Guide

For other prerequisite and related information, see “Related information” on page 615.

A note about examples

The examples in this information that illustrate the use of library functions are written in a simple style. The examples do not demonstrate all possible uses of C/C++ language constructs. Some examples are only code fragments and do not compile without additional code. The examples all assume that the C locale is used.

All complete runnable examples for library functions and machine interface instructions are in library QCPPLE, in source file QACSRC. Each example name is the same as the function name or instruction name. For example, the source code for the example illustrating the use of the _Rcommit() function in this information is in library QCPPLE, file QACSRC, member RCOMMIT. The QSYSINC library must be installed.
## Contents

- About ILE C/C++ Runtime Library Functions ................................................................. iii
- What's new .................................................................................................................. xiii

### Include Files ........................................................................................................ 1
- `<assert.h>` ............................................................................................................. 1
- `<ctype.h>` ............................................................................................................. 1
- `<decimal.h>` ......................................................................................................... 1
- `<errno.h>` ............................................................................................................. 2
- `<except.h>` .......................................................................................................... 2
- `<float.h>` ............................................................................................................. 4
- `<inttypes.h>` ....................................................................................................... 5
- `<langinfo.h>` ....................................................................................................... 5
- `<limits.h>` ........................................................................................................... 5
- `<locale.h>` .......................................................................................................... 5
- `<math.h>` ............................................................................................................. 6
- `<mallocinfo.h>` ................................................................................................... 6
- `<monetary.h>` ...................................................................................................... 6
- `<nl_types.h>` ....................................................................................................... 7
- `<pointer.h>` ........................................................................................................ 7
- `<recio.h>` ............................................................................................................ 7
- `<regex.h>` .......................................................................................................... 10
- `<setjmp.h>` ......................................................................................................... 11
- `<signal.h>` .......................................................................................................... 11
- `<stdarg.h>` .......................................................................................................... 11
- `<stddef.h>` .......................................................................................................... 11
- `<stdbool.h>` ........................................................................................................ 12
- `<stdio.h>` ........................................................................................................... 12
- `<stdlib.h>` ......................................................................................................... 12
- `<stdint.h>` ......................................................................................................... 13
- `<stdlib.h>` .......................................................................................................... 15
- `<string.h>` .......................................................................................................... 15
- `<strings.h>` ....................................................................................................... 16
- `<time.h>` ............................................................................................................. 16
- `<wchar.h>` .......................................................................................................... 16
- `<wctype.h>` ......................................................................................................... 17
- `<xxcvt.h>` ........................................................................................................... 17
- `<xxdtaa.h>` ......................................................................................................... 17
- `<xxenv.h>` .......................................................................................................... 18
- `<xxfbk.h>` ........................................................................................................... 18

### Machine Interface (MI) Include Files ................................................................... 18

### Library Functions ................................................................................................. 19
- The C/C++ Library ................................................................................................. 19
  - Error Handling .................................................................................................... 19
  - Searching and Sorting ...................................................................................... 20
  - Mathematical ................................................................................................... 20
  - Time Manipulation ............................................................................................ 26
  - Type Conversion ............................................................................................... 28
  - Conversion ....................................................................................................... 34
  - Record Input/Output ......................................................................................... 34
<table>
<thead>
<tr>
<th>Function</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>abort() — Stop a Program.</td>
<td>60</td>
</tr>
<tr>
<td>abs() — Calculate Integer Absolute Value.</td>
<td>61</td>
</tr>
<tr>
<td>acos() — Calculate Arccosine.</td>
<td>62</td>
</tr>
<tr>
<td>asctime() — Convert Time to Character String.</td>
<td>63</td>
</tr>
<tr>
<td>asctime_r() — Convert Time to Character String (Restartable).</td>
<td>65</td>
</tr>
<tr>
<td>asin() — Calculate Arcsine.</td>
<td>66</td>
</tr>
<tr>
<td>assert() — Verify Condition.</td>
<td>68</td>
</tr>
<tr>
<td>atan() — atan2() — Calculate Arctangent.</td>
<td>69</td>
</tr>
<tr>
<td>atexit() — Record Program Ending Function.</td>
<td>70</td>
</tr>
<tr>
<td>atof() — Convert Character String to Float.</td>
<td>71</td>
</tr>
<tr>
<td>atoi() — Convert Character String to Integer.</td>
<td>72</td>
</tr>
<tr>
<td>atol() — atoll() — Convert Character String to Long or Long Long Integer.</td>
<td>74</td>
</tr>
<tr>
<td>Bessel Functions.</td>
<td>75</td>
</tr>
<tr>
<td>bsearch() — Search Arrays.</td>
<td>76</td>
</tr>
<tr>
<td>btoe() — Convert Single Byte to Wide Character.</td>
<td>78</td>
</tr>
<tr>
<td>_C_Get_Ssn_Handle() — Handle to C Session.</td>
<td>79</td>
</tr>
<tr>
<td>calloc() — Reserve and Initialize Storage.</td>
<td>80</td>
</tr>
<tr>
<td>catclose() — Close Message Catalog.</td>
<td>81</td>
</tr>
<tr>
<td>catgets() — Retrieve a Message from a Message Catalog.</td>
<td>82</td>
</tr>
<tr>
<td>catopen() — Open Message Catalog.</td>
<td>84</td>
</tr>
<tr>
<td>ceil() — Find Integer &gt;=Argument.</td>
<td>86</td>
</tr>
<tr>
<td>clearerr() — Reset Error Indicators.</td>
<td>87</td>
</tr>
<tr>
<td>clock() — Determine Processor Time.</td>
<td>88</td>
</tr>
<tr>
<td>cos() — Calculate Cosine.</td>
<td>89</td>
</tr>
<tr>
<td>cosh() — Calculate Hyperbolic Cosine.</td>
<td>90</td>
</tr>
<tr>
<td>_C_Quickpool_Debug() — Modify Quick Pool Memory Manager Characteristics.</td>
<td>91</td>
</tr>
<tr>
<td>_C_Quickpool_Init() — Initialize Quick Pool Memory Manager.</td>
<td>93</td>
</tr>
<tr>
<td>_C_Quickpool_Report() — Generate Quick Pool Memory Manager Report.</td>
<td>95</td>
</tr>
<tr>
<td>ctime() — Convert Time to Character String.</td>
<td>96</td>
</tr>
<tr>
<td>ctime64() — Convert Time to Character String.</td>
<td>98</td>
</tr>
<tr>
<td>ctime_r() — Convert Time to Character String (Restartable).</td>
<td>100</td>
</tr>
<tr>
<td>ctime64_r() — Convert Time to Character String (Restartable).</td>
<td>101</td>
</tr>
<tr>
<td>_C_TS_malloc_debug() — Determine amount of terspace memory used (with optional dumps and verification).</td>
<td>103</td>
</tr>
<tr>
<td>_C_TS_malloc_info() — Determine amount of terspace memory used.</td>
<td>105</td>
</tr>
<tr>
<td>diff() — Compute Quotient and Remainder.</td>
<td>109</td>
</tr>
<tr>
<td>diff() — Compute Time Difference.</td>
<td>108</td>
</tr>
<tr>
<td>diff() — Compute Time Difference.</td>
<td>108</td>
</tr>
<tr>
<td>div() — Calculate Quotient and Remainder.</td>
<td>111</td>
</tr>
<tr>
<td>erf() — erfc() — Calculate Error Functions.</td>
<td>112</td>
</tr>
<tr>
<td>exit() — End Program.</td>
<td>113</td>
</tr>
<tr>
<td>exp() — Calculate Exponential Function.</td>
<td>114</td>
</tr>
<tr>
<td>fabs() — Calculate Floating-Point Absolute Value.</td>
<td>115</td>
</tr>
<tr>
<td>fclose() — Close Stream.</td>
<td>116</td>
</tr>
<tr>
<td>fopen() — Associates Stream With File Descriptor</td>
<td>117</td>
</tr>
<tr>
<td>Function</td>
<td>Description</td>
</tr>
<tr>
<td>----------------------------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>rename()</td>
<td>Rename File</td>
</tr>
<tr>
<td>remove()</td>
<td>Delete File</td>
</tr>
<tr>
<td>regerror()</td>
<td>Return Error Message for Regular Expression</td>
</tr>
<tr>
<td>regcomp()</td>
<td>Compile Regular Expression</td>
</tr>
<tr>
<td>_Rdelete()</td>
<td>Delete a Record</td>
</tr>
<tr>
<td>_Rclose()</td>
<td>Close a File</td>
</tr>
<tr>
<td>rand() – rand_r()</td>
<td>Generate Random Number</td>
</tr>
<tr>
<td>QXXZTOD()</td>
<td>Convert Zoned Decimal to Double</td>
</tr>
<tr>
<td>QXXRTVDA()</td>
<td>Retrieve Data Area</td>
</tr>
<tr>
<td>QXXPTOI()</td>
<td>Convert Packed Decimal to Integer</td>
</tr>
<tr>
<td>QXXPTOD()</td>
<td>Convert Packed Decimal to Double</td>
</tr>
<tr>
<td>QXXITOZ()</td>
<td>Convert Integer to Zoned Decimal</td>
</tr>
<tr>
<td>QXXITOP()</td>
<td>Convert Integer to Packed Decimal</td>
</tr>
<tr>
<td>QXXDTOZ()</td>
<td>Convert Double to Zoned Decimal</td>
</tr>
<tr>
<td>qsort()</td>
<td>Sort Array</td>
</tr>
<tr>
<td>QXXCHGDA()</td>
<td>Change Data Area</td>
</tr>
<tr>
<td>QXXDTODP()</td>
<td>Convert Double to Packed Decimal</td>
</tr>
<tr>
<td>QXXDTOZI()</td>
<td>Convert Double to Zoned Decimal</td>
</tr>
<tr>
<td>QXXITOP()</td>
<td>Convert Integer to Packed Decimal</td>
</tr>
<tr>
<td>QXXITOI()</td>
<td>Convert Integer to Zoned Decimal</td>
</tr>
<tr>
<td>QXXPTODI()</td>
<td>Convert Packed Decimal to Double</td>
</tr>
<tr>
<td>QXXPTOI()</td>
<td>Convert Packed Decimal to Integer</td>
</tr>
<tr>
<td>QXXRTVDA()</td>
<td>Retrieve Data Area</td>
</tr>
<tr>
<td>QXXZTODI()</td>
<td>Convert Zoned Decimal to Double</td>
</tr>
<tr>
<td>QXXZTOIP()</td>
<td>Convert Zoned Decimal to Integer</td>
</tr>
<tr>
<td>raise()</td>
<td>Send Signal</td>
</tr>
<tr>
<td>rand() – rand_r()</td>
<td>Generate Random Number</td>
</tr>
<tr>
<td>__Racquire()</td>
<td>Acquire a Program Device</td>
</tr>
<tr>
<td>__Rclose()</td>
<td>Close a File</td>
</tr>
<tr>
<td>__Rcommit()</td>
<td>Commit Current Record</td>
</tr>
<tr>
<td>__Rdelete()</td>
<td>Delete a Record</td>
</tr>
<tr>
<td>__Rdevatr()</td>
<td>Get Device Attributes</td>
</tr>
<tr>
<td>realloc()</td>
<td>Change Reserved Storage Block Size</td>
</tr>
<tr>
<td>regcomp()</td>
<td>Compile Regular Expression</td>
</tr>
<tr>
<td>regerror()</td>
<td>Return Error Message for Regular Expression</td>
</tr>
<tr>
<td>regexec()</td>
<td>Execute Compiled Regular Expression</td>
</tr>
<tr>
<td>regfree()</td>
<td>Free Memory for Regular Expression</td>
</tr>
<tr>
<td>remove()</td>
<td>Delete File</td>
</tr>
<tr>
<td>rename()</td>
<td>Rename File</td>
</tr>
</tbody>
</table>
rewind() — Adjust Current File Position..................................................................................303
__Redo() — Force the End-of-Data..........................................................................................304
__RedoV() — Force the End-of-File.......................................................................................305
__Rformat() — Set the Record Format Name.......................................................................306
__Rindara() — Set Separate Indicator Area..........................................................................308
__Riofbk() — Obtain I/O Feedback Information...................................................................310
__Rlocate() — Position a Record..........................................................................................312
__Ropen() — Open a Record File for I/O Operations..............................................................315
__Rpnfbk() — Obtain Open Feedback Information...............................................................319
__Rpgmdev() — Set Default Program Device.......................................................................320
__Rreadd() — Read a Record by Relative Record Number.....................................................321
__Rreadf() — Read the First Record.....................................................................................323
__Rreadindv() — Read from an Invited Device.....................................................................325
__Rreadk() — Read a Record by Key....................................................................................328
__Rreadl() — Read the Last Record......................................................................................331
__Rreadn() — Read the Next Record.....................................................................................332
__Rreadnc() — Read the Next Changed Record in a Subfile....................................................334
__Rreadp() — Read the Previous Record..............................................................................337
__Rreadr() — Read the Same Record....................................................................................339
__Rrelease() — Release a Program Device...........................................................................341
__Rlslck() — Release a Record Lock.....................................................................................342
__Rrollbk() — Roll Back Commitment Control Changes.......................................................343
__Rupdate() — Update a Record..........................................................................................345
__Rupfb() — Provide Information on Last I/O Operation.....................................................346
__Rwrite() — Write the Next Record...................................................................................348
__Rwrited() — Write a Record Directly................................................................................350
__Rwritedr() — Write and Read a Record...........................................................................353
__Rwread() — Write and Read a Record (separate buffers)..................................................354
samequantum32() - samequantum64() - samequantum128() — Determine if Quantum.
Exponents X and Y are the Same.........................................................................................356
scanf() — Read Data..............................................................................................................358
setbuf() — Control Buffering..............................................................................................364
setjmp() — Preserve Environment.......................................................................................365
setlocale() — Set Locale......................................................................................................366
setvbuf() — Control Buffering............................................................................................372
signal() — Handle Interrupt Signals...................................................................................374
sin() — Calculate Sine.........................................................................................................376
sinh() — Calculate Hyperbolic Sine....................................................................................377
sprintf() — Print Formatted Data to Buffer.......................................................................379
sqrt() — Calculate Square Root........................................................................................380
srand() — Set Seed for rand() Function.............................................................................381
sscanf() — Read Data...........................................................................................................382
strcat() — Concatenate Strings........................................................................................385
strcatp() — Concatenate Strings........................................................................................385
strch() — Search for Character..........................................................................................386
strcpai() — Compare Strings Without Case Sensitivity.....................................................388
strcoll() — Compare Strings..............................................................................................389
strcpi() — Compare Strings Without Case Sensitivity.......................................................391
strncpy() — Copy Strings....................................................................................................392
strcspn() — Find Offset of First Character Match..............................................................393
strdup() — Duplicate String...............................................................................................395
strerror() — Set Pointer to Runtime Error Message.........................................................396
strftime() — Convert Monetary Value to String..................................................................396
strftime() — Convert Date/Time to String..........................................................................399
stricmp() — Compare Strings without Case Sensitivity.....................................................403
strlen() — Determine String Length....................................................................................404
strncasecmp() — Compare Strings without Case Sensitivity.........................................405
Runtime Considerations ......................................................... 543
errno Macros ........................................................................... 543
errno Values for Integrated File System Enabled C Stream I/O .. 545
Record Input and Output Error Macro to Exception Mapping ... 547
Signal Handling Action Definitions ......................................... 548
Signal to Exception Mapping .................................................. 548
Cancel Handler Reason Codes ................................................ 551
Exception Classes .................................................................. 553
Data Type Compatibility ........................................................ 554
Runtime Character Set ............................................................ 564
Understanding CCSIDs and Locales ....................................... 565
CCSIDs of Characters and Character Strings ....................... 565
  Character Literal CCSID ...................................................... 566
  Job CCSID ........................................................................ 566
  File CCSID ........................................................................ 566
  Locale CCSID .................................................................... 567
Wide Characters ..................................................................... 568
  Wide Character Conversions to and from Single-Byte or Multibyte Characters ........................................ 568
    LOCALETYPE(*CLD) and LOCALETYPE(*LOCALE) behavior ......................................................... 568
    LOCALETYPE(*LOCALEUCS2) and LOCALETYPE(*LOCALEUTF) behavior .................................... 569
Wide Characters and File I/O .................................................. 569
  Wide character write functions ............................................ 569

Runtime Considerations ......................................................... 543
What's new

Here are the changes to this information for this edition.

• C99 Features
  – <ctype.h> (See “<ctype.h>” on page 1)
  – <math.h> (See “<math.h>” on page 6)
  – <stdarg.h> (See “<stdarg.h>” on page 11)
  – New <stdbool.h> (See “<stdbool.h>” on page 12)
  – <wchar.h> (See “<wchar.h>” on page 16)
  – <wctype.h> (See “<wctype.h>” on page 17)
  – “Mathematical” table (See “Mathematical” on page 20)
  – “Type Conversion” table (See “Type Conversion” on page 28)
  – “Handling Argument Lists” table (See “Handling Argument Lists” on page 44)
  – “Character Testing” table (See “Character Testing” on page 51)
  – “Multibyte Character Testing” table (See “Multibyte Character Testing” on page 53)
  – assert() (See “assert() — Verify Condition” on page 68)
  – erf(), erfc() (See “erf() — erfc() — Calculate Error Functions” on page 112)
  – hypot() (See “hypot() — Calculate Hypotenuse” on page 191)
  – isblank() (See “isalnum() — isxdigit() — Test Integer Value” on page 192)
  – New iswblank() (See “iswalnum() — iswxdigit() — Test Wide Integer Value” on page 194)
  – iswctype() (See “iswctype() — Test for Character Property” on page 196)
  – printf() (See “printf() — Print Formatted Characters” on page 251)
  – New quantexpd32(), quantexpd64(), quantexpd128() (See “quantexpd32() - quantexpd64() - quantexpd128() — Compute the Quantum Exponent” on page 270)
  – New quantized32(), quantized64(), quantized128() (See “quantized32() - quantized64() - quantized128() — Set the Quantum Exponent of X to the Quantum Exponent of Y” on page 271)
  – New samequantumd32(), samequantumd64(), samequantumd128() (See “samequantumd32() - samequantumd64() - samequantumd128() — Determine if Quantum Exponents X and Y are the Same” on page 356)
  – scanf() (See “scanf() — Read Data” on page 358)
  – strftime() (See “strftime() — Convert Date/Time to String” on page 399)
  – strtod() (See “strtod() - strtof() - strtold() — Convert Character String to Double, Float, and Long Double” on page 422)
  – strtod64() (See “strtod32() - strtod64() - strtod128() — Convert Character String to Decimal Floating-Point” on page 425)
  – New va_copy() (See “va_arg() — va_copy() — va_end() — va_start() — Handle Variable Argument List” on page 455)
  – wcstod() (See “wcstod() - wcstof() - wcstold() — Convert Wide-Character String to Double, Float, and Long Double” on page 510)
  – New wcstof(), wcstold() (See “wcstod() - wcstof() - wcstold() — Convert Wide-Character String to Double, Float, and Long Double” on page 510)
  – wcstod64() (See “wcstod32() - wcstod64() - wcstod128() — Convert Wide-Character String to Decimal Floating-Point” on page 512)
  – “errno Macros” table (See “errno Macros” on page 543)
- “Unicode from Other ILE Languages” section (See “Unicode from Other ILE Languages” on page 573)
- “Standard C Library Functions Table” (See “Standard C Library Functions Table, By Name” on page 591)

• Miscellaneous
  - exit() (See “exit() — End Program” on page 113)
  - _Ropen() (See “_Ropen() — Open a Record File for I/O Operations” on page 315)
  - New “Other ILE Languages” section (See “Other ILE Languages” on page 570)
  - “Data Type Compatibility” (See “Data Type Compatibility” on page 554)
Include Files

The include files that are provided with the runtime library contain macro and constant definitions, type definitions, and function declarations. Some functions require definitions and declarations from include files to work properly. The inclusion of files is optional, as long as the necessary statements from the files are coded directly into the source.

This section describes each include file, explains its contents, and lists the functions that are declared in the file.

The QSYSINC (system openness includes) library must be installed on your operating system. QSYSINC contains include files useful for C/C++ users, such as system API, Dynamic Screen Manager (DSM), and ILE header files. The QSYSINC library contains header files that include the prototypes and templates for the machine interface (MI) built-ins and the ILE C/C++ MI functions. See the ILE C/C++ for AS/400 MI Library Reference for more information about these header files.

<assert.h>

The <assert.h> include file defines the assert macro. You must include assert.h when you use assert.

The definition of assert is in an ifndef preprocessor block. If you have not defined the identifier NDEBUG through a define directive or on the compilation command, the assert macro tests the assertion expression. If the assertion is false, the system prints a message to stderr, and raises an abort signal for the program. The system also does a Dump Job (DMPJOB) OUTPUT(*PRINT) when the assertion is false.

If NDEBUG is defined, assert is defined to do nothing. You can suppress program assertions by defining NDEBUG.

<ctype.h>

The <ctype.h> include file defines functions that are used in character classification. The functions that are defined in <ctype.h> are:

<table>
<thead>
<tr>
<th>Function</th>
<th>Function</th>
<th>Function</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>isascii</td>
<td>iscntrl</td>
<td>isprint</td>
<td>isxdigit</td>
</tr>
<tr>
<td>isalnum</td>
<td>isdigit</td>
<td>ispunct</td>
<td>toascii</td>
</tr>
<tr>
<td>isalpha</td>
<td>isgraph</td>
<td>isspace</td>
<td>tolower</td>
</tr>
<tr>
<td>isblank</td>
<td>islower</td>
<td>isupper</td>
<td></td>
</tr>
</tbody>
</table>

Note: ¹ These functions are not available when LOCALETYPEx*CLD) is specified on the compilation command.

<decimal.h>

The <decimal.h> include file contains definitions of constants that specify the ranges of the packed decimal type and its attributes. The <decimal.h> file must be included with a #include directive in your source code if you use the keywords decimal, digitsof, or precisionof.
The <errno.h> include file defines macros that are set to the errno variable. The <errno.h> include file defines macros for values that are used for error reporting in the C library functions and defines the macro errno. An integer value can be assigned to errno, and its value can be tested during runtime. See "Checking the Errno Value" in the ILE C/C++ Programmer's Guide for information about displaying the current errno value.

Note: To test the value of errno after library function calls, set it to 0 before the call because its value may not be reset during the call.

The <except.h> include file declares types and macros that are used in ILE C exception handling.

The definition of _INTRPT_Hndlr_Parms_T is:

```c
typedef _Packed struct {
    unsigned int    Block_Size;
    _INVFLAGS_T     Tgt_Flags;
    char            reserved[8];
    _INVPTR         Target;
    _INVPTR         Source;
    _INVPTR         Com_Area;
    char            Compare_Data[32];
    char            Msg_Id[7];
    char            reserved1;
    _INTRPT_Mask_T  Mask;
    unsigned int    Msg_Ref_Key;
    unsigned short  Exception_Id;
    unsigned short  Compare_Data_Len;
    char            Signal_Class;
    char            Priority;
    short           Severity;
    char            reserved3[4];
    int             Msg_Data_Len;
    char            Mch_Dep_Data[10];
    char            Tgt_Inv_Type;
    _SUSPENDPTR     Tgt_Suspend;
    char            Ex_Data[48];
} _INTRPT_Hndlr_Parms_T;
```

Element Description

Block_Size
The size of the parameter block passed to the exception handler.

Tgt_Flags
Contains flags that are used by the system.

reserved
An eight byte reserved field.

Target
An invocation pointer to the call stack entry that enabled the exception handler.

Source
An invocation pointer to the call stack entry that caused the exception. If that call stack entry no longer exists, then this is a pointer to the call stack entry where control resumes when the exception is handled.

Com_Area
A pointer to the communications area variable specified as the second parameter on the #pragma exception_handler. If a communication area was not specified, this value is NULL.
**Compare_Data**

The compare data consists of 4 bytes of message prefix, for example CPF, MCH, followed by 28 bytes which are taken from the message data of the related message. In the case where the message data is greater than 28 these are the first 28 bytes. For MCH messages, these are the first 28 bytes of the exception related data that is returned by the system (substitution text).

**Msg_Id**

A message identifier, for example CPF123D. *STATUS message types are not updated in this field.

**reserved1**

A 1 byte pad.

**Mask**

This is an 8-byte exception mask, identifying the type of the exception that occurred, for example a decimal data error. The possible types are shown in Table 30 on page 553.

**Msg_Ref_Key**

A key used to uniquely identify the message.

**Exception_Id**

Binary value of the exception id, for example, 0x123D. To display value, use conversion specifier %x as information is stored in hex value.

**Compare_Data_Len**

The length of the compare data.

**Signal_Class**

Internal signal class.

**Priority**

The handler priority.

**Severity**

The message severity.

**reserved3**

A 4-byte reserved field.

**Msg_Data_Len**

The length of available message data.

**Mch_Dep_Data**

Machine-dependent data.

**Tgt_Inv_Type**

Invocation type. Macros are defined in <mimchobs.h>.

**Tgt_Suspend**

Suspend pointer of the target.

**Ex_Data**

The first 48 bytes of exception data.

The definition of _CNL_Hndlr_Parms_T is:

```c
typedef _Packed struct {
    unsigned int   Block_Size;
    _INVFLAGS_T    Inv_Flags;
    char           reserved[8];
    _INVPTTR       Invocation;
    _SPCPTR        Com_Area;
    _CNL_Mask_T    Mask;
} _CNL_Hndlr_Parms_T;
```

**Element**

**Description**

**Block_Size**

The size of the parameter block passed to the cancel handler.

**Inv_Flags**

Contains flags that are used by the system.
An eight byte reserved field.

Invocation

An invocation pointer to the invocation that is being cancelled.

Com_Area

A pointer to the handler communications area defined by the cancel handler.

Mask

A 4 byte value indicating the cancel reason.

The following built-ins are defined in <except.h>:

Built-in Description

__EXBDY

The purpose of the __EXBDY built-in or _EXBDY macro is to act as a boundary for exception-sensitive operations. An exception-sensitive operation is one that may signal an exception. An EXBDY enables programmers to selectively suppress optimizations that do code motion. For example, a divide is an exception-sensitive operation because it can signal a divide-by-zero. An execution path containing both an EXBDY and a divide will perform the two in the same order with or without optimization. For example:

```c
b = exp1;
c = exp2;
...
_EXBDY();
a = b/c;
```

__VBDY

The purpose of a __VBDY built-in or _VBDY macro is to ensure the home storage locations are current for variables that are potentially used on exception paths. This ensures the visibility of the current values of variables in exception handlers. A VBDY enables programmers to selectively suppress optimizations, such as redundant store elimination and forward store motion to enforce sequential consistency of variable updates. In the following example, the VBDYs ensure that state is in it's home storage location before each block of code that may signal an exception. A VBDY is often used in combination with an EXBDY to ensure that earlier assignments to state variables really update home storage locations and that later exception sensitive operations are not moved before these assignments.

```c
state = 1;
_VBDY();
/* Do stuff that may signal an exception. */
state = 2;
_VBDY();
/* More stuff that may signal an exception. */
state = 3;
_VBDY();
```

For more information about built-ins, see the ILE C/C++ for AS/400 MI Library Reference.

<float.h>

The <float.h> include file defines constants that specify the ranges of binary floating-point data types. For example, the maximum number of digits for objects of type double or the minimum exponent for objects of type float. In addition, if the macro variable __STDC_WANT_DEC_FP__ is defined, the include file also defines constants that specify ranges of decimal floating-point data types. For example, the maximum number of digits for objects of type _Decimal16 or the minimum exponent for objects of type _Decimal32.
The `<inttypes.h>` include file includes `<stdint.h>` and extends it with additional facilities.

The following macros are defined for format specifiers. These macros are defined for C programs. They are defined for C++ only when `__STDC_FORMAT_MACROS` is defined before `<inttypes.h>` is included.

```c
PRId8   PRIo8   PRIx8   SCnd16   SCnuLEAST16
PRId16  PRIo16  PRIx16  SCnd32   SCnuLEAST32
PRId32  PRIo32  PRIx32  SCnd64   SCnuLEAST64
PRId64  PRIo64  PRIx64  SCndFAST16  SCnuMAX
PRIdFAST8 PRIoFAST8 PRIxFAST8  SCndFAST32  SCnx16
PRIdFAST16 PRIoFAST16 PRIxFAST16  SCndFAST64  SCnx32
PRIdFAST32 PRIoFAST32 PRIxFAST32  SCndFAST64  SCnx64
PRIdFAST64 PRIoFAST64 PRIxFAST64  SCndFAST32  SCnxFAST16
PRIdLEAST8 PRIoLEAST8 PRIxLEAST8 SCndLEAST32  SCnxLEAST64
PRIdLEAST16 PRIoLEAST16 PRIxLEAST16 SCndLEAST64  SCnxLEAST32
PRIdLEAST32 PRIoLEAST32 PRIxLEAST32 SCndMAX  SCnxLEAST64
PRIdLEAST64 PRIoLEAST64 PRIxLEAST64 SCndMAX  SCnxLEAST64
PRIdMAX   PRIoMAX PRIxMAX SCNo16  SCnxLEAST64
PRId16   PRIo16   PRIx16 SCNo32  SCnxLEAST64
PRId32   PRIo32   PRIx32 SCNoFAST16 SCnxMAX
PRId64   PRIo64   PRIx64 SCNoFAST32 SCnxMAX
PRIdFAST8 PRIoFAST8 PRIxFAST8 SCNoFAST32 SCnxLEAST32
PRIdFAST16 PRIoFAST16 PRIxFAST16 SCNoFAST64 SCnxLEAST64
PRIdFAST32 PRIoFAST32 PRIxFAST32 SCNoFAST64 SCnxMAX
PRIdFAST64 PRIoFAST64 PRIxFAST64 SCNo16 SCnxFAST16
PRIdLEAST8 PRIoLEAST8 PRIxLEAST8 SCNo32 SCnxFAST16
PRIdLEAST16 PRIoLEAST16 PRIxLEAST16 SCNo64 SCnxFAST32
PRIdLEAST32 PRIoLEAST32 PRIxLEAST32 SCNo32 SCnxFAST32
PRIdLEAST64 PRIoLEAST64 PRIxLEAST64 SCNo32 SCnxFAST32
PRIdMAX   PRIoMAX PRIxMAX SCNo64 SCnxFAST64
```

The `<langinfo.h>` include file contains the declarations and definitions that are used by `nl_langinfo`.

The `<limits.h>` include file defines constants that specify the ranges of integer and character data types. For example, the maximum value for an object of type `char`.

The `<locale.h>` include file declares the `setlocale()`, `localeconv()`, and `wcslocaleconv()` library functions. These functions are useful for changing the C locale when you are creating applications for international markets.

The `<locale.h>` include file also declares the type `struct lconv` and the following macro definitions:

```c
NULL           LC_ALL          LC_C          LC_C_FRANCE
LC_C_GERMANY   LC_C_ITALY      LC_C_SPAIN     LC_C_UK
```
<math.h>

The `<math.h>` include file declares all the floating-point math functions:

<table>
<thead>
<tr>
<th>Function</th>
<th>Function</th>
<th>Function</th>
<th>Function</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>acos</td>
<td>cosh</td>
<td>frexp</td>
<td>nextafter</td>
<td>quantexpd128</td>
</tr>
<tr>
<td>asin</td>
<td>erf</td>
<td>gamma</td>
<td>nextafterl</td>
<td>quantized32</td>
</tr>
<tr>
<td>atan</td>
<td>erfc</td>
<td>hypot</td>
<td>nexttoward</td>
<td>quantized64</td>
</tr>
<tr>
<td>atan2</td>
<td>exp</td>
<td>ldexp</td>
<td>nexttowardl</td>
<td>quantized128</td>
</tr>
<tr>
<td>Bessel</td>
<td>fabs</td>
<td>log</td>
<td>pow</td>
<td>samequantum</td>
</tr>
<tr>
<td>ceil</td>
<td>floor</td>
<td>log10</td>
<td>quantexpd32</td>
<td>samequantumd</td>
</tr>
<tr>
<td>cos</td>
<td>fmod</td>
<td>modf</td>
<td>quantexpd64</td>
<td>samequantumd</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>32</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>64</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>128</td>
</tr>
</tbody>
</table>

Note:

1. The Bessel functions are a group of functions named j0, j1, jn, y0, y1, and yn.
2. Floating-point numbers are only guaranteed 15 significant digits. This can greatly affect expected results if multiple floating-point numbers are used in a calculation.

`<math.h>` defines the macro HUGE_VAL, which expands to a positive double expression, and possibly to infinity on systems that support infinity.

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

A range error occurs if the result of the function cannot be represented in a **double** value. If the magnitude of the result is too large (overflow), the function returns the positive or negative value of the macro HUGE_VAL, and sets errno to ERANGE. If the result is too small (underflow), the function returns zero.

<mallocinfo.h>

Include file with _C_TS_malloc_info and _C_TS_malloc_debug.

<monetary.h>

The `<monetary.h>` header file contains declarations and definitions that are related to the output of monetary quantities. The following monetary functions are defined: strfmon() and wcsfmon(). The strfmon() function is not available when LOCALETYPE(*CLD) is specified on the compilation command. The wcsfmon() function is available only when LOCALETYPE(*LOCALEUTF) is specified on the compilation command.
<nl_types.h>

The <nl_types.h> header file contains catalog definitions and the following catalog functions: `catclose()`, `catgets()`, and `catopen()`. These definitions are not available when either `LOCALETYPE(*CLD)` or `SYSIFCOPT(*NOIFSIO)` is specified on the compilation command.

<pointer.h>

The <pointer.h> include file contains typedefs and pragma directives for the operating system pointer types: space pointer, open pointer, invocation pointer, label pointer, system pointer, and suspend pointer. The typedefs `_ANYPTR` and `_SPCPTRCN` are also defined in <pointer.h>.

<recio.h>

The <recio.h> include file defines the types and macros, and prototypes functions for all the ILE C record input and output (I/O) operations.

The following functions are defined in <recio.h>:

- `_Racquire`
- `_Rclose`
- `_Rcommit`
- `_Rdelete`
- `_Rdevatr`
- `_Rfeed`
- `_Rfeov`
- `_Rformat`
- `_Rindara`
- `_Riofbk`
- `_Rlocate`
- `_Ropen`
- `_Ropnfbk`
- `_Rpgmdev`
- `_Rread`
- `_Rreadf`
- `_Rreadindv`
- `_Rreadk`
- `_Rreadl`
- `_Rreadn`
- `_Rreadnc`
- `_Rrlslck`
- `_Rrollbck`
- `_Rupdate`
- `_Rupfb`
- `_Rwrite`
- `_Rwrited`
- `_Rwriterd`
- `_Rwrread`

The following positioning macros are defined in <recio.h>:

- `__END`
- `__END_FRC`
- `__FIRST`
- `__KEY_EQ`
- `__KEY_GE`
- `__KEY_GT`
- `__KEY_LE`
- `__KEY_LT`
- `__KEY_NEXTEQ`
- `__KEY_NEXTUNQ`
- `__KEY_PREVEQ`
- `__KEY_PREVUNQ`
- `__KEY_LAST`
- `__KEY_NEXT`
- `__NO_POSITION`
- `__PREVIOUS`
- `__PRIOR`
- `__RRN_EQ`
- `__START`
- `__START_FRC`
- `__LAST`
- `__NEXT`
Any of the record I/O functions that include a buffer parameter may work in move mode or locate mode. In move mode, data is moved between the user-supplied buffer and the system buffer. In locate mode, the user must access the data in the system buffer. Pointers to the system buffers are exposed in the _RFILE structure. To specify that locate mode is being used, the buffer parameter of the record I/O function is coded as NULL.

A number of the functions include a size parameter. For move mode, this is the number of data bytes that are copied between the user-supplied buffer and the system buffer. All of the record I/O functions work with one record at a time regardless of the size that is specified. The size of this record is defined by the file description. It may not be equal to the size parameter that is specified by the user on the call to the record I/O functions. The amount of data that is moved between buffers is equal to the record length of the current record format or specified minimum size, whichever is smaller. The size parameter is ignored for locate mode.

The following types are defined in recio.h:

### Information for controlling opened record I/O operations

typedef _Packed struct {
    char reserved1[16];
    volatile void  *const *const in_buf;
    volatile void  *const *const out_buf;
    char _RIOFB_T riofb;
    char reserved2[48];
    const unsigned int buf_length;
    char reserved3[32];
    volatile char  *const in_null_map;
    volatile char  *const out_null_map;
    volatile char  *const null_key_map;
    char reserved4[28];
    const int min_length;
    short null_map_len;
    short null_key_map_len;
    char reserved5[48];
    int dup_key :1;
    int icf_locate :1;
    int reserved1 :6;
} _RFILE;

**Element**

**Description**

**in_null_map**
- Specifies which fields are to be considered NULL when you read from a database file.

**out_null_map**
- Specifies which fields are to be considered NULL when you write to a database file.

**null_key_map**
- Specifies which fields contain NULL if you are reading a database by key.

**null_map_len**
- Specifies the lengths of the in_null_map and out_null_map.

**null_key_map_len**
- Specifies the length of the null_key_map.

### Record I/O Feedback Information

typedef struct {
    unsigned char  *key;
    _Sys_Struct_T  *sysparm;
    unsigned long  rrn;
    long num_bytes;
    short blk_count;
    char blk_filled_by;
    int dup_key :1;
    int icf_locate :1;
    int reserved1 :6;
} _RFILE;
### Element Description

**key**
If you are processing a file using a keyed sequence access path, this field contains a pointer to the key value of the record successfully positioned to, read or written.

**sysparm**
This field is a pointer to the major and minor return code for ICF, display, and printer files.

**rrn**
This field contains the relative record number of the record that was successfully positioned to, read or written.

**num_bytes**
This field contains the number of bytes that are read or are written.

**blk_count**
This field contains the number of records that remain in the block. If the file is open for input, blkrcd=y is specified, and a read function is called, this field will be updated with the number of records remaining in the block.

**blk_filled_by**
This field indicates the operation that filled the block. If the file is open for input, blkrcd=y is specified, and a read function is called. This field will be set to the __READ_NEXT macro if the _Rreadn function filled the block or to the __READ_PREV macro if the _Rreadp function filled the block.

### System-Specific Information

```c
typedef struct {
    void              *sysparm_ext;
    _Maj_Min_rc_T    _Maj_Min;
    char              reserved1[12];
} _Sys_Struct_T;
```

### Major and Minor Return Codes

```c
typedef struct { char    major_rc[2]; char    minor_rc[2]; } _Maj_Min_rc_T;
```

The following macros are defined in recio.h:

**_FILENAME_MAX**
Expands to an integral constant expression that is the size of a character array large enough to hold the longest file name. This is the same as the stream I/O macro.

**_ROPEN_MAX**
Expands to an integral constant expression that is the maximum number of files that can be opened simultaneously.

The following null field macros are defined in recio.h:

### Element Description

**_CLEAR_NULL_MAP(file, type)**
Clears the null output field map that indicates that there are no null fields in the record to be written to file. type is a typedef that corresponds to the null field map for the current record format.

**_CLEAR_UPDATE_NULL_MAP(file, type)**
Clears the null input field map that indicates that no null fields are in the record to be written to file. type is a typedef that corresponds to the null field map for the current record format.
_QRY_NULL_MAP(file, type)
    Returns the number of fields that are null in the previously read record. type is a typedef that corresponds to the null field map for the current record format.

_CLEAR_NULL_KEY_MAP(file, type)
    Clears the null key field map so that it indicates no null key fields in the record to be written to file. type is a typedef that corresponds to the null key field map for the current record format.

_SET_NULL_MAP_FIELD(file, type, field)
    Sets the specified field in the output null field map so that field is considered NULL when the record is written to file.

_SET_UPDATE_NULL_MAP_FIELD(file, type, field)
    Sets the specified field in the input null field map so that field is considered null when the record is written to file. type is a typedef that corresponds to the null key field map for the record format.

_QRY_NULL_MAP_FIELD(file, type, field)
    Returns 1 if the specified field in the null input field map indicates that the field is to be considered null in the previously read record. If field is not null, it returns zero. type is a typedef that corresponds to the NULL key field map for the current record format.

_SET_NULL_KEY_MAP_FIELD(file, type, field)
    Sets the specified field map that indicates that the field will be considered null when the record is read from file. type is a typedef that corresponds to the null key field map for the current record format.

_QRY_NULL_KEY_MAP(file, type)
    Returns the number of fields that are null in the key of the previously read record. type is a typedef that corresponds to the null field map for the current record format.

_QRY_NULL_KEY_MAP_FIELD(file, type, field)
    Returns 1 if the specified field in the null key field map indicates that field is to be considered null in the previously read record. If field is not null, it returns zero. type is a typedef that corresponds to the null key field map for the current record format.

<regex.h>

The <regex.h> include file defines the following regular expression functions:

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

The <regex.h> include file also declares the regmatch_t type, the regex_t type, which is capable of storing a compiled regular expression, and the following macros:

Values of the cflags parameter of the regcomp() function:

    REG_BASIC
    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_NOMATCH
    REG_BADPAT
    REG_ECOLLATE
    REG_ECTYPE
    REG_EESCAPE
These declarations and definitions are not available when LOCALETYPE(*CLD) is specified on the compilation command.

**Note:** The regular expressions supported by regcomp() and regexec() follow the specification described here: "http://www.opengroup.org/onlinepubs/007908799/xbd/re.html".

**<setjmp.h>**

The `<setjmp.h>` include file declares the `setjmp()` function and `longjmp()` function. It also defines a buffer type, `jmp_buf`, that the `setjmp()` and `longjmp()` functions use to save and restore the program state.

**<signal.h>**

The `<signal.h>` include file defines the values for signals and declares the `signal()` and `raise()` functions.

The `<signal.h>` include file also defines the following macros:

```
SIGABRT  SIG_ERR  SIGILL  SIGOTHER  SIGUSR1
SIGALL   SIGFPE   SIGINT  SIGSEGV   SIGUSR2
SIG_DFL  SIG_IGN  SIGIO   SIGTERM
```

`<signal.h>` also declares the function `_GetExcData`, an IBM i extension to the C standard library.

**<stdarg.h>**

The `<stdarg.h>` include file defines macros that allow you access to arguments in functions with variable-length argument lists: `va_arg()`, `va_copy()`, `va_start()`, and `va_end()`. The `<stdarg.h>` include file also defines the type `va_list`.

**<stddef.h>**

The `<stddef.h>` include file declares the commonly used pointers, variables, and types as listed below:

```
ptrdiff_t
typedef for the type of the difference of two pointers
size_t
typedef for the type of the value that is returned by sizeof
```
typedef for a wide character constant.

The `<stddef.h>` include file also defines the macros NULL and offsetof. NULL is a pointer that is guaranteed not to point 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)
```

The `<stddef.h>` include file also declares the `extern` variable _EXCP_MSGID, an IBM i extension to C.

```
<stdbool.h>
```

The `<stdbool.h>` include file defines macros to make it easier to use the _Bool data type.

```
<stdint.h>
```

The `<stdint.h>` include file declares sets of integer types that have specified widths and defines corresponding sets of macros. It also defines macros that specify limits of integer types corresponding to the types defined in other standard include files.

The following exact-width integer types are defined:

- `int8_t`
- `int16_t`
- `int32_t`
- `int64_t`
- `uint8_t`
- `uint16_t`
- `uint32_t`
- `uint64_t`

The following minimum-width integer types are defined:

- `int_least8_t`
- `int_least16_t`
- `int_least32_t`
- `int_least64_t`
- `uint_least8_t`
- `uint_least16_t`
- `uint_least32_t`
- `uint_least64_t`

The following fastest minimum-width integer types are defined:

- `int_fast8_t`
- `int_fast16_t`
- `int_fast32_t`
- `int_fast64_t`
- `uint_fast8_t`
- `uint_fast16_t`
- `uint_fast32_t`
- `uint_fast64_t`

The following greatest-width integer types are defined:

- `intmax_t`
- `uintmax_t`

The following macros are defined for limits of exact-width integer types (See note “1” on page 13):

- `INT8_MAX`
- `INT8_MIN`
- `INT16_MAX`
- `INT16_MIN`
- `INT32_MAX`
- `INT32_MIN`
- `INT64_MAX`
- `INT64_MIN`
- `UINT16_MAX`
- `UINT16_MIN`
- `UINT32_MAX`
- `UINT32_MIN`
- `UINT64_MAX`
- `UINT64_MIN`
The following macros are defined for limits of fastest minimum-width integer types (See note “1” on page 13):

\[
\begin{align*}
\text{INT}_8\_\text{MAX} & & \text{INT}_8\_\text{MIN} & & \text{INT}_16\_\text{MAX} & & \text{INT}_16\_\text{MIN} & & \text{INT}_32\_\text{MAX} & & \text{INT}_32\_\text{MIN} & & \text{INT}_64\_\text{MAX} & & \text{INT}_64\_\text{MIN} & & \text{UINT}_8\_\text{MAX} & & \text{UINT}_8\_\text{MIN} & & \text{UINT}_16\_\text{MAX} & & \text{UINT}_16\_\text{MIN} & & \text{UINT}_32\_\text{MAX} & & \text{UINT}_32\_\text{MIN} & & \text{UINT}_64\_\text{MAX} & & \text{UINT}_64\_\text{MIN} \\
\end{align*}
\]

The following macros are defined for limits of greatest-width integer types (See note “1” on page 13):

\[
\begin{align*}
\text{INTMAX}\_\text{MIN} & & \text{INTMAX}\_\text{MAX} & & \text{UINTMAX}\_\text{MAX} \\
\end{align*}
\]

The following macros are defined for limits for other integer types (See note “1” on page 13):

\[
\begin{align*}
\text{PTRDIFF}\_\text{MAX} & & \text{SIG}_\text{ATOMIC}\_\text{MIN} & & \text{WCHAR}\_\text{MIN} & & \text{SIG}_\text{ATOMIC}\_\text{MAX} & & \text{WCHAR}\_\text{MAX} & & \text{WINT}\_\text{MAX} & & \text{WINT}\_\text{MIN} \\
\end{align*}
\]

The following macros are defined for minimum-width integer constant expressions (See note “2” on page 13):

\[
\begin{align*}
\text{INT}_8\_\text{C} & & \text{INT}_32\_\text{C} & & \text{UINT}_8\_\text{C} & & \text{UINT}_32\_\text{C} & & \text{INT}_16\_\text{C} & & \text{INT}_64\_\text{C} & & \text{UINT}_16\_\text{C} & & \text{UINT}_64\_\text{C} \\
\end{align*}
\]

The following macros are defined for greatest-width integer constant expressions (See note “2” on page 13):

\[
\begin{align*}
\text{INTMAX}\_\text{C} & & \text{UINTMAX}\_\text{C} \\
\end{align*}
\]

**Note:**

1. These macros are defined for C programs. They are defined for C++ only when `__STDC_LIMIT_MACROS` is defined before `<stdint.h>` is included.
2. These macros are defined for C programs. They are defined for C++ only when `__STDC_CONSTANT_MACROS` is defined before `<stdint.h>` is included.

---

`<stdio.h>`

The `<stdio.h>` include file defines constants, macros, and types, and declares stream input and output functions. The stream I/O functions are:
<table>
<thead>
<tr>
<th>Function</th>
<th>C_Get_Ssn_HANDL</th>
<th>fprintf</th>
<th>fwrite</th>
<th>remove</th>
<th>vfscanf</th>
</tr>
</thead>
<tbody>
<tr>
<td>e clearerr</td>
<td>_C_Get_Ssn_HANDL</td>
<td>fputc</td>
<td>fwscanf</td>
<td>rename</td>
<td>vfwscanf</td>
</tr>
<tr>
<td>fclose</td>
<td>fputchar</td>
<td>getc</td>
<td>re绕</td>
<td>vprintf</td>
<td></td>
</tr>
<tr>
<td>fdopen2</td>
<td>fputc</td>
<td>getchar</td>
<td>scanf</td>
<td>vprintf</td>
<td></td>
</tr>
<tr>
<td>feof</td>
<td>fputwc1</td>
<td>gets</td>
<td>setbuf</td>
<td>vscanf</td>
<td></td>
</tr>
<tr>
<td>ferror</td>
<td>fputws1</td>
<td>getwc1</td>
<td>setvbuf</td>
<td>vsscanf</td>
<td></td>
</tr>
<tr>
<td>fflush</td>
<td>freopen</td>
<td>perror</td>
<td>sprintf</td>
<td>vsscanf</td>
<td></td>
</tr>
<tr>
<td>fgetc</td>
<td>fscanf</td>
<td>printf</td>
<td>vsprintf</td>
<td></td>
<td></td>
</tr>
<tr>
<td>fgetpos</td>
<td>fseek</td>
<td>putc</td>
<td>vprintf</td>
<td></td>
<td></td>
</tr>
<tr>
<td>fgetws1</td>
<td>fsetpos</td>
<td>putchar</td>
<td>tmpnam</td>
<td></td>
<td></td>
</tr>
<tr>
<td>fopen</td>
<td>ftell</td>
<td>puts</td>
<td>ungetc</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>fwide1</td>
<td>putwc1</td>
<td>ungetwc1</td>
<td>wscanf1</td>
<td></td>
</tr>
<tr>
<td>_C_Get_Ssn_HANDL</td>
<td>/fwprintf1</td>
<td>putwchar1</td>
<td>vscanf</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Note:** 1 These functions are not available when either LOCALETYPE(*CLD) or SYSIFCOPT(*NOIFSIO) is specified on the compilation command.

**Note:** 2 These functions are available when SYSIFCOPT(*IFSIO) is specified on the compilation command.

The `<stdio.h>` include file also defines the macros that are listed below. You can use these constants in your programs, but you should not alter their values.

**BUFSIZ**
Specifies the buffer size that the setbuf library function will use when you are allocating buffers for stream I/O. This value establishes the size of system-allocated buffers and is used with setbuf.

**EOF**
The value that is returned by an I/O function when the end of the file (or in some cases, an error) is found.

**FOPEN_MAX**
The number of files that can be opened simultaneously.

**FILENAME_MAX**
The longest file name that is supported. If there is no reasonable limit, FILENAME_MAX will be the recommended size.

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

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

**NULL**
A pointer guaranteed not to point to a data object.

The FILE structure type is defined in `<stdio.h>`. Stream I/O 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.

When integrated file system is enabled with a compilation parameter SYSIFCOPT(*IFSIO), ifs.h is included into `<stdio.h>`.

The C standard streams stdin, stdout, and stderr are also defined in `<stdio.h>`.

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 type fpos_t is defined in <stdio.h> for use with fgetpos() and fsetpos().
See “<stddef.h>” on page 11 for more information about NULL.

**<stdlib.h>**

The <stdlib.h> include file declares the following functions:

<table>
<thead>
<tr>
<th>Function</th>
<th>Function</th>
<th>Function</th>
<th>Function</th>
<th>Function</th>
<th>Function</th>
<th>Function</th>
<th>Function</th>
<th>Function</th>
<th>Function</th>
<th>Function</th>
<th>Function</th>
<th>Function</th>
<th>Function</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>abort</td>
<td>_C_Quickpool_R</td>
<td>ldiv</td>
<td>realloc</td>
<td>strtoul</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>abs</td>
<td>report</td>
<td>lldiv</td>
<td>srand</td>
<td>strto11</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>atexit</td>
<td>div</td>
<td>malloc</td>
<td>strtod</td>
<td>system</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>atof</td>
<td>exit</td>
<td>mb1en</td>
<td>strtod32</td>
<td>_ultoa1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>atoi</td>
<td>free</td>
<td>mbstowcs</td>
<td>strtd64</td>
<td>wctombs</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>atol</td>
<td>gcvt1</td>
<td>mbotwc</td>
<td>strtd128</td>
<td>wctomb</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>bsearch</td>
<td>getenv</td>
<td>putenv</td>
<td>strtof</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>calloc</td>
<td>_itoa1</td>
<td>qsort</td>
<td>strtol</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>_C_Quickpool_D</td>
<td>_ltoa1</td>
<td>rand</td>
<td>strtd10</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ebbug</td>
<td>labs</td>
<td>rand_r</td>
<td>strterr</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>_C_Quickpool_I</td>
<td>llabs</td>
<td>strncmpl</td>
<td>strto1ll</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>init</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Note:** These functions are applicable to C++ only.

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

**NULL**
The NULL pointer value.

**EXIT_SUCCESS**
Expands to 0; used by the atexit function.

**EXIT_FAILURE**
Expands to 8; used by the atexit function.

**RAND_MAX**
Expands to an integer that represents the largest number that the rand function can return.

**MB_CUR_MAX**
Expands to an integral expression to represent the maximum number of bytes in a multibyte character for the current locale.

For more information about NULL and the types size_t and wchar_t, see “<stddef.h>” on page 11.

**<string.h>**

The <string.h> include file declares the string manipulation functions:

<table>
<thead>
<tr>
<th>Function</th>
<th>Function</th>
<th>Function</th>
<th>Function</th>
<th>Function</th>
<th>Function</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>memchr</td>
<td>strcat</td>
<td>strcspn</td>
<td>strncmp</td>
<td>strset1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>memcmp</td>
<td>strchr</td>
<td>strndup1</td>
<td>strncpy</td>
<td>strspn</td>
<td></td>
<td></td>
</tr>
<tr>
<td>memicmp</td>
<td>strcmp</td>
<td>strerror</td>
<td>strnicmp1</td>
<td>strsr1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>memmove</td>
<td>strcmpl</td>
<td>strlen</td>
<td>strpbrk</td>
<td>strtok</td>
<td></td>
<td></td>
</tr>
<tr>
<td>memset</td>
<td>strcpy</td>
<td>strncat</td>
<td>strrrchr</td>
<td>strtok1_r</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>strxfrm</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Include Files 15
Note: These functions are available for C++ programs. They are available for C only when the program defines the __cplusplus__ strings macro.

The <string.h> include file also defines the macro NULL, and the type size_t.
For more information about NULL and the type size_t, see “<stddef.h>” on page 11.

<strings.h>
Contains the functions strcasecmp and strncasecmp.

<time.h>
The <time.h> include file declares the time and date functions:

<table>
<thead>
<tr>
<th>asctime</th>
<th>ct ime_r</th>
<th>gmtime64</th>
<th>localtime_r</th>
<th>strftime1</th>
</tr>
</thead>
<tbody>
<tr>
<td>asctime_r</td>
<td>ctime64_r</td>
<td>gmtime_r</td>
<td>localtime64_r</td>
<td>time</td>
</tr>
<tr>
<td>clock</td>
<td>difftime</td>
<td>gmtime64_r</td>
<td>mktime</td>
<td>time64</td>
</tr>
<tr>
<td>ctime</td>
<td>difftime64</td>
<td>localtime</td>
<td>mktime64</td>
<td>strftime</td>
</tr>
<tr>
<td>ctime64</td>
<td>gmtime</td>
<td>localtime64</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Note: These functions are not available when LOCALETYPE(*CLD) is specified on the compilation command.

The <time.h> include file also provides:
- A structure tm that contains the components of a calendar time. See “gmtime() — Convert Time” on page 183 for a list of the tm structure members.
- A macro CLOCKS_PER_SEC equal to the number per second of the value that is returned by the clock function.
- Types clock_t, time_t, time64_t, and size_t.
- The NULL pointer value.
For more information about NULL and the type size_t, see “<stddef.h>” on page 11.

<wchar.h>
The <wchar.h> header file contains declarations and definitions that are related to the manipulation of wide character strings. Any functions which deal with files are accessible if SYSIFCOPT(*IFSIO) is specified.
<wchar.h> also defines the macro NULL and the types size_t and wchar_t.

For more information about NULL and the types size_t and wchar_t, see "<stddef.h>" on page 11.

<wctype.h>

The <wctype.h> header file declares the following wide character functions:

iswalnum  iswdigit  iswpunct  iswctype  wcctype
iswalpha  iswgraph  iswspace  towlower  wcrtans
iswblank  iswlower  iswupper  towupper
iswcntrl  iswprint  iswxdigit  towctrans

The <wctype.h> header file also contains declarations and definitions for wide character classification. These declarations and definitions are not available when LOCALETYPE(*CLD) is specified on the compilation command.

<xxcvt.h>

The <xxcvt.h> include file contains the declarations that are used by the QXXDTOP, QXXDTOZ, QXXITOP, QXXITOZ, QXXPTOI, QXXPTOD, QXXZTOD, and QXXZTOI conversion functions.

<xxdtaa.h>

The <xxdtaa.h> include file contains the declarations for the data area interface functions QXXCHGDA, QXXRTVDA, and the type _DTAA_NAME_T.

The definition of _DTAA_NAME_T is:
typedef struct _DTAA_NAME_T {
    char dtaa_name[10];
    char dtaa_lib[10];
} _DTAA_NAME_T;

<xxenv.h>

The <xxenv.h> include file contains the declarations for the QPXXCALL and QPXXDLTE EPM environment handling program. ILE procedures cannot be called from this interface.

The definition of _ENVPGM_T is:

typedef struct _ENVPGM_T {
    char pgmname[10];
    char pgmlib[10];
} _ENVPGM_T;

<xxfdbk.h>

The <xxfdbk.h> include file contains the declarations that are used by the operating system feedback areas. To retrieve information from feedback areas, see “_Riofbk() — Obtain I/O Feedback Information” on page 310 and “_Ropnfbk() — Obtain Open Feedback Information” on page 319.

The following is an example of a type that is defined in the <xxfdbk.h> include file:

typedef _Packed struct _XXIOFB_T {
    short  file_dep_fb_offset;
    int    write_count;
    int    read_count;
    int    write_read_count;
    int    other_io_count;
    char   reserved1;
    char   cur_operation;
    char   rec_format[10];
    char   dev_class[2];
    char   dev_name[10];
    int    last_io_rec_len;
    char   reserved2[80];
    short  num_recs_retrieved;
    short  last_io_rec_len2;
    char   reserved3[2];
    int    cur_blk_count;
    char   reserved4[8];
} _XXIOFB_T;

For further information about the open feedback areas, see the Files and file systems category in the Information Center.

Machine Interface (MI) Include Files

See the ILE C/C++ for AS/400 MI Library Reference for a description of the MI header files.
Library Functions

This topic describes the standard C/C++ library functions and the ILE C/C++ extensions to the library functions, except for the ILE C/C++ MI functions. See the *ILE C/C++ for AS/400 MI Library Reference* for more information about the MI functions.

Each library function that is listed in this section contains:

- A format description that shows the include file that declares the function.
- The data type that is returned by the function.
- The required data types of the arguments to the function.

This example shows the format of the log() function:

```c
#include <math.h>
double log(double x);
```

The example shows that:

- you must include the file math.h in the program.
- the log() function returns type double.
- the log() function requires an argument x of type double.

Examples throughout the section illustrate the use of library functions and are not necessarily complete.

This topic lists the library functions in alphabetic order. If you are unsure of the function you want to use, see the summary of the library functions in “The C/C++ Library” on page 19.

**Note:** All functions are considered threadsafe unless noted otherwise.

The C/C++ Library

This topic summarizes the available C/C++ library functions and their location in this book. It also briefly describes what the function does. Each library function is listed according to the type of function it performs.

### Error Handling

<table>
<thead>
<tr>
<th>Function</th>
<th>Header File</th>
<th>Page</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>assert()</td>
<td>assert.h</td>
<td>“assert() — Verify Condition” on page 68</td>
<td>Prints diagnostic messages.</td>
</tr>
<tr>
<td>atexit()</td>
<td>stdlib.h</td>
<td>“atexit() — Record Program Ending Function” on page 70</td>
<td>Registers a function to be executed at program end.</td>
</tr>
<tr>
<td>clearerr()</td>
<td>stdio.h</td>
<td>“clearerr() — Reset Error Indicators” on page 87</td>
<td>Resets error indicators.</td>
</tr>
<tr>
<td>feof()</td>
<td>stdio.h</td>
<td>“feof() — Test End-of-File Indicator” on page 119</td>
<td>Tests end-of-file indicator for stream input.</td>
</tr>
<tr>
<td>ferror()</td>
<td>stdio.h</td>
<td>“ferror() — Test for Read/Write Errors” on page 120</td>
<td>Tests the error indicator for a specified stream.</td>
</tr>
<tr>
<td>Function</td>
<td>Header File</td>
<td>Page</td>
<td>Description</td>
</tr>
<tr>
<td>----------------</td>
<td>-------------</td>
<td>-------------------------------------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>_GetExcData()</td>
<td>signal.h</td>
<td>“_GetExcData() — Get Exception Data” on page 176</td>
<td>Retrieves information about an exception from within a C signal handler. This function is not defined when SYSIFCOPT(*SYNCSIGNAL) is specified on the compilation command.</td>
</tr>
<tr>
<td>perror()</td>
<td>stdio.h</td>
<td>“perror() — Print Error Message” on page 249</td>
<td>Prints an error message to stderr.</td>
</tr>
<tr>
<td>raise()</td>
<td>signal.h</td>
<td>“raise() — Send Signal” on page 283</td>
<td>Initiates a signal.</td>
</tr>
<tr>
<td>signal()</td>
<td>signal.h</td>
<td>“signal() — Handle Interrupt Signals” on page 374</td>
<td>Allows handling of an interrupt signal from the operating system.</td>
</tr>
<tr>
<td>strerror()</td>
<td>string.h</td>
<td>“strerror() — Set Pointer to Runtime Error Message” on page 396</td>
<td>Retrieves pointer to system error message.</td>
</tr>
</tbody>
</table>

### Searching and Sorting

<table>
<thead>
<tr>
<th>Function</th>
<th>Header File</th>
<th>Page</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>bsearch()</td>
<td>stdlib.h</td>
<td>“bsearch() — Search Arrays” on page 76</td>
<td>Performs a binary search of a sorted array.</td>
</tr>
<tr>
<td>qsort()</td>
<td>stdlib.h</td>
<td>“qsort() — Sort Array” on page 272</td>
<td>Performs a quick sort on an array of elements.</td>
</tr>
</tbody>
</table>

### Mathematical

<table>
<thead>
<tr>
<th>Function</th>
<th>Header File</th>
<th>Page</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>abs()</td>
<td>stdlib.h</td>
<td>“abs() — Calculate Integer Absolute Value” on page 61</td>
<td>Calculates the absolute value of an integer.</td>
</tr>
<tr>
<td>ceil()</td>
<td>math.h</td>
<td>“ceil() — Find Integer &gt;=Argument” on page 86</td>
<td>Calculates the double value representing the smallest integer that is greater than or equal to a number.</td>
</tr>
<tr>
<td>div()</td>
<td>stdlib.h</td>
<td>“div() — Calculate Quotient and Remainder” on page 111</td>
<td>Calculates the quotient and remainder of an integer.</td>
</tr>
<tr>
<td>erf()</td>
<td>math.h</td>
<td>“erf() – erfc() — Calculate Error Functions” on page 112</td>
<td>Calculates the error function.</td>
</tr>
<tr>
<td>Function</td>
<td>Header File</td>
<td>Page</td>
<td>Description</td>
</tr>
<tr>
<td>----------</td>
<td>-------------</td>
<td>------</td>
<td>-------------</td>
</tr>
<tr>
<td>erfc()</td>
<td>math.h</td>
<td>“erfc() — Calculate Error Functions” on page 112</td>
<td>Calculates the error function for large numbers.</td>
</tr>
<tr>
<td>exp()</td>
<td>math.h</td>
<td>“exp() — Calculate Exponential Function” on page 114</td>
<td>Calculates an exponential function.</td>
</tr>
<tr>
<td>fabs()</td>
<td>math.h</td>
<td>“fabs() — Calculate Floating-Point Absolute Value” on page 115</td>
<td>Calculates the absolute value of a floating-point number.</td>
</tr>
<tr>
<td>floor()</td>
<td>math.h</td>
<td>“floor() — Find Integer &lt;= Argument” on page 132</td>
<td>Calculates the double value representing the largest integer that is less than or equal to a number.</td>
</tr>
<tr>
<td>fmod()</td>
<td>math.h</td>
<td>“fmod() — Calculate Floating-Point Remainder” on page 133</td>
<td>Calculates the floating-point remainder of one argument divided by another.</td>
</tr>
<tr>
<td>frexp()</td>
<td>math.h</td>
<td>“frexp() — Separate Floating-Point Value” on page 154</td>
<td>Separates a floating-point number into its mantissa and exponent.</td>
</tr>
<tr>
<td>gamma()</td>
<td>math.h</td>
<td>“gamma() — Gamma Function” on page 172</td>
<td>Calculates the gamma function.</td>
</tr>
<tr>
<td>hypot()</td>
<td>math.h</td>
<td>“hypot() — Calculate Hypotenuse” on page 191</td>
<td>Calculates the hypotenuse.</td>
</tr>
<tr>
<td>labs()</td>
<td>stdlib.h</td>
<td>“labs() — Calculate Absolute Value of Long and Long Long Integer” on page 199</td>
<td>Calculates the absolute value of a long integer.</td>
</tr>
<tr>
<td>llabs()</td>
<td>stdlib.h</td>
<td>“llabs() — Calculate Absolute Value of Long and Long Long Integer” on page 199</td>
<td>Calculates the absolute value of a long long integer.</td>
</tr>
<tr>
<td>Function</td>
<td>Header File</td>
<td>Page</td>
<td>Description</td>
</tr>
<tr>
<td>---------------</td>
<td>-------------</td>
<td>----------------------------------------------------------------------</td>
<td>--------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>ldexp()</td>
<td>math.h</td>
<td>“ldexp() — Multiply by a Power of Two” on page 200</td>
<td>Multiplies a floating-point number by an integral power of 2.</td>
</tr>
<tr>
<td>ldiv()</td>
<td>stdlib.h</td>
<td>“ldiv() – lldiv() — Perform Long and Long Long Division” on page 201</td>
<td>Calculates the quotient and remainder of a long integer.</td>
</tr>
<tr>
<td>lldiv()</td>
<td>stdlib.h</td>
<td>“ldiv() – lldiv() — Perform Long and Long Long Division” on page 201</td>
<td>Calculates the quotient and remainder of a long long integer.</td>
</tr>
<tr>
<td>log()</td>
<td>math.h</td>
<td>“log() — Calculate Natural Logarithm” on page 212</td>
<td>Calculates natural logarithm.</td>
</tr>
<tr>
<td>log10()</td>
<td>math.h</td>
<td>“log10() — Calculate Base 10 Logarithm” on page 213</td>
<td>Calculates base 10 logarithm.</td>
</tr>
<tr>
<td>modf()</td>
<td>math.h</td>
<td>“modf() — Separate Floating-Point Value” on page 244</td>
<td>Calculates the signed fractional portion of the argument.</td>
</tr>
<tr>
<td>nextafter()</td>
<td>math.h</td>
<td>“nextafter() – nextafterl() – nexttoward() – nexttowardl() — Calculate the Next Representable Floating-Point Value” on page 245</td>
<td>Calculates the next representable floating-point value.</td>
</tr>
<tr>
<td>nextafterl()</td>
<td>math.h</td>
<td>“nextafter() – nextafterl() – nexttoward() – nexttowardl() — Calculate the Next Representable Floating-Point Value” on page 245</td>
<td>Calculates the next representable floating-point value.</td>
</tr>
<tr>
<td>Function</td>
<td>Header File</td>
<td>Page</td>
<td>Description</td>
</tr>
<tr>
<td>-------------------</td>
<td>-------------</td>
<td>----------------------------------------------------------------------</td>
<td>---------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>nexttoward()</td>
<td>math.h</td>
<td>“nextafter() – nextafterl() – nexttoward() – nexttowardl() — Calculate the Next Representable Floating-Point Value” on page 245</td>
<td>Calculates the next representable floating-point value.</td>
</tr>
<tr>
<td>nexttowardl()</td>
<td>math.h</td>
<td>“nextafter() – nextafterl() – nexttoward() – nexttowardl() — Calculate the Next Representable Floating-Point Value” on page 245</td>
<td>Calculates the next representable floating-point value.</td>
</tr>
<tr>
<td>pow()</td>
<td>math.h</td>
<td>“pow() — Compute Power” on page 250</td>
<td>Calculates the value of an argument raised to a power.</td>
</tr>
<tr>
<td>quantexpd32()</td>
<td>math.h</td>
<td>“quantexpd32() - quantexpd64() - quantexpd128() — Compute the Quantum Exponent” on page 270</td>
<td>Compute the quantum exponent of a single-precision decimal floating-point value.</td>
</tr>
<tr>
<td>quantexpd64()</td>
<td>math.h</td>
<td>“quantexpd32() - quantexpd64() - quantexpd128() — Compute the Quantum Exponent” on page 270</td>
<td>Compute the quantum exponent of a double-precision decimal floating-point value.</td>
</tr>
<tr>
<td>quantexpd128()</td>
<td>math.h</td>
<td>“quantexpd32() - quantexpd64() - quantexpd128() — Compute the Quantum Exponent” on page 270</td>
<td>Compute the quantum exponent of a quad-precision decimal floating-point value.</td>
</tr>
<tr>
<td>Function</td>
<td>Header File</td>
<td>Page</td>
<td>Description</td>
</tr>
<tr>
<td>------------------</td>
<td>-------------</td>
<td>----------------------------------------------------------------------</td>
<td>-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>quantized32()</td>
<td>math.h</td>
<td>“quantized32() - quantized64() - quantized128() — Set the Quantum Exponent of X to the Quantum Exponent of Y” on page 271</td>
<td>Set the quantum exponent of a single-precision decimal floating-point value to the quantum exponent of another single-precision decimal floating-point value.</td>
</tr>
<tr>
<td>quantized64()</td>
<td>math.h</td>
<td>“quantized32() - quantized64() - quantized128() — Set the Quantum Exponent of X to the Quantum Exponent of Y” on page 271</td>
<td>Set the quantum exponent of a double-precision decimal floating-point value to the quantum exponent of another double-precision decimal floating-point value.</td>
</tr>
<tr>
<td>quantized128()</td>
<td>math.h</td>
<td>“quantized32() - quantized64() - quantized128() — Set the Quantum Exponent of X to the Quantum Exponent of Y” on page 271</td>
<td>Set the quantum exponent of a quad-precision decimal floating-point value to the quantum exponent of another quad-precision decimal floating-point value.</td>
</tr>
<tr>
<td>samequantumd32()</td>
<td>math.h</td>
<td>“samequantumd32() - samequantumd64() - samequantumd128() — Determine if Quantum Exponents X and Y are the Same” on page 356</td>
<td>Determine if the quantum exponents of two single-precision decimal floating-point values are the same.</td>
</tr>
<tr>
<td>samequantumd64()</td>
<td>math.h</td>
<td>“samequantumd32() - samequantumd64() - samequantumd128() — Determine if Quantum Exponents X and Y are the Same” on page 356</td>
<td>Determine if the quantum exponents of two double-precision decimal floating-point values are the same.</td>
</tr>
<tr>
<td>Function</td>
<td>Header File</td>
<td>Page</td>
<td>Description</td>
</tr>
<tr>
<td>--------------------------</td>
<td>-------------</td>
<td>-------------------------------------------</td>
<td>-------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>samequantum32()</td>
<td>math.h</td>
<td>“samequantum d32() - samequantum d64() -</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>samequantum d128() — Determine if Quantum</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Exponents X and Y are the Same” on page</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>356</td>
<td>Determine if the quantum exponents of two quad-precision decimal floating-point values are the same.</td>
</tr>
<tr>
<td>sqrt()</td>
<td>math.h</td>
<td>“sqrt() — Calculate Square Root” on page 380</td>
<td>Calculates the square root of a number.</td>
</tr>
</tbody>
</table>

**Trigonometric Functions**

<table>
<thead>
<tr>
<th>Function</th>
<th>Header File</th>
<th>Page</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>acos()</td>
<td>math.h</td>
<td>“acos() — Calculate Arccosine” on page 62</td>
<td>Calculates the arc cosine.</td>
</tr>
<tr>
<td>asin()</td>
<td>math.h</td>
<td>“asin() — Calculate Arcsine” on page 66</td>
<td>Calculates the arc sine.</td>
</tr>
<tr>
<td>atan()</td>
<td>math.h</td>
<td>“atan() – atan2() – Calculate Arctangent” on page 69</td>
<td>Calculates the arc tangent.</td>
</tr>
<tr>
<td>atan2()</td>
<td>math.h</td>
<td>“atan() – atan2() – Calculate Arctangent” on page 69</td>
<td>Calculates the arc tangent.</td>
</tr>
<tr>
<td>cos()</td>
<td>math.h</td>
<td>“cos() — Calculate Cosine” on page 89</td>
<td>Calculates the cosine.</td>
</tr>
<tr>
<td>cosh()</td>
<td>math.h</td>
<td>“cosh() — Calculate Hyperbolic Cosine” on page 90</td>
<td>Calculates the hyperbolic cosine.</td>
</tr>
<tr>
<td>sin()</td>
<td>math.h</td>
<td>“sin() — Calculate Sine” on page 376</td>
<td>Calculates the sine.</td>
</tr>
</tbody>
</table>
### Function

<table>
<thead>
<tr>
<th>Function</th>
<th>Header File</th>
<th>Page</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>sinh()</td>
<td>math.h</td>
<td>“sinh() — Calculate Hyperbolic Sine” on page 377</td>
<td>Calculates the hyperbolic sine.</td>
</tr>
<tr>
<td>tan()</td>
<td>math.h</td>
<td>“tan() — Calculate Tangent” on page 439</td>
<td>Calculates the tangent.</td>
</tr>
<tr>
<td>tanh()</td>
<td>math.h</td>
<td>“tanh() — Calculate Hyperbolic Tangent” on page 440</td>
<td>Calculates the hyperbolic tangent.</td>
</tr>
</tbody>
</table>

### Bessel Functions

<table>
<thead>
<tr>
<th>Function</th>
<th>Header File</th>
<th>Page</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>j0()</td>
<td>math.h</td>
<td>“Bessel Functions” on page 75</td>
<td>0 order differential equation of the first kind.</td>
</tr>
<tr>
<td>j1()</td>
<td>math.h</td>
<td>“Bessel Functions” on page 75</td>
<td>1st order differential equation of the first kind.</td>
</tr>
<tr>
<td>jn()</td>
<td>math.h</td>
<td>“Bessel Functions” on page 75</td>
<td>nth order differential equation of the first kind.</td>
</tr>
<tr>
<td>y0()</td>
<td>math.h</td>
<td>“Bessel Functions” on page 75</td>
<td>0 order differential equation of the second kind.</td>
</tr>
<tr>
<td>y1()</td>
<td>math.h</td>
<td>“Bessel Functions” on page 75</td>
<td>1st order differential equation of the second kind.</td>
</tr>
<tr>
<td>yn()</td>
<td>math.h</td>
<td>“Bessel Functions” on page 75</td>
<td>nth order differential equation of the second kind.</td>
</tr>
</tbody>
</table>

### Time Manipulation

<table>
<thead>
<tr>
<th>Function</th>
<th>Header File</th>
<th>Page</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>asctime()</td>
<td>time.h</td>
<td>“asctime() — Convert Time to Character String” on page 63</td>
<td>Converts time stored as a structure to a character string in storage.</td>
</tr>
<tr>
<td>Function</td>
<td>Header File</td>
<td>Page</td>
<td>Description</td>
</tr>
<tr>
<td>---------------</td>
<td>-------------</td>
<td>----------------------------------------------------------------------</td>
<td>------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>asctime_r()</td>
<td>time.h</td>
<td>“asctime_r() — Convert Time to Character String (Restartable)” on page 65</td>
<td>Converts time stored as a structure to a character string in storage. (Restartable version of asctime())</td>
</tr>
<tr>
<td>clock()</td>
<td>time.h</td>
<td>“clock() — Determine Processor Time” on page 88</td>
<td>Determines processor time.</td>
</tr>
<tr>
<td>ctime()</td>
<td>time.h</td>
<td>“ctime() — Convert Time to Character String” on page 96</td>
<td>Converts time stored as a long value to a character string.</td>
</tr>
<tr>
<td>ctime64()</td>
<td>time.h</td>
<td>“ctime64() — Convert Time to Character String” on page 98</td>
<td>Converts time stored as a long long value to a character string.</td>
</tr>
<tr>
<td>ctime_r()</td>
<td>time.h</td>
<td>“ctime_r() — Convert Time to Character String (Restartable)” on page 100</td>
<td>Converts time stored as a long value to a character string. (Restartable version of ctime())</td>
</tr>
<tr>
<td>ctime64_r()</td>
<td>time.h</td>
<td>“ctime64_r() — Convert Time to Character String (Restartable)” on page 101</td>
<td>Converts time stored as a long long value to a character string. (Restartable version of ctime64())</td>
</tr>
<tr>
<td>difftime()</td>
<td>time.h</td>
<td>“difftime() — Compute Time Difference” on page 108</td>
<td>Calculates the difference between two times.</td>
</tr>
<tr>
<td>difftime64()</td>
<td>time.h</td>
<td>“difftime64() — Compute Time Difference” on page 109</td>
<td>Calculates the difference between two times.</td>
</tr>
<tr>
<td>gmtime()</td>
<td>time.h</td>
<td>“gmtime() — Convert Time” on page 183</td>
<td>Converts time to Coordinated Universal Time structure.</td>
</tr>
<tr>
<td>gmtime_r()</td>
<td>time.h</td>
<td>“gmtime_r() — Convert Time (Restartable)” on page 187</td>
<td>Converts time to Coordinated Universal Time structure. (Restartable version of gmtime())</td>
</tr>
<tr>
<td>Function</td>
<td>Header File</td>
<td>Page</td>
<td>Description</td>
</tr>
<tr>
<td>------------------</td>
<td>-------------</td>
<td>---------------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>gmtime64()</td>
<td>time.h</td>
<td>“gmtime64() — Convert Time” on page 185</td>
<td>Converts time to Coordinated Universal Time structure.</td>
</tr>
<tr>
<td>gmtime64_r()</td>
<td>time.h</td>
<td>“gmtime64_r() — Convert Time (Restartable)” on page 189</td>
<td></td>
</tr>
<tr>
<td>localtime()</td>
<td>time.h</td>
<td>“localtime() — Convert Time” on page 207</td>
<td>Converts time to local time.</td>
</tr>
<tr>
<td>localtime64()</td>
<td>time.h</td>
<td>“localtime64() — Convert Time” on page 208</td>
<td>Converts time to local time.</td>
</tr>
<tr>
<td>localtime_r()</td>
<td>time.h</td>
<td>“localtime_r() — Convert Time (Restartable)” on page 210</td>
<td>Converts time to local time. (Restartable version of localtime())</td>
</tr>
<tr>
<td>localtime64_r()</td>
<td>time.h</td>
<td>“localtime64_r() — Convert Time (Restartable)” on page 211</td>
<td>Converts time to local time. (Restartable version of localtime64())</td>
</tr>
<tr>
<td>mktime()</td>
<td>time.h</td>
<td>“mktime() — Convert Local Time” on page 241</td>
<td>Converts local time into calendar time.</td>
</tr>
<tr>
<td>mktime64()</td>
<td>time.h</td>
<td>“mktime64() — Convert Local Time” on page 242</td>
<td>Converts local time into calendar time.</td>
</tr>
<tr>
<td>time()</td>
<td>time.h</td>
<td>“time() — Determine Current Time” on page 441</td>
<td>Returns the time in seconds.</td>
</tr>
<tr>
<td>time64()</td>
<td>time.h</td>
<td>“time64() — Determine Current Time” on page 443</td>
<td>Returns the time in seconds.</td>
</tr>
</tbody>
</table>

**Type Conversion**

<table>
<thead>
<tr>
<th>Function</th>
<th>Header File</th>
<th>Page</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>atof()</td>
<td>stdlib.h</td>
<td>“atof() — Convert Character String to Float” on page 71</td>
<td>Converts a character string to a floating-point value.</td>
</tr>
<tr>
<td>Function</td>
<td>Header File</td>
<td>Page</td>
<td>Description</td>
</tr>
<tr>
<td>----------</td>
<td>-------------</td>
<td>------</td>
<td>-------------</td>
</tr>
<tr>
<td>atoi()</td>
<td>stdlib.h</td>
<td>“atoi() — Convert Character String to Integer” on page 72</td>
<td>Converts a character string to an integer.</td>
</tr>
<tr>
<td>atol()</td>
<td>stdlib.h</td>
<td>“atol() — atoll() — Convert Character String to Long or Long Long Integer” on page 74</td>
<td>Converts a character string to a long integer.</td>
</tr>
<tr>
<td>atoll()</td>
<td>stdlib.h</td>
<td>“atol() — atoll() — Convert Character String to Long or Long Long Integer” on page 74</td>
<td>Converts a character string to a long integer.</td>
</tr>
<tr>
<td>_gcvt()</td>
<td>stdlib.h</td>
<td>“_gcvt() — Convert Floating-Point to String” on page 173</td>
<td>Converts a floating-point value to a string.</td>
</tr>
<tr>
<td>_itoa()</td>
<td>stdlib.h</td>
<td>“_itoa() — Convert Integer to String” on page 198</td>
<td>Converts an integer to a string.</td>
</tr>
<tr>
<td>_ltoa()</td>
<td>stdlib.h</td>
<td>“_ltoa() — Convert Long Integer to String” on page 214</td>
<td>Converts a long integer to a string.</td>
</tr>
<tr>
<td>strtod()</td>
<td>stdlib.h</td>
<td>“strtod() - strtof() - strtold() — Convert Character String to Double, Float, and Long Double” on page 422</td>
<td>Converts a character string to a double-precision binary floating-point value.</td>
</tr>
<tr>
<td>Function</td>
<td>Header File</td>
<td>Page</td>
<td>Description</td>
</tr>
<tr>
<td>--------------</td>
<td>-------------</td>
<td>----------------------------------------------------------------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td><code>strtod32()</code></td>
<td>stplib.h</td>
<td>“strtod32() - strtod64() - strtod128() — Convert Character String to Decimal Floating-Point” on page 425</td>
<td>Converts a character string to a single-precision decimal floating-point value.</td>
</tr>
<tr>
<td><code>strtod64()</code></td>
<td>stplib.h</td>
<td>“strtod32() - strtod64() - strtod128() — Convert Character String to Decimal Floating-Point” on page 425</td>
<td>Converts a character string to a double-precision decimal floating-point value.</td>
</tr>
<tr>
<td><code>strtod128()</code></td>
<td>stplib.h</td>
<td>“strtod32() - strtod64() - strtod128() — Convert Character String to Decimal Floating-Point” on page 425</td>
<td>Converts a character string to a quad-precision decimal floating-point value.</td>
</tr>
<tr>
<td><code>strtof()</code></td>
<td>stplib.h</td>
<td>“strtod() - strtof() - strtold() — Convert Character String to Double, Float, and Long Double” on page 422</td>
<td>Converts a character string to a binary floating-point value.</td>
</tr>
<tr>
<td><code>strtol()</code></td>
<td>stdlib.h</td>
<td>“strtol() — strtoll() — Convert Character String to Long and Long Long Integer” on page 430</td>
<td>Converts a character string to a long integer.</td>
</tr>
<tr>
<td>Function</td>
<td>Header File</td>
<td>Page</td>
<td>Description</td>
</tr>
<tr>
<td>--------------</td>
<td>-------------</td>
<td>----------------------------------------------------------------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td><code>strtold()</code></td>
<td>stdlib.h</td>
<td>“strtod() - strtold() — Convert Character String to Double, Float, and Long Double” on page 422</td>
<td>Converts a character string to a double-precision binary floating-point value.</td>
</tr>
<tr>
<td><code>strtoll()</code></td>
<td>stdlib.h</td>
<td>“strtoll() — Convert Character String to Long and Long Long Integer” on page 430</td>
<td>Converts a character string to a long long integer.</td>
</tr>
<tr>
<td><code>strtoul()</code></td>
<td>stdlib.h</td>
<td>“strtoul() — Convert Character String to Unsigned Long and Unsigned Long Long Integer” on page 432</td>
<td>Converts a string to an unsigned long integer.</td>
</tr>
<tr>
<td><code>strtoull()</code></td>
<td>stdlib.h</td>
<td>“strtoull() — Convert Character String to Unsigned Long and Unsigned Long Long Integer” on page 432</td>
<td>Converts a string to an unsigned long long integer.</td>
</tr>
<tr>
<td><code>toascii()</code></td>
<td>ctype.h</td>
<td>“toascii() — Convert Character to Character Representable by ASCII” on page 446</td>
<td>Converts a character to the corresponding ASCII value.</td>
</tr>
<tr>
<td><code>_ultoa()</code></td>
<td>stdlib.h</td>
<td>“ultoa() — Convert Unsigned Long Integer to String” on page 451</td>
<td>Converts an unsigned long integer to a string.</td>
</tr>
<tr>
<td>Function</td>
<td>Header File</td>
<td>Page</td>
<td>Description</td>
</tr>
<tr>
<td>--------------</td>
<td>-------------</td>
<td>----------------------------------------------------------------------</td>
<td>-----------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>wcstod()</td>
<td>wchar.h</td>
<td>“wcstod() - wcstof() - wcstold() — Convert Wide-Character String to Double, Float, and Long Double” on page 510</td>
<td>Converts a wide-character string to a double-precision binary floating-point value.</td>
</tr>
<tr>
<td>wcstod32()</td>
<td>wchar.h</td>
<td>“wcstod32() - wcstod64() - wcstod128() — Convert Wide-Character String to Decimal Floating-Point” on page 512</td>
<td>Converts a wide-character string to a single-precision decimal floating-point value.</td>
</tr>
<tr>
<td>wcstod64()</td>
<td>wchar.h</td>
<td>“wcstod32() - wcstod64() - wcstod128() — Convert Wide-Character String to Decimal Floating-Point” on page 512</td>
<td>Converts a wide-character string to a double-precision decimal floating-point value.</td>
</tr>
<tr>
<td>wcstod128()</td>
<td>wchar.h</td>
<td>“wcstod32() - wcstod64() - wcstod128() — Convert Wide-Character String to Decimal Floating-Point” on page 512</td>
<td>Converts a wide-character string to a quad-precision decimal floating-point value.</td>
</tr>
<tr>
<td>wcstof()</td>
<td>wchar.h</td>
<td>“wcstod() - wcstof() - wcstold() — Convert Wide-Character String to Double, Float, and Long Double” on page 510</td>
<td>Converts a wide-character string to a binary floating-point value.</td>
</tr>
<tr>
<td>Function</td>
<td>Header File</td>
<td>Page</td>
<td>Description</td>
</tr>
<tr>
<td>--------------</td>
<td>-------------</td>
<td>-------------------------------------------</td>
<td>------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>wcstol()</td>
<td>wchar.h</td>
<td>“wcstol() — wcstoll() — Convert Wide Character String to Long and Long Long Integer” on page 515</td>
<td>Converts a wide-character string to a long integer.</td>
</tr>
<tr>
<td>wcstold()</td>
<td>wchar.h</td>
<td>“wcstold() — wcstof() — wcstold() — Convert Wide-Character String to Double, Float, and Long Double” on page 510</td>
<td>Converts a wide-character string to a double-precision binary floating-point value.</td>
</tr>
<tr>
<td>wcstoll()</td>
<td>wchar.h</td>
<td>“wcstol() — wcstoll() — Convert Wide Character String to Long and Long Long Integer” on page 515</td>
<td>Converts a wide-character string to a long long integer.</td>
</tr>
<tr>
<td>wcstoul()</td>
<td>wchar.h</td>
<td>“wcstoull() — wcstoul() — Convert Wide Character String to Unsigned Long and Unsigned Long Long Integer” on page 520</td>
<td>Converts a wide-character string to an unsigned long integer.</td>
</tr>
<tr>
<td>wcstoull()</td>
<td>wchar.h</td>
<td>“wcstoull() — wcstoul() — Convert Wide Character String to Unsigned Long and Unsigned Long Long Integer” on page 520</td>
<td>Converts a wide-character string to an unsigned long long integer.</td>
</tr>
</tbody>
</table>
### Conversion

<table>
<thead>
<tr>
<th>Function</th>
<th>Header File</th>
<th>Page</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>QXXDTOP()</td>
<td>xxcvt.h</td>
<td>“QXXDTOP() — Convert Double to Packed Decimal” on page 275</td>
<td>Converts a floating-point value to a packed decimal value.</td>
</tr>
<tr>
<td>QXXDTOZ()</td>
<td>xxcvt.h</td>
<td>“QXXDTOZ() — Convert Double to Zoned Decimal” on page 276</td>
<td>Converts a floating-point value to a zoned decimal value.</td>
</tr>
<tr>
<td>QXXITOP()</td>
<td>xxcvt.h</td>
<td>“QXXITOP() — Convert Integer to Packed Decimal” on page 277</td>
<td>Converts an integer value to a packed decimal value.</td>
</tr>
<tr>
<td>QXXITOZ()</td>
<td>xxcvt.h</td>
<td>“QXXITOZ() — Convert Integer to Zoned Decimal” on page 277</td>
<td>Converts an integer value to a zoned decimal value.</td>
</tr>
<tr>
<td>QXXPTOD()</td>
<td>xxcvt.h</td>
<td>“QXXPTOD() — Convert Packed Decimal to Double” on page 278</td>
<td>Converts a packed decimal value to a floating-point value.</td>
</tr>
<tr>
<td>QXXPTOI()</td>
<td>xxcvt.h</td>
<td>“QXXPTOI() — Convert Packed Decimal to Integer” on page 279</td>
<td>Converts a packed decimal value to an integer value.</td>
</tr>
<tr>
<td>QXXZTOD()</td>
<td>xxcvt.h</td>
<td>“QXXZTOD() — Convert Zoned Decimal to Double” on page 281</td>
<td>Converts a zoned decimal value to a floating-point value.</td>
</tr>
<tr>
<td>QXXZTOI()</td>
<td>xxcvt.h</td>
<td>“QXXZTOI() — Convert Zoned Decimal to Integer” on page 282</td>
<td>Converts a zoned decimal value to an integer value.</td>
</tr>
</tbody>
</table>

### Record Input/Output

<table>
<thead>
<tr>
<th>Function</th>
<th>Header File</th>
<th>Page</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>_Racquire()</td>
<td>recio.h</td>
<td>“_Racquire() — Acquire a Program Device” on page 285</td>
<td>Prepares a device for record I/O operations.</td>
</tr>
<tr>
<td>Function</td>
<td>Header File</td>
<td>Page</td>
<td>Description</td>
</tr>
<tr>
<td>------------</td>
<td>---------------</td>
<td>-----------------------------------------------------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>_Rclose()</td>
<td>recio.h</td>
<td>“_Rclose() — Close a File” on page 286</td>
<td>Closes a file that is opened for record I/O operations.</td>
</tr>
<tr>
<td>_Rcommit()</td>
<td>recio.h</td>
<td>“_Rcommit() — Commit Current Record” on page 287</td>
<td>Completes the current transaction, and establishes a new commitment boundary.</td>
</tr>
<tr>
<td>_Rdelete()</td>
<td>recio.h</td>
<td>“_Rdelete() — Delete a Record” on page 289</td>
<td>Deletes the currently locked record.</td>
</tr>
<tr>
<td>_Rdevatr()</td>
<td>recio.h /xxfdbk.h</td>
<td>“_Rdevatr() — Get Device Attributes” on page 290</td>
<td>Returns a pointer to a copy of the device attributes feedback area for the file reference by fp and the device pgmdev.</td>
</tr>
<tr>
<td>_Rfeod()</td>
<td>recio.h</td>
<td>“_Rfeod() — Force the End-of-Data” on page 304</td>
<td>Forces an end-of-file condition for the file referenced by fp.</td>
</tr>
<tr>
<td>_Rfeov()</td>
<td>recio.h</td>
<td>“_Rfeov() — Force the End-of-File” on page 305</td>
<td>Forces an end-of-volume condition for tapes.</td>
</tr>
<tr>
<td>_Rformat()</td>
<td>recio.h</td>
<td>“_Rformat() — Set the Record Format Name” on page 306</td>
<td>Sets the record format to fmt for the file referenced by fp.</td>
</tr>
<tr>
<td>_Rindara()</td>
<td>recio.h</td>
<td>“_Rindara() — Set Separate Indicator Area” on page 308</td>
<td>Sets up the separate indicator area to be used for subsequent record I/O operations.</td>
</tr>
<tr>
<td>_Riofbk()</td>
<td>recio.h /xxfdbk.h</td>
<td>“_Riofbk() — Obtain I/O Feedback Information” on page 310</td>
<td>Returns a pointer to a copy of the I/O feedback area for the file referenced by fp.</td>
</tr>
<tr>
<td>_Rlocate()</td>
<td>recio.h</td>
<td>“_Rlocate() — Position a Record” on page 312</td>
<td>Positions to the record in the files associated with fp and specified by the key, klen_rrn and opt parameters.</td>
</tr>
<tr>
<td>_Ropen()</td>
<td>recio.h</td>
<td>“_Ropen() — Open a Record File for I/O Operations” on page 315</td>
<td>Opens a file for record I/O operations.</td>
</tr>
<tr>
<td>Function</td>
<td>Header File</td>
<td>Page</td>
<td>Description</td>
</tr>
<tr>
<td>---------------</td>
<td>-------------------</td>
<td>----------------------------------------------------------------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>_Ropnfbk()</td>
<td>recio.h</td>
<td>“_Ropnfbk() — Obtain Open Feedback Information” on page 319</td>
<td>Returns a pointer to a copy of the open feedback area for the file referenced by fp.</td>
</tr>
<tr>
<td>_Rpgmdev()</td>
<td>recio.h</td>
<td>“_Rpgmdev() — Set Default Program Device” on page 320</td>
<td>Sets the default program device.</td>
</tr>
<tr>
<td>_Rreadd()</td>
<td>recio.h</td>
<td>“_Rreadd() — Read a Record by Relative Record Number” on page 321</td>
<td>Reads a record by relative record number.</td>
</tr>
<tr>
<td>_Rreadf()</td>
<td>recio.h</td>
<td>“_Rreadf() — Read the First Record” on page 323</td>
<td>Reads the first record.</td>
</tr>
<tr>
<td>_Rreadindv()</td>
<td>recio.h</td>
<td>“_Rreadindv() — Read from an Invited Device” on page 325</td>
<td>Reads data from an invited device.</td>
</tr>
<tr>
<td>_Rreadk()</td>
<td>recio.h</td>
<td>“_Rreadk() — Read a Record by Key” on page 328</td>
<td>Reads a record by key.</td>
</tr>
<tr>
<td>_Rreadl()</td>
<td>recio.h</td>
<td>“_Rreadl() — Read the Last Record” on page 331</td>
<td>Reads the last record.</td>
</tr>
<tr>
<td>_Rreadn()</td>
<td>recio.h</td>
<td>“_Rreadn() — Read the Next Record” on page 332</td>
<td>Reads the next record.</td>
</tr>
<tr>
<td>_Rreadnc()</td>
<td>recio.h</td>
<td>“_Rreadnc() — Read the Next Changed Record in a Subfile” on page 334</td>
<td>Reads the next changed record in the subfile.</td>
</tr>
<tr>
<td>_Rreadp()</td>
<td>recio.h</td>
<td>“_Rreadp() — Read the Previous Record” on page 337</td>
<td>Reads the previous record.</td>
</tr>
<tr>
<td>Function</td>
<td>Header File</td>
<td>Page</td>
<td>Description</td>
</tr>
<tr>
<td>---------------</td>
<td>-------------</td>
<td>------</td>
<td>-----------------------------------------------------</td>
</tr>
<tr>
<td>_Rreads()</td>
<td>recio.h</td>
<td>“_Rreads() — Read the Same Record” on page 339</td>
<td>Reads the same record.</td>
</tr>
<tr>
<td>_Rrelease()</td>
<td>recio.h</td>
<td>“_Rrelease() — Release a Program Device” on page 341</td>
<td>Makes the specified device ineligible for record I/O operations.</td>
</tr>
<tr>
<td>_Rrlslck()</td>
<td>recio.h</td>
<td>“_Rrlslck() — Release a Record Lock” on page 342</td>
<td>Releases the currently locked record.</td>
</tr>
<tr>
<td>_Rrollbck()</td>
<td>recio.h</td>
<td>“_Rrollbck() — Roll Back Commitment Control Changes” on page 343</td>
<td>Reestablishes the last commitment boundary as the current commitment boundary.</td>
</tr>
<tr>
<td>_Rupdate()</td>
<td>recio.h</td>
<td>“_Rupdate() — Update a Record” on page 345</td>
<td>Writes to the record that is currently locked for update.</td>
</tr>
<tr>
<td>_Rupfb()</td>
<td>recio.h</td>
<td>“_Rupfb() — Provide Information on Last I/O Operation” on page 346</td>
<td>Updates the feedback structure with information about the last record I/O operation.</td>
</tr>
<tr>
<td>_Rwrite()</td>
<td>recio.h</td>
<td>“_Rwrite() — Write the Next Record” on page 348</td>
<td>Writes a record to the end of the file.</td>
</tr>
<tr>
<td>_Rwrited()</td>
<td>recio.h</td>
<td>“_Rwrited() — Write a Record Directly” on page 350</td>
<td>Writes a record by relative record number. It will only write over deleted records.</td>
</tr>
<tr>
<td>_Rwriterd()</td>
<td>recio.h</td>
<td>“_Rwriterd() — Write and Read a Record” on page 353</td>
<td>Writes and reads a record.</td>
</tr>
<tr>
<td>_Rwrread()</td>
<td>recio.h</td>
<td>“_Rwrread() — Write and Read a Record (separate buffers)” on page 354</td>
<td>Functions as _Rwriterd(), except separate buffers can be specified for input and output data.</td>
</tr>
</tbody>
</table>
### Stream Input/Output

#### Formatted Input/Output

<table>
<thead>
<tr>
<th>Function</th>
<th>Header File</th>
<th>Page</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>fprintf()</td>
<td>stdio.h</td>
<td>“fprintf() — Write Formatted Data to a Stream” on page 141</td>
<td>Formats and prints characters to the output stream.</td>
</tr>
<tr>
<td>fscanf()</td>
<td>stdio.h</td>
<td>“fscanf() — Read Formatted Data” on page 155</td>
<td>Reads data from a stream into locations given by arguments.</td>
</tr>
<tr>
<td>fwprintf()</td>
<td>stdio.h</td>
<td>“fwprintf() — Format Data as Wide Characters and Write to a Stream” on page 165</td>
<td>Formats data as wide characters, and writes to a stream.</td>
</tr>
<tr>
<td>fwscanf()</td>
<td>stdio.h</td>
<td>“fwscanf() — Read Data from Stream Using Wide Character” on page 169</td>
<td>Reads wide data from stream into locations given by arguments.</td>
</tr>
<tr>
<td>printf()</td>
<td>stdio.h</td>
<td>“printf() — Print Formatted Characters” on page 251</td>
<td>Formats and prints characters to stdout.</td>
</tr>
<tr>
<td>scanf()</td>
<td>stdio.h</td>
<td>“scanf() — Read Data” on page 358</td>
<td>Reads data from stdin into locations given by arguments.</td>
</tr>
<tr>
<td>snprintf()</td>
<td>stdio.h</td>
<td>“ snprintf() — Print Formatted Data to Buffer” on page 378</td>
<td>Same as sprintf, except that the snprintf() function will stop after n characters have been written to a buffer.</td>
</tr>
<tr>
<td>sprintf()</td>
<td>stdio.h</td>
<td>“sprintf() — Print Formatted Data to Buffer” on page 379</td>
<td>Formats and writes characters to a buffer.</td>
</tr>
<tr>
<td>sscanf()</td>
<td>stdio.h</td>
<td>“sscanf() — Read Data” on page 382</td>
<td>Reads data from a buffer into locations given by arguments.</td>
</tr>
<tr>
<td>swprintf()</td>
<td>wchar.h</td>
<td>“swprintf() — Format and Write Wide Characters to Buffer” on page 435</td>
<td>Formats and writes wide characters to buffer.</td>
</tr>
<tr>
<td>Function</td>
<td>Header File</td>
<td>Page</td>
<td>Description</td>
</tr>
<tr>
<td>---------------</td>
<td>-------------</td>
<td>-------------------------------------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>swscanf()</td>
<td>wchar.h</td>
<td>“swscanf() — Read Wide Character Data” on page 437</td>
<td>Reads wide data from a buffer into locations given by arguments.</td>
</tr>
<tr>
<td>vfprintf()</td>
<td>stdio.h</td>
<td>“vfprintf() — Print Argument Data to Stream” on page 457</td>
<td>Formats and prints characters to the output stream using a variable number of arguments.</td>
</tr>
<tr>
<td>vfscanf()</td>
<td>stdarg.h stdio.h</td>
<td>“vfscanf() — Read Formatted Data” on page 458</td>
<td>Reads data from a specified stream into locations given by a variable number of arguments.</td>
</tr>
<tr>
<td>vfwprintf()</td>
<td>stdio.h stdarg.h</td>
<td>“vfprintf() — Format Argument Data as Wide Characters and Write to a Stream” on page 460</td>
<td>Formats argument data as wide characters and writes to a stream using a variable number of arguments.</td>
</tr>
<tr>
<td>vfwscanf()</td>
<td>stdarg.h stdio.h</td>
<td>“vfwscanf() — Read Formatted Wide Character Data” on page 462</td>
<td>Reads wide data from a specified stream into locations given by a variable number of arguments.</td>
</tr>
<tr>
<td>vprintf()</td>
<td>stdarg.h stdio.h</td>
<td>“vprintf() — Print Argument Data” on page 464</td>
<td>Formats and writes characters to stdout using a variable number of arguments.</td>
</tr>
<tr>
<td>scanf()</td>
<td>stdio.h</td>
<td>“scanf() — Read Formatted Data” on page 457</td>
<td>Reads data from stdin into locations given by a variable number of arguments.</td>
</tr>
<tr>
<td>vsnprintf()</td>
<td>stdio.h stdarg.h</td>
<td>“vsnprintf() — Print Argument Data to Buffer” on page 467</td>
<td>Same as vsprintf, except that the vsnprintf function will stop after n characters have been written to a buffer.</td>
</tr>
<tr>
<td>vsprintf()</td>
<td>stdarg.h stdio.h</td>
<td>“vsprintf() — Print Argument Data to Buffer” on page 468</td>
<td>Formats and writes characters to a buffer using a variable number of arguments.</td>
</tr>
<tr>
<td>vsscanf()</td>
<td>stdarg.h stdio.h</td>
<td>“vsscanf() — Read Formatted Data” on page 460</td>
<td>Reads data from a buffer into locations given by a variable number of arguments.</td>
</tr>
<tr>
<td>Function</td>
<td>Header File</td>
<td>Page</td>
<td>Description</td>
</tr>
<tr>
<td>-------------</td>
<td>-------------</td>
<td>-------------------------------------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>vswprintf()</td>
<td>wchar.h, stdarg.h</td>
<td>“vswprintf() — Format and Write Wide Characters to Buffer” on page 471</td>
<td>Formats and writes wide characters to buffer using a variable number of arguments.</td>
</tr>
<tr>
<td>vswscanf()</td>
<td>stdarg.h, wchar.h</td>
<td>“vswscanf() — Read Formatted Wide Character Data” on page 473</td>
<td>Reads wide data from a buffer into locations given by a variable number of arguments.</td>
</tr>
<tr>
<td>vwprintf()</td>
<td>wchar.h, stdarg.h</td>
<td>“vwprintf() — Format Argument Data as Wide Characters and Print” on page 475</td>
<td>Formats and writes wide characters to stdout using a variable number of arguments.</td>
</tr>
<tr>
<td>vwscanf()</td>
<td>stdarg.h, stdio.h</td>
<td>“vwscanf() — Read Formatted Wide Character Data” on page 476</td>
<td>Reads wide data from stdin into locations given by a variable number of arguments.</td>
</tr>
<tr>
<td>wprintf()</td>
<td>stdio.h</td>
<td>“wprintf() — Format Data as Wide Characters and Print” on page 538</td>
<td>Formats and writes wide characters to stdout</td>
</tr>
<tr>
<td>wscanf()</td>
<td>stdio.h</td>
<td>“wscanf() — Read Data Using Wide-Character Format String” on page 539</td>
<td>Reads wide data from stdin into locations given by arguments.</td>
</tr>
</tbody>
</table>

**Character and String Input/Output**

<table>
<thead>
<tr>
<th>Function</th>
<th>Header File</th>
<th>Page</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>fgetc()</td>
<td>stdio.h</td>
<td>“fgetc() — Read a Character” on page 123</td>
<td>Reads a character from a specified input stream.</td>
</tr>
<tr>
<td>fgets()</td>
<td>stdio.h</td>
<td>“fgets() — Read a String” on page 126</td>
<td>Reads a string from a specified input stream.</td>
</tr>
<tr>
<td>fgetwc()</td>
<td>stdio.h</td>
<td>“fgetwc() — Read Wide Character from Stream” on page 127</td>
<td>Reads a wide character from a specified stream.</td>
</tr>
<tr>
<td>Function</td>
<td>Header File</td>
<td>Page</td>
<td>Description</td>
</tr>
<tr>
<td>------------</td>
<td>-------------</td>
<td>-------------------------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>fgetws()</td>
<td>stdio.h</td>
<td>“fgetws() — Read Wide-Character String from Stream” on page 129</td>
<td>Reads a wide-character string from a specified stream.</td>
</tr>
<tr>
<td>fputs()</td>
<td>stdio.h</td>
<td>“fputs() — Write String” on page 145</td>
<td>Prints a string to a specified output stream.</td>
</tr>
<tr>
<td>_fputchar()</td>
<td>stdio.h</td>
<td>“_fputchar() — Write Character” on page 144</td>
<td>Writes a character to stdout.</td>
</tr>
<tr>
<td>fputc()</td>
<td>stdio.h</td>
<td>“fputc() — Write Character” on page 142</td>
<td>Prints a character to a specified output stream.</td>
</tr>
<tr>
<td>_fputwc()</td>
<td>stdio.h</td>
<td>“_fputwc() — Write Wide Character” on page 146</td>
<td>Writes a wide character to a specified stream.</td>
</tr>
<tr>
<td>fputs()</td>
<td>stdio.h</td>
<td>“fputs() — Write String” on page 145</td>
<td>Prints a string to a specified output stream.</td>
</tr>
<tr>
<td>fputwc()</td>
<td>stdio.h</td>
<td>“fputwc() — Write Wide Character” on page 146</td>
<td>Writes a wide character to a specified stream.</td>
</tr>
<tr>
<td>fputws()</td>
<td>stdio.h</td>
<td>“fputws() — Write Wide-Character String” on page 148</td>
<td>Writes a wide-character string to a specified stream.</td>
</tr>
<tr>
<td>getc()</td>
<td>stdio.h</td>
<td>“getc() – getchar() — Read a Character” on page 174</td>
<td>Reads a character from a specified input stream.</td>
</tr>
<tr>
<td>getchar()</td>
<td>stdio.h</td>
<td>“getc() – getchar() — Read a Character” on page 174</td>
<td>Reads a character from stdin.</td>
</tr>
<tr>
<td>gets()</td>
<td>stdio.h</td>
<td>“gets() — Read a Line” on page 178</td>
<td>Reads a line from stdin.</td>
</tr>
<tr>
<td>getwc()</td>
<td>stdio.h</td>
<td>“getwc() — Read Wide Character from Stream” on page 179</td>
<td>Reads a wide character from a specified stream.</td>
</tr>
<tr>
<td>getwchar()</td>
<td>stdio.h</td>
<td>“getwchar() — Get Wide Character from stdin” on page 181</td>
<td>Gets a wide character from stdin.</td>
</tr>
<tr>
<td>Function</td>
<td>Header File</td>
<td>Page</td>
<td>Description</td>
</tr>
<tr>
<td>------------</td>
<td>-------------</td>
<td>----------------------------------------------------------------------</td>
<td>-------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td><code>putc()</code></td>
<td>stdio.h</td>
<td>“putc() – putchar() — Write a Character” on page 263</td>
<td>Prints a character to a specified output stream.</td>
</tr>
<tr>
<td><code>putchar()</code></td>
<td>stdio.h</td>
<td>“putc() – putchar() — Write a Character” on page 263</td>
<td>Prints a character to stdout.</td>
</tr>
<tr>
<td><code>puts()</code></td>
<td>stdio.h</td>
<td>“puts() — Write a String” on page 265</td>
<td>Prints a string to stdout.</td>
</tr>
<tr>
<td><code>putwc()</code></td>
<td>stdio.h</td>
<td>“putwc() — Write Wide Character” on page 266</td>
<td>Writes a wide character to a specified stream.</td>
</tr>
<tr>
<td><code>putwchar()</code></td>
<td>stdio.h</td>
<td>“putwchar() — Write Wide Character to stdout ” on page 268</td>
<td>Writes a wide character to stdout.</td>
</tr>
<tr>
<td><code>ungetc()</code></td>
<td>stdio.h</td>
<td>“ungetc() — Push Character onto Input Stream” on page 452</td>
<td>Pushes a character back onto a specified input stream.</td>
</tr>
<tr>
<td><code>ungetwc()</code></td>
<td>stdio.h</td>
<td>“ungetwc() — Push Wide Character onto Input Stream ” on page 453</td>
<td>Pushes a wide character back onto a specified input stream.</td>
</tr>
</tbody>
</table>

**Direct Input/Output**

<table>
<thead>
<tr>
<th>Function</th>
<th>Header File</th>
<th>Page</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>fread()</code></td>
<td>stdio.h</td>
<td>“fread() — Read Items” on page 149</td>
<td>Reads items from a specified input stream.</td>
</tr>
<tr>
<td><code>fwrite()</code></td>
<td>stdio.h</td>
<td>“fwrite() — Write Items” on page 168</td>
<td>Writes items to a specified output stream.</td>
</tr>
</tbody>
</table>

**File Positioning**

<table>
<thead>
<tr>
<th>Function</th>
<th>Header File</th>
<th>Page</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>fgetpos()</code></td>
<td>stdio.h</td>
<td>“fgetpos() — Get File Position” on page 124</td>
<td>Gets the current position of the file pointer.</td>
</tr>
<tr>
<td>Function</td>
<td>Header File</td>
<td>Page</td>
<td>Description</td>
</tr>
<tr>
<td>-----------</td>
<td>-------------</td>
<td>------</td>
<td>-------------</td>
</tr>
<tr>
<td>fseek()</td>
<td>stdio.h</td>
<td>“fseek() – fseeko() — Reposition File Position” on page 157</td>
<td>Moves the file pointer to a new location.</td>
</tr>
<tr>
<td>fseeko()</td>
<td>stdio.h</td>
<td>“fseek() – fseeko() — Reposition File Position” on page 157</td>
<td>Same as fseek().</td>
</tr>
<tr>
<td>fsetpos()</td>
<td>stdio.h</td>
<td>“fsetpos() — Set File Position” on page 159</td>
<td>Moves the file pointer to a new location.</td>
</tr>
<tr>
<td>ftell()</td>
<td>stdio.h</td>
<td>“ftell() – ftello() — Get Current Position” on page 161</td>
<td>Gets the current position of the file pointer.</td>
</tr>
<tr>
<td>ftello()</td>
<td>stdio.h</td>
<td>“ftell() – ftello() — Get Current Position” on page 161</td>
<td>Same as ftell().</td>
</tr>
<tr>
<td>rewind()</td>
<td>stdio.h</td>
<td>“rewind() — Adjust Current File Position” on page 303</td>
<td>Repositions the file pointer to the beginning of the file.</td>
</tr>
</tbody>
</table>

### File Access

<table>
<thead>
<tr>
<th>Function</th>
<th>Header File</th>
<th>Page</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>fclose()</td>
<td>stdio.h</td>
<td>“fclose() — Close Stream” on page 116</td>
<td>Closes a specified stream.</td>
</tr>
<tr>
<td>fdopen()</td>
<td>stdio.h</td>
<td>“fdopen() — Associates Stream With File Descriptor” on page 117</td>
<td>Associates an input or output stream with a file.</td>
</tr>
<tr>
<td>fflush()</td>
<td>stdio.h</td>
<td>“fflush() — Write Buffer to File” on page 121</td>
<td>Causes the system to write the contents of a buffer to a file.</td>
</tr>
<tr>
<td>fopen()</td>
<td>stdio.h</td>
<td>“fopen() — Open Files” on page 134</td>
<td>Opens a specified stream.</td>
</tr>
<tr>
<td>freopen()</td>
<td>stdio.h</td>
<td>“freopen() — Redirect Open Files” on page 153</td>
<td>Closes a file and reassigns a stream.</td>
</tr>
<tr>
<td>Function</td>
<td>Header File</td>
<td>Page</td>
<td>Description</td>
</tr>
<tr>
<td>----------</td>
<td>-------------</td>
<td>------</td>
<td>-------------</td>
</tr>
<tr>
<td>fwide()</td>
<td>stdio.h</td>
<td>“fwide() — Determine Stream Orientation” on page 162</td>
<td>Determines stream orientation.</td>
</tr>
<tr>
<td>setbuf()</td>
<td>stdio.h</td>
<td>“setbuf() — Control Buffering” on page 364</td>
<td>Allows control of buffering.</td>
</tr>
<tr>
<td>setvbuf()</td>
<td>stdio.h</td>
<td>“setvbuf() — Control Buffering” on page 372</td>
<td>Controls buffering and buffer size for a specified stream.</td>
</tr>
<tr>
<td>wfopen()</td>
<td>stdio.h</td>
<td>“wfopen() — Open Files” on page 532</td>
<td>Opens a specified stream, accepting file name and mode as wide characters.</td>
</tr>
</tbody>
</table>

**File Operations**

<table>
<thead>
<tr>
<th>Function</th>
<th>Header File</th>
<th>Page</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>fileno()</td>
<td>stdio.h</td>
<td>“fileno() — Determine File Handle” on page 131</td>
<td>Determines the file handle.</td>
</tr>
<tr>
<td>remove()</td>
<td>stdio.h</td>
<td>“remove() — Delete File” on page 301</td>
<td>Deletes a specified file.</td>
</tr>
<tr>
<td>rename()</td>
<td>stdio.h</td>
<td>“rename() — Rename File” on page 302</td>
<td>Renames a specified file.</td>
</tr>
<tr>
<td>tmpfile()</td>
<td>stdio.h</td>
<td>“tmpfile() — Create Temporary File” on page 444</td>
<td>Creates a temporary file and returns a pointer to that file.</td>
</tr>
<tr>
<td>tmpnam()</td>
<td>stdio.h</td>
<td>“tmpnam() — Produce Temporary File Name” on page 445</td>
<td>Produces a temporary file name.</td>
</tr>
</tbody>
</table>

**Handling Argument Lists**

<table>
<thead>
<tr>
<th>Function</th>
<th>Header File</th>
<th>Page</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>va_arg()</td>
<td>stdarg.h</td>
<td>“va_arg() – va_copy() – va_end() – va_start() — Handle Variable Argument List” on page 455</td>
<td>Allows access to variable number of function arguments.</td>
</tr>
<tr>
<td>Function</td>
<td>Header File</td>
<td>Page</td>
<td>Description</td>
</tr>
<tr>
<td>------------------</td>
<td>-------------</td>
<td>----------------------------------------------------------------------</td>
<td>-------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>va_copy()</td>
<td>stdarg.h</td>
<td>“va_arg() – va_copy() – va_end() – va_start() – Handle Variable Argument List” on page 455</td>
<td>Allows access to variable number of function arguments.</td>
</tr>
<tr>
<td>va_end()</td>
<td>stdarg.h</td>
<td>“va_arg() – va_copy() – va_end() – va_start() – Handle Variable Argument List” on page 455</td>
<td>Allows access to variable number of function arguments.</td>
</tr>
<tr>
<td>va_start()</td>
<td>stdarg.h</td>
<td>“va_arg() – va_copy() – va_end() – va_start() – Handle Variable Argument List” on page 455</td>
<td>Allows access to variable number of function arguments.</td>
</tr>
</tbody>
</table>

**Pseudorandom Numbers**

<table>
<thead>
<tr>
<th>Function</th>
<th>Header File</th>
<th>Page</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>rand(), rand_r()</td>
<td>stdlib.h</td>
<td>“rand() – rand_r() – Generate Random Number” on page 284</td>
<td>Returns a pseudorandom integer. (rand_r() is the restartable version of rand().)</td>
</tr>
<tr>
<td>srand()</td>
<td>stdlib.h</td>
<td>“srand() — Set Seed for rand() Function” on page 381</td>
<td>Sets the starting point for pseudorandom numbers.</td>
</tr>
</tbody>
</table>

**Dynamic Memory Management**

<table>
<thead>
<tr>
<th>Function</th>
<th>Header File</th>
<th>Page</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>calloc()</td>
<td>stdlib.h</td>
<td>“calloc() — Reserve and Initialize Storage” on page 80</td>
<td>Reserves storage space for an array and initializes the values of all elements to 0.</td>
</tr>
<tr>
<td>_C_Quickpool_Debug()</td>
<td>stdlib.h</td>
<td>“_C_Quickpool_Debug() — Modify Quick Pool Memory Manager Characteristics” on page 91</td>
<td>Modifies Quick Pool memory manager characteristics.</td>
</tr>
<tr>
<td>Function</td>
<td>Header File</td>
<td>Page</td>
<td>Description</td>
</tr>
<tr>
<td>--------------------------</td>
<td>---------------</td>
<td>----------------------------------------------------------------------</td>
<td>----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>_C_Quickpool_Init()</td>
<td>stdlib.h</td>
<td>“_C_Quickpool_Init() — Initialize Quick Pool Memory Manager” on page 93</td>
<td>Initializes the use of the Quick Pool memory manager algorithm.</td>
</tr>
<tr>
<td>_C_Quickpool_Report()</td>
<td>stdlib.h</td>
<td>“_C_Quickpool_Report() — Generate Quick Pool Memory Manager Report” on page 95</td>
<td>Generates a spooled file that contains a snapshot of the memory used by the Quick Pool memory manager algorithm in the current activation group.</td>
</tr>
<tr>
<td>_C_TS_malloc_debug()</td>
<td>mallocinfo.h</td>
<td>“_C_TS_malloc_debug() — Determine amount of teraspace memory used (with optional dumps and verification)” on page 103</td>
<td>Returns the same information as _C_TS_malloc_info, plus produces a spool file of detailed information about the memory structure used by malloc functions when compiled with teraspace.</td>
</tr>
<tr>
<td>_C_TS_malloc_info()</td>
<td>mallocinfo.h</td>
<td>“_C_TS_malloc_info() — Determine amount of teraspace memory used” on page 105</td>
<td>Returns the current memory usage information.</td>
</tr>
<tr>
<td>free()</td>
<td>stdlib.h</td>
<td>“free() — Release Storage Blocks” on page 151</td>
<td>Frees storage blocks.</td>
</tr>
<tr>
<td>malloc()</td>
<td>stdlib.h</td>
<td>“malloc() — Reserve Storage Block” on page 217</td>
<td>Reserves storage blocks.</td>
</tr>
<tr>
<td>realloc()</td>
<td>stdlib.h</td>
<td>“realloc() — Change Reserved Storage Block Size” on page 291</td>
<td>Changes storage size allocated for an object.</td>
</tr>
</tbody>
</table>

**Memory Objects**

<table>
<thead>
<tr>
<th>Function</th>
<th>Header File</th>
<th>Page</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>memchr()</td>
<td>string.h</td>
<td>“memchr() — Search Buffer” on page 234</td>
<td>Searches a buffer for the first occurrence of a given character.</td>
</tr>
<tr>
<td>memcmp()</td>
<td>string.h</td>
<td>“memcmp() — Compare Buffers” on page 235</td>
<td>Compares two buffers.</td>
</tr>
<tr>
<td>memcpy()</td>
<td>string.h</td>
<td>“memcpy() — Copy Bytes” on page 236</td>
<td>Copies a buffer.</td>
</tr>
<tr>
<td>Function</td>
<td>Header File</td>
<td>Page</td>
<td>Description</td>
</tr>
<tr>
<td>-------------</td>
<td>-------------</td>
<td>----------------------------------------------------------------------</td>
<td>----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>memicmp()</td>
<td>string.h</td>
<td>“memicmp() — Compare Bytes” on page 237</td>
<td>Compare two buffers without regard to case.</td>
</tr>
<tr>
<td>memmove()</td>
<td>string.h</td>
<td>“memmove() — Copy Bytes” on page 239</td>
<td>Moves a buffer.</td>
</tr>
<tr>
<td>memset()</td>
<td>string.h</td>
<td>“memset() — Set Bytes to Value” on page 240</td>
<td>Sets a buffer to a given value.</td>
</tr>
<tr>
<td>wmemchr()</td>
<td>wchar.h</td>
<td>“wmemchr() — Locate Wide Character in Wide-Character Buffer” on page 532</td>
<td>Locates a wide character in a wide-character buffer.</td>
</tr>
<tr>
<td>wmemcmp()</td>
<td>wchar.h</td>
<td>“wmemcmp() — Compare Wide-Character Buffers” on page 533</td>
<td>Compares two wide-character buffers.</td>
</tr>
<tr>
<td>wmemcpy()</td>
<td>wchar.h</td>
<td>“wmemcpy() — Copy Wide-Character Buffer” on page 535</td>
<td>Copies a wide-character buffer.</td>
</tr>
<tr>
<td>wmemmove()</td>
<td>wchar.h</td>
<td>“wmemmove() — Copy Wide-Character Buffer” on page 536</td>
<td>Moves a wide-character buffer.</td>
</tr>
<tr>
<td>wmemset()</td>
<td>wchar.h</td>
<td>“wmemset() — Set Wide Character Buffer to a Value” on page 537</td>
<td>Sets a wide-character buffer to a given value.</td>
</tr>
</tbody>
</table>

**Environment Interaction**

<table>
<thead>
<tr>
<th>Function</th>
<th>Header File</th>
<th>Page</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>abort()</td>
<td>stdlib.h</td>
<td>“abort() — Stop a Program” on page 60</td>
<td>Ends a program abnormally.</td>
</tr>
<tr>
<td>_C_Get_Ssn_Handle()</td>
<td>stdio.h</td>
<td>“_C_Get_Ssn_Handle() — Handle to C Session” on page 79</td>
<td>Returns a handle to the C session for use with DSM APIs.</td>
</tr>
<tr>
<td>Function</td>
<td>Header File</td>
<td>Page</td>
<td>Description</td>
</tr>
<tr>
<td>-------------------</td>
<td>-------------</td>
<td>------------------------------------------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>exit()</td>
<td>stdlib.h</td>
<td>“exit() — End Program” on page 113</td>
<td>Ends the program normally if called in the initial thread.</td>
</tr>
<tr>
<td>getenv()</td>
<td>stdlib.h</td>
<td>“getenv() — Search for Environment Variables” on page 176</td>
<td>Searches environment variables for a specified variable.</td>
</tr>
<tr>
<td>localeconv()</td>
<td>locale.h</td>
<td>“localeconv() — Retrieve Information from the Environment” on page 202</td>
<td>Formats numeric quantities in struct lconv according to the current locale.</td>
</tr>
<tr>
<td>longjmp()</td>
<td>setjmp.h</td>
<td>“longjmp() — Restore Stack Environment” on page 215</td>
<td>Restores a stack environment.</td>
</tr>
<tr>
<td>nl_langinfo()</td>
<td>langinfo.h</td>
<td>“nl_langinfo() — Retrieve Locale Information” on page 246</td>
<td>Retrieves information from the current locale.</td>
</tr>
<tr>
<td>putenv()</td>
<td>stdlib.h</td>
<td>“putenv() — Change/Add Environment Variables” on page 264</td>
<td>Sets the value of an environment variable by altering an existing variable or creating a new one.</td>
</tr>
<tr>
<td>setjmp()</td>
<td>setjmp.h</td>
<td>“setjmp() — Preserve Environment” on page 365</td>
<td>Saves a stack environment.</td>
</tr>
<tr>
<td>setlocale()</td>
<td>locale.h</td>
<td>“setlocale() — Set Locale” on page 366</td>
<td>Changes or queries locale.</td>
</tr>
<tr>
<td>system()</td>
<td>stdlib.h</td>
<td>“system() — Execute a Command” on page 438</td>
<td>Passes a string to the operating system's command interpreter.</td>
</tr>
<tr>
<td>wcslocaleconv()</td>
<td>locale.h</td>
<td>“wcslocaleconv() — Retrieve Wide Locale Information” on page 495</td>
<td>Formats numeric quantities in struct wcslconv according to the current locale.</td>
</tr>
</tbody>
</table>

### String Operations

<table>
<thead>
<tr>
<th>Function</th>
<th>Header File</th>
<th>Page</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>strcasecmp()</td>
<td>strings.h</td>
<td>“strcasecmp() — Compare Strings without Case Sensitivity” on page 384</td>
<td>Compares strings without case sensitivity.</td>
</tr>
<tr>
<td>strcat()</td>
<td>string.h</td>
<td>“strcat() — Concatenate Strings” on page 385</td>
<td>Concatenates two strings.</td>
</tr>
<tr>
<td>Function</td>
<td>Header File</td>
<td>Page</td>
<td>Description</td>
</tr>
<tr>
<td>------------</td>
<td>-------------</td>
<td>------------------------------------------------</td>
<td>--------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>strchr()</td>
<td>string.h</td>
<td>“strchr() — Search for Character” on page 386</td>
<td>Locates the first occurrence of a specified character in a string.</td>
</tr>
<tr>
<td>strcmp()</td>
<td>string.h</td>
<td>“strcmp() — Compare Strings” on page 388</td>
<td>Compares the value of two strings.</td>
</tr>
<tr>
<td>strcmpi()</td>
<td>string.h</td>
<td>“strcmpi() — Compare Strings Without Case Sensitivity” on page 389</td>
<td>Compares the value of two strings without regard to case.</td>
</tr>
<tr>
<td>strcoll()</td>
<td>string.h</td>
<td>“strcoll() — Compare Strings” on page 391</td>
<td>Compares the locale-defined value of two strings.</td>
</tr>
<tr>
<td>strcpy()</td>
<td>string.h</td>
<td>“strcpy() — Copy Strings” on page 392</td>
<td>Copies one string into another.</td>
</tr>
<tr>
<td>strcspn()</td>
<td>string.h</td>
<td>“strcspn() — Find Offset of First Character Match” on page 393</td>
<td>Finds the length of the first substring in a string of characters not in a second string.</td>
</tr>
<tr>
<td>strdup()</td>
<td>string.h</td>
<td>“strdup() — Duplicate String” on page 395</td>
<td>Duplicates a string.</td>
</tr>
<tr>
<td>strfmon()</td>
<td>string.h</td>
<td>“strfmon() — Convert Monetary Value to String” on page 396</td>
<td>Converts monetary value to string.</td>
</tr>
<tr>
<td>strftime()</td>
<td>time.h</td>
<td>“strftime() — Convert Date/Time to String” on page 399</td>
<td>Converts date and time to a formatted string.</td>
</tr>
<tr>
<td>stricmp()</td>
<td>string.h</td>
<td>“stricmp() — Compare Strings without Case Sensitivity” on page 403</td>
<td>Compares the value of two strings without regard to case.</td>
</tr>
<tr>
<td>strlen()</td>
<td>string.h</td>
<td>“strlen() — Determine String Length” on page 404</td>
<td>Calculates the length of a string.</td>
</tr>
<tr>
<td>Function</td>
<td>Header File</td>
<td>Page</td>
<td>Description</td>
</tr>
<tr>
<td>---------------</td>
<td>-------------</td>
<td>----------------------------------------------------------------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>strncasecmp()</td>
<td>strings.h</td>
<td>“strncasecmp() — Compare Strings without Case Sensitivity” on page 405</td>
<td>Compares strings without case sensitivity.</td>
</tr>
<tr>
<td>strncat()</td>
<td>string.h</td>
<td>“strncat() — Concatenate Strings” on page 406</td>
<td>Adds a specified length of one string to another string.</td>
</tr>
<tr>
<td>strncmp()</td>
<td>string.h</td>
<td>“strncmp() — Compare Strings” on page 408</td>
<td>Compares two strings up to a specified length.</td>
</tr>
<tr>
<td>strncpyy()</td>
<td>string.h</td>
<td>“strncpy() — Copy Strings” on page 409</td>
<td>Copies a specified length of one string into another.</td>
</tr>
<tr>
<td>strnicmp()</td>
<td>string.h</td>
<td>“strnicmp() — Compare Substrings Without Case Sensitivity” on page 411</td>
<td>Compares the value of two substrings without regard to case.</td>
</tr>
<tr>
<td>strnset()</td>
<td>string.h</td>
<td>“strnset() – strset() — Set Characters in String” on page 412</td>
<td>Sets character in a string.</td>
</tr>
<tr>
<td>strpbrk()</td>
<td>string.h</td>
<td>“strpbrk() — Find Characters in String” on page 413</td>
<td>Locates specified characters in a string.</td>
</tr>
<tr>
<td>strptime()</td>
<td>time.h</td>
<td>“strptime() — Convert String to Date/Time” on page 414</td>
<td>Converts string to formatted time.</td>
</tr>
<tr>
<td>strrchr()</td>
<td>string.h</td>
<td>“strrchr() — Locate Last Occurrence of Character in String” on page 418</td>
<td>Locates the last occurrence of a character within a string.</td>
</tr>
<tr>
<td>strspn()</td>
<td>string.h</td>
<td>“strspn() — Find Offset of First Non-matching Character” on page 419</td>
<td>Locates the first character in a string that is not part of specified set of characters.</td>
</tr>
<tr>
<td>Function</td>
<td>Header File</td>
<td>Page</td>
<td>Description</td>
</tr>
<tr>
<td>--------------</td>
<td>-------------</td>
<td>------------------------------------------------</td>
<td>---------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>strstr()</td>
<td>string.h</td>
<td>“strstr() — Locate Substring” on page 421</td>
<td>Locates the first occurrence of a string in another string.</td>
</tr>
<tr>
<td>strtok()</td>
<td>string.h</td>
<td>“strtok() — Tokenize String” on page 428</td>
<td>Locates a specified token in a string.</td>
</tr>
<tr>
<td>strtok_r()</td>
<td>string.h</td>
<td>“strtok_r() — Tokenize String (Restartable)” on page 429</td>
<td>Locates a specified token in a string. (Restartable version of strtok()).</td>
</tr>
<tr>
<td>strxfrm()</td>
<td>string.h</td>
<td>“strxfrm() — Transform String” on page 434</td>
<td>Transforms strings according to locale.</td>
</tr>
<tr>
<td>wcsftime()</td>
<td>wchar.h</td>
<td>“wcsftime() — Convert to Formatted Date and Time” on page 490</td>
<td>Converts to formatted date and time.</td>
</tr>
<tr>
<td>wcsptime()</td>
<td>wchar.h</td>
<td>“wcsptime() — Convert Wide Character String to Date/Time” on page 502</td>
<td>Converts string to formatted time.</td>
</tr>
<tr>
<td>wcsstr()</td>
<td>wchar.h</td>
<td>“wcsstr() — Locate Wide-Character Substring” on page 509</td>
<td>Locates a wide-character substring.</td>
</tr>
<tr>
<td>wcstok()</td>
<td>wchar.h</td>
<td>“wcstok() — Tokenize Wide-Character String” on page 514</td>
<td>Tokens a wide-character string.</td>
</tr>
</tbody>
</table>

### Character Testing

<table>
<thead>
<tr>
<th>Function</th>
<th>Header File</th>
<th>Page</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>isalnum()</td>
<td>ctype.h</td>
<td>“isalnum() — isxdigit() — Test Integer Value” on page 192</td>
<td>Tests for alphanumeric characters.</td>
</tr>
<tr>
<td>isalpha()</td>
<td>ctype.h</td>
<td>“isalnum() — isxdigit() — Test Integer Value” on page 192</td>
<td>Tests for alphabetic characters.</td>
</tr>
</tbody>
</table>

Library Functions 51
<table>
<thead>
<tr>
<th>Function</th>
<th>Header File</th>
<th>Page</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>isascii()</td>
<td>ctype.h</td>
<td>“isascii() — Test for Character Representable as ASCII Value” on page 193</td>
<td>Tests for ASCII values.</td>
</tr>
<tr>
<td>isblank()</td>
<td>ctype.h</td>
<td>“isalnum() – isxdigit() — Test Integer Value” on page 192</td>
<td>Tests for blank or tab characters.</td>
</tr>
<tr>
<td>iscndtrl()</td>
<td>ctype.h</td>
<td>“isalnum() – isxdigit() — Test Integer Value” on page 192</td>
<td>Tests for control characters.</td>
</tr>
<tr>
<td>isdigit()</td>
<td>ctype.h</td>
<td>“isalnum() – isxdigit() — Test Integer Value” on page 192</td>
<td>Tests for decimal digits.</td>
</tr>
<tr>
<td>isgraph()</td>
<td>ctype.h</td>
<td>“isalnum() – isxdigit() — Test Integer Value” on page 192</td>
<td>Tests for printable characters excluding the space.</td>
</tr>
<tr>
<td>isprint()</td>
<td>ctype.h</td>
<td>“isalnum() – isxdigit() — Test Integer Value” on page 192</td>
<td>Tests for printable characters including the space.</td>
</tr>
<tr>
<td>ispunct()</td>
<td>ctype.h</td>
<td>“isalnum() – isxdigit() — Test Integer Value” on page 192</td>
<td>Tests for punctuation characters as defined in the locale.</td>
</tr>
<tr>
<td>isspace()</td>
<td>ctype.h</td>
<td>“isalnum() – isxdigit() — Test Integer Value” on page 192</td>
<td>Tests for white-space characters.</td>
</tr>
<tr>
<td>isxdigit()</td>
<td>ctype.h</td>
<td>“isalnum() – isxdigit() — Test Integer Value” on page 192</td>
<td>Tests for wide hexadecimal digits 0 through 9, a through f, or A through F.</td>
</tr>
</tbody>
</table>
## Multibyte Character Testing

<table>
<thead>
<tr>
<th>Function</th>
<th>Header File</th>
<th>Page</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>iswalnum()</td>
<td>wctype.h</td>
<td>“iswalnum() – iswxdigit() — Test Wide Integer Value” on page 194</td>
<td>Tests for wide alphanumeric characters.</td>
</tr>
<tr>
<td>iswalpha()</td>
<td>wctype.h</td>
<td>“iswalnum() – iswxdigit() — Test Wide Integer Value” on page 194</td>
<td>Tests for wide alphabetic characters.</td>
</tr>
<tr>
<td>iswblank()</td>
<td>wctype.h</td>
<td>“iswalnum() – iswxdigit() — Test Wide Integer Value” on page 194</td>
<td>Tests for wide blank or tab characters.</td>
</tr>
<tr>
<td>iswcntrl()</td>
<td>wctype.h</td>
<td>“iswalnum() – iswxdigit() — Test Wide Integer Value” on page 194</td>
<td>Tests for wide control characters.</td>
</tr>
<tr>
<td>iswctype()</td>
<td>wctype.h</td>
<td>“iswctype() — Test for Character Property” on page 196</td>
<td>Tests for character property.</td>
</tr>
<tr>
<td>iswdigit()</td>
<td>wctype.h</td>
<td>“iswalnum() – iswxdigit() — Test Wide Integer Value” on page 194</td>
<td>Tests for wide decimal digits.</td>
</tr>
<tr>
<td>iswgraph()</td>
<td>wctype.h</td>
<td>“iswalnum() – iswxdigit() — Test Wide Integer Value” on page 194</td>
<td>Tests for wide printing characters excluding the space.</td>
</tr>
<tr>
<td>iswlower()</td>
<td>wctype.h</td>
<td>“iswalnum() – iswxdigit() — Test Wide Integer Value” on page 194</td>
<td>Tests for wide lowercase letters.</td>
</tr>
<tr>
<td>iswprint()</td>
<td>wctype.h</td>
<td>“iswalnum() – iswxdigit() — Test Wide Integer Value” on page 194</td>
<td>Tests for wide printing characters.</td>
</tr>
<tr>
<td>Function</td>
<td>Header File</td>
<td>Page</td>
<td>Description</td>
</tr>
<tr>
<td>--------------</td>
<td>-------------</td>
<td>-----------------------------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>iswpunct()</td>
<td>wctype.h</td>
<td>“iswalnum() – iswxdigit() — Test Wide Integer Value” on page 194</td>
<td>Tests for wide punctuation characters as defined in the locale.</td>
</tr>
<tr>
<td>iswupper()</td>
<td>wctype.h</td>
<td>“iswalnum() – iswxdigit() — Test Wide Integer Value” on page 194</td>
<td>Tests for wide uppercase letters.</td>
</tr>
<tr>
<td>iswxdigit()</td>
<td>wctype.h</td>
<td>“iswalnum() – iswxdigit() — Test Wide Integer Value” on page 194</td>
<td>Tests for wide hexadecimal digits 0 through 9, a through f, or A through F.</td>
</tr>
</tbody>
</table>

### Character Case Mapping

<table>
<thead>
<tr>
<th>Function</th>
<th>Header File</th>
<th>Page</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>tolower()</td>
<td>ctype.h</td>
<td>“tolower() – toupper() – Convert Character Case” on page 447</td>
<td>Converts a character to lowercase.</td>
</tr>
<tr>
<td>toupper()</td>
<td>ctype.h</td>
<td>“tolower() – toupper() – Convert Character Case” on page 447</td>
<td>Converts a character to uppercase.</td>
</tr>
<tr>
<td>towlower()</td>
<td>ctype.h</td>
<td>“tolower() – toupper() – Convert Wide Character Case” on page 449</td>
<td>Converts a wide character to lowercase.</td>
</tr>
<tr>
<td>towupper()</td>
<td>ctype.h</td>
<td>“tolower() – toupper() – Convert Wide Character Case” on page 449</td>
<td>Converts a wide character to uppercase.</td>
</tr>
</tbody>
</table>
## Multibyte Character Manipulation

<table>
<thead>
<tr>
<th>Function</th>
<th>Header File</th>
<th>Page</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>btowc()</code></td>
<td>stdio.h wchar.h</td>
<td>“btowc() — Convert Single Byte to Wide Character” on page 78</td>
<td>Converts a single byte to a wide character.</td>
</tr>
<tr>
<td><code>mblen()</code></td>
<td>stdlib.h</td>
<td>“mblen() — Determine Length of a Multibyte Character” on page 219</td>
<td>Determines the length of a multibyte character.</td>
</tr>
<tr>
<td><code>mbrlen()</code></td>
<td>stdlib.h</td>
<td>“mbrlen() — Determine Length of a Multibyte Character (Restartable)” on page 221</td>
<td>Determines the length of a multibyte character. (Restartable version of <code>mblen()</code>)</td>
</tr>
<tr>
<td><code>mbtowc()</code></td>
<td>stdlib.h</td>
<td>“mbtowc() — Convert a Multibyte Character to a Wide Character (Restartable)” on page 223</td>
<td>Converts a multibyte character to a wide character. (Restartable version of <code>mbtowc()</code>)</td>
</tr>
<tr>
<td><code>mbsinit()</code></td>
<td>stdlib.h</td>
<td>“mbsinit() — Test State Object for Initial State” on page 226</td>
<td>Tests state object for initial state.</td>
</tr>
<tr>
<td><code>mbsrtowcs()</code></td>
<td>stdlib.h</td>
<td>“mbsrtowcs() — Convert a Multibyte String to a Wide Character String (Restartable)” on page 227</td>
<td>Converts a multibyte string to a wide character string. (Restartable version of <code>mbstowcs()</code>)</td>
</tr>
<tr>
<td><code>mbstowcs()</code></td>
<td>stdlib.h</td>
<td>“mbstowcs() — Convert a Multibyte String to a Wide Character String” on page 229</td>
<td>Converts a multibyte string to a wide character string.</td>
</tr>
<tr>
<td>Function</td>
<td>Header File</td>
<td>Page</td>
<td>Description</td>
</tr>
<tr>
<td>------------</td>
<td>-------------</td>
<td>----------------------------------------------------------------------</td>
<td>----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>mbtowc()</td>
<td>stdlib.h</td>
<td>“mbtowc() — Convert Multibyte Character to a Wide Character” on page 233</td>
<td>Converts multibyte characters to a wide character.</td>
</tr>
<tr>
<td>towctrans()</td>
<td>wctype.h</td>
<td>“towctrans() — Translate Wide Character” on page 448</td>
<td>Translates wide character.</td>
</tr>
<tr>
<td>wcrtomb()</td>
<td>stdlib.h</td>
<td>“wcrtomb() — Convert a Wide Character to a Multibyte Character (Restartable)” on page 478</td>
<td>Converts a wide character to a multibyte character. (Restartable version of wcrtomb()).</td>
</tr>
<tr>
<td>wcscat()</td>
<td>wchar.h</td>
<td>“wcscat() — Concatenate Wide-Character Strings” on page 483</td>
<td>Concatenates wide character strings.</td>
</tr>
<tr>
<td>wcschr()</td>
<td>wchar.h</td>
<td>“wcschr() — Search for Wide Character” on page 484</td>
<td>Searches a wide character string for a wide character.</td>
</tr>
<tr>
<td>wcscmp()</td>
<td>wchar.h</td>
<td>“wcscmp() — Compare Wide-Character Strings” on page 485</td>
<td>Compares two wide character strings.</td>
</tr>
<tr>
<td>wcscoll()</td>
<td>wchar.h</td>
<td>“wcscoll() — Language Collation String Comparison” on page 487</td>
<td>Compares the locale-defined value of two wide-character strings.</td>
</tr>
<tr>
<td>wcscpy()</td>
<td>wchar.h</td>
<td>“wcscpy() — Copy Wide-Character Strings” on page 488</td>
<td>Copies a wide character string.</td>
</tr>
<tr>
<td>wcscspn()</td>
<td>wchar.h</td>
<td>“wcscspn() — Find Offset of First Wide-Character Match” on page 489</td>
<td>Searches a wide character string for characters.</td>
</tr>
<tr>
<td>Function</td>
<td>Header File</td>
<td>Page</td>
<td>Description</td>
</tr>
<tr>
<td>------------------</td>
<td>-------------</td>
<td>----------------------------------------------------------------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>_wcsicmp()</td>
<td>wchar.h</td>
<td>&quot;_wcsicmp() — Compare Wide Character Strings without Case Sensitivity&quot; on page 492</td>
<td>Compares two wide character strings without regard to case.</td>
</tr>
<tr>
<td>wcslen()</td>
<td>wchar.h</td>
<td>&quot;wcslen() — Calculate Length of Wide-Character String&quot; on page 494</td>
<td>Finds length of a wide character string.</td>
</tr>
<tr>
<td>wcsncat()</td>
<td>wchar.h</td>
<td>&quot;wcsncat() — Concatenate Wide-Character Strings&quot; on page 496</td>
<td>Concatenates a wide character string segment.</td>
</tr>
<tr>
<td>wcsncmp()</td>
<td>wchar.h</td>
<td>&quot;wcsncmp() — Compare Wide-Character Strings&quot; on page 497</td>
<td>Compares wide character string segments.</td>
</tr>
<tr>
<td>wcsncpy()</td>
<td>wchar.h</td>
<td>&quot;wcsncpy() — Copy Wide-Character Strings&quot; on page 499</td>
<td>Copies wide character string segments.</td>
</tr>
<tr>
<td>__wcsnicmp()</td>
<td>wchar.h</td>
<td>&quot;__wcsnicmp() — Compare Wide Character Strings without Case Sensitivity&quot; on page 500</td>
<td>Compares two wide character substrings without regard to case.</td>
</tr>
<tr>
<td>wcspbbrk()</td>
<td>wchar.h</td>
<td>&quot;wcspbbrk() — Locate Wide Characters in String&quot; on page 501</td>
<td>Locates wide characters in string.</td>
</tr>
<tr>
<td>wcsrchr()</td>
<td>wchar.h</td>
<td>&quot;wcsrchr() — Locate Last Occurrence of Wide Character in String&quot; on page 504</td>
<td>Locates wide character in string.</td>
</tr>
<tr>
<td>Function</td>
<td>Header File</td>
<td>Page</td>
<td>Description</td>
</tr>
<tr>
<td>--------------</td>
<td>-------------</td>
<td>----------------------------------------------------------------------</td>
<td>-------------</td>
</tr>
<tr>
<td>wcsrtombs()</td>
<td>stdlib.h</td>
<td>“wcsrtombs() — Convert Wide Character String to Multibyte String (Restartable)” on page 505</td>
<td>Converts a wide character string to a multibyte character string. (Restartable version of wcstombs()).</td>
</tr>
<tr>
<td>wcsspn()</td>
<td>wchar.h</td>
<td>“wcsspn() — Find Offset of First Non-matching Wide Character” on page 507</td>
<td>Finds offset of first nonmatching wide character.</td>
</tr>
<tr>
<td>wcstombs()</td>
<td>stdlib.h</td>
<td>“wcstombs() — Convert Wide-Character String to Multibyte String” on page 517</td>
<td>Converts a wide character string to a multibyte character string.</td>
</tr>
<tr>
<td>wcswcs()</td>
<td>wchar.h</td>
<td>“wcswcs() — Locate Wide-Character Substring” on page 522</td>
<td>Locates a wide character string in another wide character string.</td>
</tr>
<tr>
<td>wcswidth()</td>
<td>wchar.h</td>
<td>“wcswidth() — Determine the Display Width of a Wide Character String” on page 523</td>
<td>Determines the display width of a wide character string.</td>
</tr>
<tr>
<td>wcsxfrm()</td>
<td>wchar.h</td>
<td>“wcsxfrm() — Transform a Wide-Character String” on page 524</td>
<td>Transforms wide-character strings according to locale.</td>
</tr>
<tr>
<td>wctob()</td>
<td>stdlib.h</td>
<td>“wctob() — Convert Wide Character to Byte” on page 525</td>
<td>Converts a wide character to a single byte.</td>
</tr>
<tr>
<td>wctomb()</td>
<td>stdlib.h</td>
<td>“wctomb() — Convert Wide Character to Multibyte Character” on page 527</td>
<td>Converts a wide character to multibyte characters.</td>
</tr>
<tr>
<td>Function</td>
<td>Header File</td>
<td>Page</td>
<td>Description</td>
</tr>
<tr>
<td>------------</td>
<td>-------------</td>
<td>----------------------------------------------------------------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>wctrans()</td>
<td>wctype.h</td>
<td>“wctrans() — Get Handle for Character Mapping” on page 528</td>
<td>Gets a handle for character mapping.</td>
</tr>
<tr>
<td>wctype()</td>
<td>wchar.h</td>
<td>“wctype() — Get Handle for Character Property Classification” on page 529</td>
<td>Obtains a handle for character property classification.</td>
</tr>
<tr>
<td>wcwidth()</td>
<td>wchar.h</td>
<td>“wcwidth() — Determine the Display Width of a Wide Character” on page 531</td>
<td>Determines the display width of a wide character.</td>
</tr>
</tbody>
</table>

**Data Areas**

<table>
<thead>
<tr>
<th>Function</th>
<th>Header File</th>
<th>Page</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>QXXCHGDA()</td>
<td>xxdtaa.h</td>
<td>“QXXCHGDA() — Change Data Area” on page 274</td>
<td>Changes the data area.</td>
</tr>
<tr>
<td>QXXRTVDA()</td>
<td>xxdtaa.h</td>
<td>“QXXRTVDA() — Retrieve Data Area” on page 280</td>
<td>Retrieves a copy of the data area specified by dtaname.</td>
</tr>
</tbody>
</table>

**Message Catalogs**

<table>
<thead>
<tr>
<th>Function</th>
<th>Header File</th>
<th>Page</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>catclose()</td>
<td>nl_types.h</td>
<td>“catclose() — Close Message Catalog” on page 81</td>
<td>Closes a message catalog.</td>
</tr>
<tr>
<td>catgets()</td>
<td>nl_types.h</td>
<td>“catgets() — Retrieve a Message from a Message Catalog” on page 82</td>
<td>Reads a message from an opened message catalog.</td>
</tr>
<tr>
<td>catopen()</td>
<td>nl_types.h</td>
<td>“catopen() — Open Message Catalog” on page 84</td>
<td>Opens a message catalog.</td>
</tr>
</tbody>
</table>
## Regular Expression

<table>
<thead>
<tr>
<th>Function</th>
<th>Header File</th>
<th>Page</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>regcomp()</code></td>
<td>regex.h</td>
<td>“regcomp() — Compile Regular Expression” on page 294</td>
<td>Compiles a regular expression.</td>
</tr>
<tr>
<td><code>regerror()</code></td>
<td>regex.h</td>
<td>“regerror() — Return Error Message for Regular Expression” on page 296</td>
<td>Returns error message for regular expression.</td>
</tr>
<tr>
<td><code>regexec()</code></td>
<td>regex.h</td>
<td>“regexec() — Execute Compiled Regular Expression” on page 297</td>
<td>Executes a compiled regular expression.</td>
</tr>
<tr>
<td><code>regfree()</code></td>
<td>regex.h</td>
<td>“regfree() — Free Memory for Regular Expression” on page 300</td>
<td>Frees memory for regular expression.</td>
</tr>
</tbody>
</table>

## abort() — Stop a Program

### Format

```c
#include <stdlib.h>
void abort(void);
```

### Language Level

ANSI

### Threadsafe

Yes

### Description

The `abort()` function causes an abnormal end of the program and returns control to the host environment. Like the `exit()` function, the `abort()` function deletes buffers and closes open files before ending the program.

Calls to the `abort()` function raise the SIGABRT signal. The `abort()` function will not result in the ending of the program if SIGABRT is caught by a signal handler, and the signal handler does not return.

**Note:** When compiled with SYSIFCOPT(*ASYNCSIGNAL), the `abort()` function cannot be called in a signal handler.
Return Value
There is no return value.

Example
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.

```c
#include <stdio.h>
#include <stdlib.h>

int main(void)
{
    FILE *stream;
    if ((stream = fopen("mylib/myfile", "r")) == NULL)
    {
        perror("Could not open data file");
        abort();
    }
}
```

Related Information
- “exit() — End Program” on page 113
- “signal() — Handle Interrupt Signals” on page 374
- “<stdlib.h>” on page 15
- See the signal() API in the APIs topic in the Information Center.

abs() — Calculate Integer Absolute Value

Format
```c
#include <stdlib.h>
int abs(int n);
```

Language Level
ANSI

Threadsafe
Yes

Description
The abs() function returns the absolute value of an integer argument n.

Return Value
There is no error return value. The result is undefined when the absolute value of the argument cannot be represented as an integer. The value of the minimum allowable integer is defined by INT_MIN in the <limits.h> include file.

Example
This example calculates the absolute value of an integer x and assigns it to y.

```c
#include <stdlib.h>
#include <stdio.h>
```
int main(void)
{
    int x = -4, y;
    y = abs(x);
    printf("The absolute value of x is %d.\n", y);
    
    /* Output */
    The absolute value of x is 4.
    ******************************************
}

Related Information

- “fabs() — Calculate Floating-Point Absolute Value” on page 115
- “labs() – llabs() — Calculate Absolute Value of Long and Long Long Integer” on page 199
- “<limits.h>” on page 5
- “<stdlib.h>” on page 15

acos() — Calculate Arccosine

Format

#include <math.h>
double acos(double x);

Language Level

ANSI

Threadsafe

Yes

Description

The acos() function calculates the arccosine of x, expressed in radians, in the range 0 to π.

Return Value

The acos() function returns the arccosine of x. The value of x must be between -1 and 1 inclusive. If x is less than -1 or greater than 1, acos() sets errno to EDOM and returns 0.

Example

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: %lf too large for acos\n", x );
else if ( x < MIN )
    printf( "Error: %lf too small for acos\n", x );
else {
    y = acos( x );
    printf( "acos( %lf ) = %lf\n", x, y );
}

/******** Expected output if 0.4 is entered: **********
Enter x
acos( 0.400000 ) = 1.159279
*/

Related Information
• “asin() — Calculate Arcsine” on page 66
• “atan() – atan2() — Calculate Arctangent” on page 69
• “cos() — Calculate Cosine” on page 89
• “cosh() — Calculate Hyperbolic Cosine” on page 90
• “sin() — Calculate Sine” on page 376
• “sinh() — Calculate Hyperbolic Sine” on page 377
• “tan() — Calculate Tangent” on page 439
• “tanh() — Calculate Hyperbolic Tangent” on page 440
• “<math.h>” on page 6

asctime() — Convert Time to Character String

Format

#include <time.h>
char *asctime(const struct tm *time);

Language Level
ANSI

Threading Safe
No

Use asctime_r() instead.

Description
The asctime() function converts time, stored as a structure pointed to by time, to a character string. You can obtain the time value from a call to the gmtime(), gmtime64(), localtime(), or localtime64() function.

The string result that asctime() produces contains exactly 26 characters and has the format:

"%.3s %.3s%3d %.2d:%.2d:%.2d %d\n"

The following are examples of the string returned:

Sat Jul 16 02:03:55 1994\n
or
Sat Jul 16 2:03:55 1994\n
The \texttt{asctime()} function uses a 24-hour-clock format. The days are abbreviated to: Sun, Mon, Tue, Wed, Thu, Fri, and Sat. The months are abbreviated to: Jan, Feb, Mar, Apr, May, Jun, Jul, Aug, Sep, Oct, Nov, and Dec. All fields have constant width. Dates with only one digit are preceded either with a zero or a blank space. The new-line character (\texttt{\n}) and the null character (\texttt{\0}) occupy the last two positions of the string.

The time and date functions begin at 00:00:00 Universal Time, January 1, 1970.

\textbf{Return Value}

The \texttt{asctime()} function returns a pointer to the resulting character string. If the function is unsuccessful, it returns NULL.

\textbf{Note:} The \texttt{asctime()}, \texttt{ctime()} functions, and other time functions can use a common, statically allocated buffer to hold the return string. Each call to one of these functions might destroy the result of the previous call. The \texttt{asctime\_r()}, \texttt{ctime\_r()}, \texttt{gmtime\_r()}, and \texttt{localtime\_r()} functions do not use a common, statically allocated buffer to hold the return string. These functions can be used in place of the \texttt{asctime()}, \texttt{ctime()}, \texttt{gmtime()}, and \texttt{localtime()} functions if reentrancy is desired.

\textbf{Example}

This example polls the system clock and prints a message that gives the current time.

```c
#include <time.h>
#include <stdio.h>

int main(void)
{
    struct tm *newtime;
    time_t ltime;

    /* Get the time in seconds */
    time(&ltime);

    /* Convert it to the structure tm */
    newtime = localtime(&ltime);

    /* Print the local time as a string */
    printf("The current date and time are \%s",
            asctime(newtime));
}

//*************** Output should be similar to: ******************
The current date and time are Fri Sep 16 13:29:51 1994
/*

\textbf{Related Information}

\begin{itemize}
\item \texttt{asctime\_r()} — Convert Time to Character String (Restartable)" on page 65
\item \texttt{ctime()} — Convert Time to Character String" on page 96
\item \texttt{ctime64()} — Convert Time to Character String" on page 98
\item \texttt{ctime\_r()} — Convert Time to Character String (Restartable)" on page 101
\item \texttt{ctime\_r()} — Convert Time to Character String (Restartable)" on page 100
\item \texttt{gmtime()} — Convert Time" on page 183
\item \texttt{gmtime64()} — Convert Time" on page 185
\item \texttt{gmtime\_r()} — Convert Time (Restartable)" on page 189
\item \texttt{gmtime\_r()} — Convert Time (Restartable)" on page 187
\item \texttt{localtime()} — Convert Time" on page 207
\item \texttt{localtime64()} — Convert Time" on page 208
\item \texttt{localtime\_r()} — Convert Time (Restartable)" on page 211
\item \texttt{localtime\_r()} — Convert Time (Restartable)" on page 210
\item \texttt{mktime()} — Convert Local Time" on page 241
\end{itemize}

64 IBM i: ILE C/C++ Runtime Library Functions
asctime_r() — Convert Time to Character String (Restartable)

Format

```c
#include <time.h>
char *asctime_r(const struct tm *tm, char *buf);
```

Language Level

XPG4

Threading

Yes

Description

This function is the restartable version of the `asctime()` function. The `asctime_r()` function converts time, stored as a structure pointed to by `tm`, to a character string. You can obtain the `tm` value from a call to `gmtime_r()`, `gmtime64_r()`, `localtime_r()`, or `localtime64_r()`.

The string result that `asctime_r()` produces contains exactly 26 characters and has the format:

```
%.3s %.3s%3d %.2d:%.2d:%.2d %d
```

The following are examples of the string returned:

```
Sat Jul 16 02:03:55 1994
or
Sat Jul 16  2:03:55 1994
```

The `asctime_r()` function uses a 24-hour-clock format. The days are abbreviated to: Sun, Mon, Tue, Wed, Thu, Fri, and Sat. The months are abbreviated to: Jan, Feb, Mar, Apr, May, Jun, Jul, Aug, Sep, Oct, Nov, and Dec. All fields have constant width. Dates with only one digit are preceded either with a zero or a blank space. The new-line character (`\n`) and the null character (`\0`) occupy the last two positions of the string.

The time and date functions begin at 00:00:00 Universal Time, January 1, 1970.

Return Value

The `asctime_r()` function returns a pointer to the resulting character string. If the function is unsuccessful, it returns NULL.

Example

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;
    char  mybuf[50];
    /* Get the time in seconds */
    time(&ltime);
    /* Convert it to the structure tm */
    newtime = localtime_r(&ltime);
    /* Print the local time as a string */
    printf("The current date and time are %s",
        asctime_r(newtime, mybuf));
}

/****************  Output should be similar to  ******************
The current date and time are Fri Sep 16 132951 1994
*/

Related Information

• “asctime() — Convert Time to Character String” on page 63
• “ctime() — Convert Time to Character String” on page 96
• “ctime64() — Convert Time to Character String” on page 98
• “ctime64_r() — Convert Time to Character String (Restartable)” on page 101
• “ctime_r() — Convert Time to Character String (Restartable)” on page 100
• “gmtime() — Convert Time” on page 183
• “gmtime64() — Convert Time” on page 185
• “gmtime64_r() — Convert Time (Restartable)” on page 189
• “gmtime_r() — Convert Time (Restartable)” on page 187
• “localtime() — Convert Time” on page 207
• “localtime64() — Convert Time” on page 208
• “localtime64_r() — Convert Time (Restartable)” on page 211
• “localtime_r() — Convert Time (Restartable)” on page 210
• “mktime() — Convert Local Time” on page 241
• “mktime64() — Convert Local Time” on page 242
• “strftime() — Convert Date/Time to String” on page 399
• “time() — Determine Current Time” on page 441
• “printf() — Print Formatted Characters” on page 251
• “<time.h>” on page 16

asin() — Calculate Arcsine

Format

#include <math.h>
double asin(double x);

Language Level

ANSI
Threadsafe

Yes

Description

The `asin()` function calculates the arcsine of `x`, in the range \(-\pi/2\) to \(\pi/2\) radians.

Return Value

The `asin()` function returns the arcsine of `x`. The value of `x` must be between -1 and 1. If `x` is less than -1 or greater than 1, the `asin()` function sets `errno` to EDOM, and returns a value of 0.

Example

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`.

```c
#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: %lf too large for asin\n", x );
    else if ( x < MIN )
        printf( "Error: %lf too small for asin\n", x );
    else
    {
        y = asin( x );
        printf( "asin( %lf ) = %lf\n", x, y );
    }
}
/****************  Output should be similar to  *******************/
Enter x
asin( 0.200000 ) = 0.201358
*/
```

Related Information

- “acos() — Calculate Arccosine” on page 62
- “atan() — atan2() — Calculate Arctangent” on page 69
- “cos() — Calculate Cosine” on page 89
- “cosh() — Calculate Hyperbolic Cosine” on page 90
- “sin() — Calculate Sine” on page 376
- “sinh() — Calculate Hyperbolic Sine” on page 377
- “tan() — Calculate Tangent” on page 439
- “tanh() — Calculate Hyperbolic Tangent” on page 440
- “<math.h>” on page 6
**assert() — Verify Condition**

**Format**

```c
#include <assert.h>
void assert(int expression);
```

**Language Level**

ANSI

**Threadsafe**

No

**Description**

The `assert()` function prints a diagnostic message to `stderr` and aborts the program if `expression` is false (zero). The diagnostic message has one of the following formats, depending on the language level used during the compilation:

- `Assertion failed: expression, file filename, line line-number.`
- `Assertion failed: expression, file filename, line line-number, function function-name.`

The `assert()` function takes no action if the `expression` is true (nonzero).

Use the `assert()` function to identify program logic errors. Choose an `expression` that holds true only if the program is operating as you intend. After you have debugged the program, you can use the special no-debug identifier `NDEBUG` to remove the `assert()` calls from the program. If you define `NDEBUG` to any value with a `#define` directive, the C preprocessor expands all `assert()` calls to void expressions. If you use `NDEBUG`, you must define it before you include `<assert.h>` in the program.

**Return Value**

There is no return value.

**Note:** The `assert()` function is defined as a macro. Do not use the `#undef` directive with `assert()`.

**Example**

In this example, the `assert()` function tests `string` for a null string and an empty string, and verifies that `length` is positive before processing these arguments.

```c
#include <stdio.h>
#include <assert.h>

void analyze (char *, int);

int main(void)
{
    char *string = "ABC";
    int length = 3;
    analyze(string, length);
    printf("The string %s is not null or empty, "
           "and has length %d \n", string, 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 */
}
```
atan() – atan2() — Calculate Arctangent

Format

```c
#include <math.h>
double atan(double x);
double atan2(double y, double x);
```

Language Level

ANSI

Threadsafe

Yes

Description

The `atan()` and `atan2()` functions calculate the arctangent of `x` and `y/x`, respectively.

Return Value

The `atan()` function returns a value in the range -π/2 to π/2 radians. The `atan2()` function returns a value in the range -π to π radians. If both arguments of the `atan2()` function are zero, the function sets `errno` to EDOM, and returns a value of 0.

Example

This example calculates arctangents using the `atan()` and `atan2()` functions.

```c
#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( %lf ) = %lf
", c, a);
    printf("atan2( %lf, %lf ) = %lf/n", c, d, b);
}
```

Library Functions 69
Related Information

- “acos() — Calculate Arccosine” on page 62
- “asin() — Calculate Arcsine” on page 66
- “cos() — Calculate Cosine” on page 89
- “cosh() — Calculate Hyperbolic Cosine” on page 90
- “sin() — Calculate Sine” on page 376
- “sinh() — Calculate Hyperbolic Sine” on page 377
- “tan() — Calculate Tangent” on page 439
- “tanh() — Calculate Hyperbolic Tangent” on page 440
- “<math.h>” on page 6

**atexit() — Record Program Ending Function**

**Format**

```c
#include <stdlib.h>
int atexit(void (*func)(void));
```

**Language Level**

ANSI

**Threadsafe**

Yes

**Description**

The `atexit()` function records the function, pointed to by `func`, that the system calls at normal program end. For portability, you should use the `atexit()` function to register a maximum of 32 functions. The functions are processed in a last-in, first-out order. The `atexit()` function cannot be called from the OPM default activation group. Most functions can be used with the `atexit` function; however, if the exit function is used the `atexit` function will fail.

**Return Value**

The `atexit()` function returns 0 if it is successful, and nonzero if it fails.

**Example**

This example uses the `atexit()` function to call `goodbye()` at program end.

```c
#include <stdlib.h>
#include <stdio.h>

int main(void)
{
    void goodbye(void);
    int rc;
    rc = atexit(goodbye);
    if (rc != 0)
        perror("Error in atexit");
    exit(0);
}

void goodbye(void)
/* This function is called at normal program end */
```
The function goodbye was called at program end
}

/** Output should be similar to: ***/
The function goodbye was called at program end
*/

Related Information

- “exit() — End Program” on page 113
- “signal() — Handle Interrupt Signals” on page 374
- “<stdlib.h>” on page 15

### atof() — Convert Character String to Float

#### Format

```
#include <stdlib.h>
double atof(const char *string);
```

#### Language Level

ANSI

#### Threadsafe

Yes

#### Locale Sensitive

The behavior of this function might be affected by the LC_CTYPE and LC_NUMERIC categories of the current locale. For more information, see “Understanding CCSIDs and Locales” on page 565.

#### Description

The `atof()` function converts a character string to a double-precision floating-point value.

The input `string` is a sequence of characters that can be interpreted as a numeric value of the specified return type. The function stops reading the input string at the first character that it cannot recognize as part of a number. This character can be the null character that ends the string.

The `atof()` function expects a `string` in the following form:

- White space consists of the same characters for which the `isspace()` function is true, such as spaces and tabs. The `atof()` function ignores leading white-space characters.
- For the `atof()` function, `digits` is one or more decimal digits; if no digits appear before the decimal point, at least one digit must appear after the decimal point. The decimal digits can precede an exponent, introduced by the letter `e` or `E`. The exponent is a decimal integer, which might be signed.
The `atof()` function will not fail if a character other than a digit follows an E or if e is read in as an exponent. For example, 100elf will be converted to the floating-point value 100.0. The accuracy is up to 17 significant character digits.

**Return Value**

The `atof()` function returns a double value that is produced by interpreting the input characters as a number. The return value is 0 if the function cannot convert the input to a value of that type. In case of overflow, the function sets `errno` to ERANGE and returns the value -HUGE_VAL or +HUGE_VAL.

**Example**

This example shows how to convert numbers that are stored as strings to numeric values.

```c
#include <stdlib.h>
#include <stdio.h>

int main(void)
{
    double x;
    char *s;
    s = " -2309.12E-15";
    x = atof(s);  /* x = -2309.12E-15 */
    printf("x = %.4e\n",x);
}
```

/********************  Output should be similar to:  ***************

x = -2.3091e-12
*/

**Related Information**

- “`atoi()` — Convert Character String to Integer” on page 72
- “`atoll()` — Convert Character String to Long or Long Long Integer” on page 74
- “`strtol()` — Convert Character String to Long and Long Long Integer” on page 430
- “`strtod()` - `strtod()` - `strtof()` — Convert Character String to Double, Float, and Long Double” on page 422
- “`strtod32()` - `strtod64()` - `strtod128()` — Convert Character String to Decimal Floating-Point” on page 425
- “`<stdlib.h>`” on page 15

---

**atoi() — Convert Character String to Integer**

**Format**

```c
#include <stdlib.h>
int atoi(const char *string);
```

**Language Level**

ANSI

**Threadsafe**

Yes
Locale Sensitive

The behavior of this function might be affected by the LC_CTYPE category of the current locale. For more information, see “Understanding CCSIDs and Locales” on page 565.

Description

The atoi() function converts a character string to an integer value. The input string is a sequence of characters that can be interpreted as a numeric value of the specified return type. The function stops reading the input string at the first character that it cannot recognize as part of a number. This character can be the null character that ends the string.

The atoi() function does not recognize decimal points or exponents. The string argument for this function has the form:

```
whitespace + - digits
```

where whitespace consists of the same characters for which the isspace() function is true, such as spaces and tabs. The atoi() function ignores leading white-space characters. The value digits represents one or more decimal digits.

Return Value

The atoi() function returns an int value that is produced by interpreting the input characters as a number. The return value is 0 if the function cannot convert the input to a value of that type. The return value is undefined in the case of an overflow.

Example

This example shows how to convert numbers that are stored as strings to numeric values.

```c
#include <stdlib.h>
#include <stdio.h>

int main(void)
{
    int i;
    char *s;
    s = " -9885";
    i = atoi(s);     /* i = -9885 */
    printf("i = %d\n",i);
}
```

```
Output should be similar to: ***************
i = -9885
```

Related Information

- “atof() — Convert Character String to Float” on page 71
- “atol() – atoll() — Convert Character String to Long or Long Long Integer” on page 74
- “strtod() - strtof() - strtold() — Convert Character String to Double, Float, and Long Double” on page 422
- “strtod32() - strtod64() - strtod128() — Convert Character String to Decimal Floating-Point” on page 425
- “strtol() – strtoll() — Convert Character String to Long and Long Long Integer” on page 430
- “<stdlib.h>” on page 15
atol() – atoll() — Convert Character String to Long or Long Long Integer

**Format (atol())**

```c
#include <stdlib.h>
long int atol(const char *string);
```

**Format (atoll())**

```c
#include <stdlib.h>
long long int atoll(const char *string);
```

**Language Level**

ANSI

**Threadsafe**

Yes

**Locale Sensitive**

The behavior of these functions might be affected by the LC_CTYPE category of the current locale. For more information, see “Understanding CCSIDs and Locales” on page 565.

**Description**

The `atol()` function converts a character string to a long value. The `atoll()` function converts a character string to a long long value.

The input `string` is a sequence of characters that can be interpreted as a numeric value of the specified return type. The function stops reading the input string at the first character that it cannot recognize as part of a number. This character can be the null character that ends the string.

The `atol()` and `atoll()` functions do not recognize decimal points or exponents. The `string` argument for this function has the form:

```
whitespace + digits
```

where `whitespace` consists of the same characters for which the `isspace()` function is true, such as spaces and tabs. The `atol()` and `atoll()` functions ignore leading white-space characters. The value `digits` represents one or more decimal digits.

**Return Value**

The `atol()` and `atoll()` functions return a long or a long long value that is produced by interpreting the input characters as a number. The return value is 0L if the function cannot convert the input to a value of that type. The return value is undefined in case of overflow.

**Example**

This example shows how to convert numbers that are stored as strings to numeric values.

```c
#include <stdlib.h>
#include <stdio.h>

int main(void)
{

```

74 IBM i: ILE C/C++ Runtime Library Functions
```c
long l;
char *s;
s = "98854 dollars";
l = atol(s); /* l = 98854 */
printf("l = %.ld\n", l);
}
/********************
Output should be similar to: ********************
l = 98854
*/
```

### Related Information
- “atof() — Convert Character String to Float” on page 71
- “atoi() — Convert Character String to Integer” on page 72
- “strtod() - strtodf() - strtold() — Convert Character String to Double, Float, and Long Double” on page 422
- “strtod32() - strtod64() - strtod128() — Convert Character String to Decimal Floating-Point” on page 425
- “strtol() – strtoll() — Convert Character String to Long and Long Long Integer” on page 430
- “<stdlib.h>” on page 15

### Bessel Functions

**Format**

```c
#include <math.h>
double j0(double x);
double j1(double x);
double jn(int n, double x);
double y0(double x);
double y1(double x);
double yn(int n, double x);
```

**Language Level**

ILE C Extension

**Threadsafe**

Yes

**Description**

Bessel functions solve certain types of differential equations. The \( j_0() \), \( j_1() \), and \( j_n() \) functions are Bessel functions of the first kind for orders 0, 1, and \( n \), respectively. The \( y_0() \), \( y_1() \), and \( y_n() \) 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, it will be a negative exponent.

**Return Value**

For \( j_0() \), \( j_1() \), \( y_0() \), or \( y_1() \), if the absolute value of \( x \) is too large, the function sets errno to ERANGE, and returns 0. For \( y_0() \), \( y_1() \), or \( y_n() \), if \( x \) is negative, the function sets errno to EDOM and returns the value -HUGE_VAL. For \( y_0 \), \( y_1 \), or \( y_n \), if \( x \) causes overflow, the function sets errno to ERANGE and returns the value -HUGE_VAL.
Example
This example computes $y$ to be the order 0 Bessel function of the first kind for $x$. It also computes $z$ to be the order 3 Bessel function of the second kind for $x$.

```c
#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("y = %lf\n", y);
    printf("z = %lf\n", z);
}
```

/* Output should be similar to: */
```
y = -0.366022
z = -0.087482
```

Related Information
- “erf() – erfc() — Calculate Error Functions” on page 112
- “gamma() — Gamma Function” on page 172
- “<math.h>” on page 6

bsearch() — Search Arrays

Format
```c
#include <stdlib.h>
void *bsearch(const void *key, const void *base,
              size_t num, size_t size,
              int (*compare)(const void *key, const void *element));
```

Language Level
ANSI

Threading
Yes

Description
The `bsearch()` function performs a binary search of an array of `num` elements, each of `size` bytes. The array must be sorted in ascending order by the function pointed to by `compare`. The `base` is a pointer to the base of the array to search, and `key` is the value being sought.

The `compare` argument is a pointer to a function you must supply that compares two items and returns a value specifying their relationship. The first item in the argument list of the `compare()` function is the pointer to the value of the item that is being searched for. The second item in the argument list of the `compare()` function is a pointer to the array `element` being compared with the `key`. The `compare()` function must compare the key value with the array element and then return one of the following values:
<table>
<thead>
<tr>
<th>Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>Less than 0</td>
<td><em>key</em> less than <em>element</em></td>
</tr>
<tr>
<td>0</td>
<td><em>key</em> identical to <em>element</em></td>
</tr>
<tr>
<td>Greater than 0</td>
<td><em>key</em> greater than <em>element</em></td>
</tr>
</tbody>
</table>

**Return Value**

The `bsearch()` function returns a pointer to *key* in the array to which *base* points. If two keys are equal, the element that *key* will point to is unspecified. If the `bsearch()` function cannot find the *key*, it returns NULL.

**Example**

This example performs a binary search on the argv array of pointers to the program parameters 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 `compare1()` and `compare2()` functions compare the values pointed to by `arg1` and `arg2` and return the result to the `bsearch()` function.

```c
#include <stdlib.h>
#include <stdio.h>
#include <string.h>

int compare1(const void *, const void *);  
int compare2(const void *, const void *);  

main(int argc, char *argv[])  
{  
    char **result;  
    char *key = "PATH";  
    int i;  
    argv++;  
    argc--;  

    qsort((char *)argv, argc, sizeof(char *), compare1);  
    result = (char**)bsearch(&key, (char *)argv, argc, sizeof(char *), compare2);  
    if (result != NULL)  
    {  
        printf("result =<"%s">\n",*result);  
    }  
    else printf("result is null\n");  
}  

/*This function compares the values pointed to by arg1 */  
/*and arg2 and returns the result to qsort. arg1 and */  
/*arg2 are both pointers to elements of the argv array. */  
int compare1(const void *arg1, const void *arg2)  
{  
    return (strcmp(*(char **)arg1, *(char **)arg2));  
}  

/*This function compares the values pointed to by arg1 */  
/*and arg2 and returns the result to bsearch */  
/*arg1 is a pointer to the key value, arg2 points to */  
/*the element of argv that is being compared to the key */  
/*value. */  
int compare2(const void *arg1, const void *arg2)  
{  
    return (strcmp(*(char **)arg1, *(char **)arg2));  
}  
```
btowc() — Convert Single Byte to Wide Character

Format

```c
#include <stdio.h>
#include <wchar.h>
wint_t btowc(int c);
```

Language Level

ANSI

Threadsafe

Yes

Locale Sensitive

The behavior of this function might be affected by the LC_CTYPE category of the current locale. The behavior might also be affected by the LC_UNI_CTYPE category of the current locale if LOCALETYPE(*LOCALEUCS2) or LOCALETYPE(*LOCALEUTF) is specified on the compilation command. This function is not available when LOCALETYPE(*CLD) is specified on the compilation command. For more information, see “Understanding CCSIDs and Locales” on page 565.

Wide Character Function

See “Wide Characters” on page 568 for more information.

Description

The `btowc()` function converts the single byte value `c` to the wide-character representation of `c`. If `c` does not constitute a valid (1-byte) multibyte character in the initial shift state, the `btowc()` function returns WEOF.

Return Value

The `btowc()` function returns WEOF if `c` has the value EOF, or if (unsigned char) `c` does not constitute a valid (1-byte) multibyte character in the initial shift state. Otherwise, it returns the wide-character representation of that character.

If a conversion error occurs, `errno` might be set to ECONVERT.

Example

This example scans various types of data.

```c
#include <stdio.h>
#include <stdlib.h>
#include <wchar.h>
#include <local.h>

#define UPPER_LIMIT 0xFF

int main(void)
{
    int wc;
    int ch;
```
if (NULL == setlocale(LC_ALL, "/QSYS.LIB/EN_US.LOCALE")) {
    printf("Locale could not be loaded\n");
    exit(1);
}
for (ch = 0; ch <= UPPER_LIMIT; ++ch) {
    wc = btowc(ch);
    if (wc==WEOF) {
        printf("%#04x is not a one-byte multibyte character\n", ch);
    } else {
        printf("%#04x has wide character representation: %#06x\n", ch, wc);
    }
}
w = btowc(EOF);
if (wc==WEOF) {
    printf("The character is EOF.\n", ch);
} else {
    printf("EOF has wide character representation: %#06x\n", wc);
}
return 0;

/***********************************************************************
| If the locale is bound to SBCS, the output should be similar to:    |
| 0000 has wide character representation: 000000                   |
| 0x01 has wide character representation: 0x0001                    |
| ...                                                               |
| 0xfe has wide character representation: 0x00fe                    |
| 0xff has wide character representation: 0x00ff                    |
| The character is EOF.                                            |
************************************************************************/

Related Information

- “mblen() — Determine Length of a Multibyte Character” on page 219
- “mbtowc() — Convert Multibyte Character to a Wide Character” on page 233
- “mbrtowc() — Convert a Multibyte Character to a Wide Character (Restartable)” on page 223
- “mbsrtowcs() — Convert a Multibyte String to a Wide Character String (Restartable)” on page 227
- “setlocale() — Set Locale” on page 366
- “wcrtomb() — Convert a Wide Character to a Multibyte Character (Restartable)” on page 478
- “wcsrtombs() — Convert Wide Character String to Multibyte String (Restartable)” on page 505
- “<stdio.h>” on page 13
- “<wchar.h>” on page 16

_C_Get_Ssn_Handle() — Handle to C Session

Format

```c
#include <stdio.h>
_SSN_HANDLE_T _C_Get_Ssn_Handle (void)
```

Language Level

ILE C Extension

Threading

Yes

Description

Returns a handle to the C session for use with Dynamic Screen Manager (DSM) APIs.
Return Value

The _C_Get_Ssn_Handle() function returns a handle to the C session. If an error occurs, _SSN_HANDLE_T is set to zero. See the APIs topic in the Information Center for more information about using the _C_Get_Ssn_Handle() function with DSM APIs.

calloc() — Reserve and Initialize Storage

Format

```c
#include <stdlib.h>
void *calloc(size_t num, size_t size);
```

Language Level

ANSI

Threadsafe

Yes

Description

The calloc() function 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.

Return Value

The calloc() function returns a pointer to the reserved space. The storage space to which the return value points is suitably aligned for storage of any type of object. To get a pointer to a type, use a type cast on the return value. The return value is NULL if there is not enough storage, or if num or size is 0.

Notes:

1. All heap storage is associated with the activation group of the calling function. As such, storage should be allocated, deallocated, and reallocated within the same activation group. You cannot allocate heap storage within one activation group and deallocate or reallocate that storage from a different activation group. For more information about activation groups, see the ILE Concepts manual.

2. To use teraspace storage instead of single-level store storage without changing the C source code, specify the TERASPACE(*YES *TSIFC) parameter on the compiler command. This maps the calloc() library function to _C_TScalloc(), its teraspace storage counterpart. The maximum amount of teraspace storage that can be allocated by each call to _C_TScalloc() is 2GB - 224, or 2147483424 bytes.

   For more information about teraspace storage, see the ILE Concepts manual.

3. If the Quick Pool memory manager has been enabled in the current activation group, then the storage is retrieved using Quick Pool memory manager. See “_C_Quickpool_Init() — Initialize Quick Pool Memory Manager” on page 93 for more information.

Example

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.
#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 )
    {
        for ( i = 0; i < num; ++i ) /* put values in arr */
            *index++ = i; /* using pointer no */
        for ( i = 0; i < num; ++i ) /* print the array out */
            printf( "array[%i ] = %i\n", i, array[i] );
    }
    else
    {
        perror( "Out of storage" );
        abort();
    }

} /* Output should be similar to: */

Enter the size of the array
array[ 0 ] = 0
array[ 1 ] = 1
array[ 2 ] = 2
*/

Related Information
• “_C_Quickpool_Debug() — Modify Quick Pool Memory Manager Characteristics” on page 91
• “_C_Quickpool_Init() — Initialize Quick Pool Memory Manager” on page 93
• “_C_Quickpool_Report() — Generate Quick Pool Memory Manager Report” on page 95
• “Heap Memory” on page 578
• “free() — Release Storage Blocks” on page 151
• “malloc() — Reserve Storage Block” on page 217
• “realloc() — Change Reserved Storage Block Size” on page 291
• “<stdlib.h>” on page 15

catclose() — Close Message Catalog

Format

#include <nl_types.h>
int catclose (nl_catd catd);

Language Level
XPG4

Threatsafe
Yes
Locale Sensitive
This function is not available when LOCALETYP(*CLD) is specified on the compilation command.

Integrated File System Interface
This function is not available when SYSIFCOPT(*NOIFSIO) is specified on the compilation command.

Description
The catclose() function closes the previously opened message catalog that is identified by catd.

Return Value
If the close is performed successfully, 0 is returned. Otherwise, -1 is returned indicating failure, which might happen if catd is not a valid message catalog descriptor.

The value of errno can be set to:

EBADF
The catalog descriptor is not valid.

EINTR
The function was interrupted by a signal.

Example

```c
#include <stdio.h>
#include <nl_types.h>
#include <locale.h>

/* Name of the message catalog is "/qsys.lib/mylib.lib/msgs.usrspc" */
int main(void) {
    nl_catd msg_file;
    char * my_msg;
    char * my_locale;
    setlocale(LC_ALL, NULL);
    msg_file = catopen("/qsys.lib/mylib.lib/msgs.usrspc", 0);
    if (msg_file != CATD_ERR)  {
        my_msg = catgets(msg_file, 1, 2, "oops");
        printf("%s\n", my_msg);
        catclose(msg_file);
    }
}
```

Related Information
- “catopen() — Open Message Catalog” on page 84
- “catgets() — Retrieve a Message from a Message Catalog” on page 82

**catgets() — Retrieve a Message from a Message Catalog**

Format

```c
#include <nl_types.h>
char *catgets(nl_catd catd, int set_id, int msg_id, char *s);
```
Language Level
XPG4

Threadsafe
Yes

Locale Sensitive
The behavior of this function might be affected by the LC_CTYPE category of the current locale. This function is not available when LOCALETYPE(*CLD) is specified on the compilation command. For more information, see “Understanding CCSIDs and Locales” on page 565.

Integrated File System Interface
This function is not available when SYSIFCOPT(*NOIFSIO) is specified on the compilation command.

Description
The catgets() function retrieves message msg_id, in set set_id from the message catalog that is identified by catd. catd is a message catalog descriptor that is returned by a previous call to catopen(). The s argument points to a default message which will be returned by catgets() if the identified message cannot be retrieved.

Return Value
If the message is retrieved successfully, then catgets() returns a pointer to the message string that is contained in the message catalog. The CCSID of the retrieved message is determined by the flags specified in the oflag parameter on the previous call to the catopen() function, when the message catalog file was opened.

- If the NL_CAT_JOB_MODE flag was specified, then the retrieved message is in the CCSID of the job.
- If the NL_CAT_CTYPE_MODE flag was specified, then the retrieved message is in the CCSID of the LC_CTYPE category of the current locale.
- If neither flag was specified, the CCSID of the retrieved message matches the CCSID of the message catalog file.

If the message is retrieved unsuccessfully, then a pointer to the default string s is returned.

The value of errno can be set to the following:

EBADF
The catalog descriptor is not valid.

ECONVERT
A conversion error occurred.

EINTR
The function was interrupted by a signal.
Example

```c
#include <stdio.h>
#include <nl_types.h>
#include <locale.h>
/* Name of the message catalog is "/qsys.lib/mylib.lib/msgs.usrspc" */
int main(void) {
    nl_catd msg_file;
    char * my_msg;
    char * my_locale;
    setlocale(LC_ALL, NULL);
    msg_file = catopen("/qsys.lib/mylib.lib/msgs.usrspc", 0);
    if (msg_file != CATD_ERR)  {
        my_msg = catgets(msg_file, 1, 2, "oops");
        printf("%s\n", my_msg);
        catclose(msg_file);
    }
}
```

Related Information

- “catclose() — Close Message Catalog” on page 81
- “catopen() — Open Message Catalog” on page 84

**catopen() — Open Message Catalog**

**Format**

```c
#include <nl_types.h>
nl_catd catopen(const char *name, int oflag);
```

**Language Level**

XPG4

**Threadsafe**

Yes

**Locale Sensitive**

The behavior of this function might be affected by the LCMESSAGES category of the current locale. This function is not available when LOCALETYPE(*CLD) is specified on the compilation command. For more information, see “Understanding CCSIDs and Locales” on page 565.

**Integrated File System Interface**

This function is not available when SYSIFCOPT(*NOIFSIO) is specified on the compilation command.

**Description**

The catopen() function opens a message catalog, which must be done before a message can be retrieved. The NLSPATH environment variable and the LCMESSAGES category are used to find the specified message catalog if no slash (/) characters are found in the name. If the name contains one or more slash (/) characters, then the name is interpreted as a path name of the catalog to open.
If there is no NLSPATH environment variable, or if a message catalog cannot be found in the path specified by NLSPATH, then a default path is used. The default path might be affected by the setting of the LANG environment variable; if the NL_CAT_LOCALE flag is set in the oflag parameter or if the LANG environment variable is not set, the default path might be affected by the LC_MESSAGES locale category.

Three values can be specified for the oflag parameter: NL_CAT_LOCALE, NL_CAT_JOB_MODE, and NL_CAT_CTYPE_MODE. NL_CAT_JOB_MODE and NL_CAT_CTYPE_MODE are mutually exclusive. If the NL_CAT_JOB_MODE and NL_CAT_CTYPE_MODE flags are both set in the oflag parameter, the catopen() function will fail with a return value of CATD_ERR.

If you want the catalog messages to be converted to the job CCSID before they are returned by the catgets() function, set the parameter to NL_CAT_JOB_MODE. If you want the catalog messages to be converted to the LC_CTYPE CCSID before they are returned by catgets(), set the parameter to NL_CAT_CTYPE_MODE. If you do not set the parameter to NL_CAT_JOB_MODE or NL_CAT_CTYPE_MODE, the messages are returned without conversion and are in the CCSID of the message file.

The message catalog descriptor will remain valid until it is closed by a call to catclose(). If the LC_MESSAGES locale category is changed, it might invalidate existing open message catalogs.

**Note:** The name of the message catalog must be a valid integrated file system file name.

**Return Value**

If the message catalog is opened successfully, then a valid catalog descriptor is returned. If catopen() is unsuccessful, then it returns CATD_ERR ((nl_catd)-1).

The catopen() function might fail under the following conditions, and the value of errno can be set to:

- **EACCES**
  Insufficient authority to read the message catalog specified, or to search the component of the path prefix of the message catalog specified.

- **ECONVERT**
  A conversion error occurred.

- **EMFILE**
  NL_MAXOPEN message catalogs are currently open.

- **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.
Example

```c
#include <stdio.h>
#include <nl_types.h>
#include <locale.h>
/* Name of the message catalog is "/qsys.lib/mylib.lib/msgs.usrspc" */

int main(void) {
    nl_catd msg_file;
    char * my_msg;
    char * my_locale;
    setlocale(LC_ALL, NULL);
    msg_file = catopen("/qsys.lib/mylib.lib/msgs.usrspc", 0);
    if (msg_file != CATD_ERR)  {
        my_msg = catgets(msg_file, 1, 2, "oops");
        printf("%s\n", my_msg);
        catclose(msg_file);
    }
}
```

Related Information
- “catclose() — Close Message Catalog” on page 81
- “catgets() — Retrieve a Message from a Message Catalog” on page 82

**ceil() — Find Integer >=Argument**

**Format**

```c
#include <math.h>
double ceil(double x);
```

**Language Level**

ANSI

**Threading-safe**

Yes

**Description**

The `ceil()` function computes the smallest integer that is greater than or equal to `x`.

**Return Value**

The `ceil()` function returns the integer as a double value.

**Example**

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.
```c
#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 = %.2f ; z = %.2f\n", y, z);
}
/*********************** Output should be similar to: ***********************
   y = 2.00 ; z = -1.00
   *********************************************/
```

Related Information
- “floor() — Find Integer <=Argument” on page 132
- “fmod() — Calculate Floating-Point Remainder” on page 133
- “<math.h>” on page 6

---

clearerr() — Reset Error Indicators

**Format**
```c
#include <stdio.h>
void clearerr (FILE *stream);
```

**Language Level**
ANSI

**Threadsafe**
Yes

**Description**
The `clearerr()` function resets the error indicator and end-of-file indicator for the specified `stream`. Once set, the indicators for a specified stream remain set until your program calls the `clearerr()` function or the `rewind()` function. The `fseek()` function also clears the end-of-file indicator. The ILE C/C++ runtime environment does not automatically clear error or end of file indicators.

**Return Value**
There is no return value.

The value of errno can be set to:

**Value**
- **Meaning**

  **EBADF**
  - The file pointer or descriptor is not valid.

  **ENOTOPEN**
  - The file is not open.

  **ESTDIN**
  - stdin cannot be opened.
**EIOERROR**  
A non-recoverable I/O error occurred.

**EIORECERR**  
A recoverable I/O error occurred.

**Example**  
This example reads a data stream, and then checks that a read error has not occurred.

```c
#include <stdio.h>
#include <stdlib.h>

FILE *stream;
int c;

int main(void)
{
  if ((stream = fopen("mylib/myfile", "r")) != NULL)
  {
    if ((c=getc(stream)) == EOF)
    {
      if (ferror(stream))
      {
        perror("Read error");
        clearerr(stream);
      }
    }
  }
  else
    exit(0);
}
```

**Related Information**

- “feof() — Test End-of-File Indicator” on page 119
- “ferror() — Test for Read/Write Errors” on page 120
- “fseek() – fseeko() — Reposition File Position” on page 157
- “perror() — Print Error Message” on page 249
- “rewind() — Adjust Current File Position” on page 303
- “strerror() — Set Pointer to Runtime Error Message” on page 396
- “<stdio.h>” on page 13

---

**clock() — Determine Processor Time**

**Format**

```c
#include <time.h>
clock_t clock(void);
```

**Language Level**

ANSI

**Threadsafe**

Yes

**Description**

The `clock()` function returns an approximation of the processor time used by the program since the beginning of an implementation-defined time-period that is related to the process invocation. To obtain
the time in seconds, divide the value that is returned by \texttt{clock()} by the value of the macro \texttt{CLOCKS\_PER\_SEC}.

\textbf{Return Value}

If the value of the processor time is not available or cannot be represented, the \texttt{clock()} function returns the value \texttt{(clock_t)-1}.

To measure the time spent in a program, call \texttt{clock()} at the start of the program, and subtract its return value from the value returned by subsequent calls to \texttt{clock()}. On other platforms, you cannot always rely on the \texttt{clock()} function because calls to the \texttt{system()} function might reset the clock.

\textbf{Example}

This example prints the time that has elapsed since the program was called.

```c
#include <time.h>
#include <stdio.h>

double time1, timedif;        /* use doubles to show small values */
int main(void)
{
    int  i;
    time1 = (double) clock();    /* get initial time */
    time1 = time1 / CLOCKS_PER_SEC;    /* in seconds */
    /* running the FOR loop 10000 times */
    for (i=0; i<10000; i++)
    {
        /* call clock a second time */
        timedif = ( ((double) clock()) / CLOCKS_PER_SEC) - time1;
        printf("The elapsed time is %lf seconds\n", timedif);
    }
}
```

\textbf{Related Information}

- “\texttt{difftime()} — Compute Time Difference” on page 108
- “\texttt{difftime64()} — Compute Time Difference” on page 109
- “\texttt{time()} — Determine Current Time” on page 441
- “\texttt{time64()} — Determine Current Time” on page 443
- “\texttt{<time.h>” on page 16

\textbf{\texttt{cos}() — Calculate Cosine}

\textbf{Format}

```c
#include <math.h>
double cos(double x);
```

\textbf{Language Level}

ANSI

\textbf{Threadsafe}

Yes
Description
The cos() function calculates the cosine of \( x \). The value \( x \) is expressed in radians. If \( x \) is too large, a partial loss of significance in the result might occur.

Return Value
The cos() function returns the cosine of \( x \). The value of errno can be set to either EDOM or ERANGE.

Example
This example calculates \( y \) to be the cosine of \( x \).

```c
#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 should be similar to: *******************
cos( 7.200000 ) = 0.608351
*/

Related Information
• “acos() — Calculate Arccosine” on page 62
• “cosh() — Calculate Hyperbolic Cosine” on page 90
• “sin() — Calculate Sine” on page 376
• “sinh() — Calculate Hyperbolic Sine” on page 377
• “tan() — Calculate Tangent” on page 439
• “tanh() — Calculate Hyperbolic Tangent” on page 440
• “<math.h>” on page 6

cosh() — Calculate Hyperbolic Cosine

Format
```
#include <math.h>
double cosh(double x);
```

Language Level
ANSI

Threading
Yes

Description
The cosh() function calculates the hyperbolic cosine of \( x \). The value \( x \) is expressed in radians.
Return Value
The cosh() function returns the hyperbolic cosine of x. If the result is too large, cosh() returns the value HUGE_VAL and sets errno to ERANGE.

Example
This example calculates y to be the hyperbolic cosine of x.

```c
#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 should be similar to: ***********************
cosh( 7.200000 ) = 669.715755
*/
```

Related Information
• “acos() — Calculate Arccosine” on page 62
• “cos() — Calculate Cosine” on page 89
• “sin() — Calculate Sine” on page 376
• “sinh() — Calculate Hyperbolic Sine” on page 377
• “tan() — Calculate Tangent” on page 439
• “tanh() — Calculate Hyperbolic Tangent” on page 440
• “<math.h>” on page 6

_C_Quickpool_Debug() — Modify Quick Pool Memory Manager Characteristics

Format
```c
#include <stdlib.h>
_C_Quickpool_Debug_T _C_Quickpool_Debug(_C_Quickpool_Debug_T *newval);
```

Language Level
Extended

Threading
Yes

Description
The _C_Quickpool_Debug() function modifies Quick Pool memory manager characteristics. Environment variables can also be used to configure this support (reference section “Environment Variables” on page 586).

The parameters for _C_Quickpool_Debug() are as follows:
newval
   A pointer to a _C_Quickpool_Debug_T structure. The structure contains the following fields:

   **flags**
   An unsigned integer value that indicates the characteristics to be modified. The flags field can
   contain the following values (which can be used in any combination):

   _C_INIT_MALLOC
   Initializes all allocated storage to a specified value.

   _C_INIT_FREE
   Initializes all freed storage to a specified value.

   _C_COLLECT_STATS
   Collects statistics on the Quick Pool memory manager for use with the
   _C_Quickpool_Report() function.

   **malloc_val**
   A 1-byte unsigned character value that is used to initialize each byte of allocated memory. This
   field is in use only when the _C_INIT_MALLOC flag is specified.

   **free_val**
   A 1-byte unsigned character value that is used to initialize each byte of freed memory. This field is
   in use only when the _C_INIT_FREE flag is specified.

If the value of newval is NULL, a structure containing the current Quick Pool memory manager
characteristics is returned and none of the Quick Pool memory manager characteristics are modified.

**Return Value**

The return value is a structure that contains the _C_Quickpool_Debug() values before the changes
requested by the current function call are made. This value can be used on a later call to restore the
_C_Quickpool_Debug() values to a prior state.

**Example**

The following example uses _C_Quickpool_Debug() with the _C_INIT_MALLOC and _C_INIT_FREE
flags to initialize memory on the malloc and free functions.
```c
#include <stdlib.h>
#include <stdio.h>

int main(void) {
    char *p;
    char *mtest = "AAAAAAAAAA";
    char *ftest = "BBBBBBBBBB";
    unsigned int cell_sizes[2] = { 16, 64 };
    unsigned int cells_per_extent[2] = { 16, 16 };
    _C_Quickpool_Debug_T dbgVals = { _C_INIT_MALLOC | _C_INIT_FREE, 'A', 'B' };

    if (_C_Quickpool_Init(2, cell_sizes, cells_per_extent)) {
        printf("Error initializing Quick Pool memory manager.\n");
        return -1;
    }
    _C_Quickpool_Debug(&dbgVals);
    if ([(p = malloc(10)) == NULL]) {
        printf("Error during malloc.\n");
        return -2;
    }
    if (memcmp(p, mtest, 10)) {
        printf("malloc test failed\n");
    }
    free(p);
    if (memcmp(p, ftest, 10)) {
        printf("free test failed\n");
    }
    printf("Test successful!\n");
    return 0;
}
```

**Related Information**

- "_C_Quickpool_Init() — Initialize Quick Pool Memory Manager“ on page 93
- "_C_Quickpool_Report() — Generate Quick Pool Memory Manager Report” on page 95
- “<stdlib.h>” on page 15
- “Heap Memory” on page 578

---

_**_C_Quickpool_Init() — Initialize Quick Pool Memory Manager_**_

**Format**

```c
#include <stdlib.h>
int _C_Quickpool_Init(unsigned int numpools, unsigned int *cell_sizes, unsigned int *num_cells);
```

**Language Level**

Extended

**Threading**

Yes

**Description**

When the _C_Quickpool_Init() function is called, all subsequent calls to memory manager functions (malloc, calloc, realloc, and free) in the same activation group use the Quick Pool memory manager. This memory manager provides improved performance for some applications.

The Quick Pool memory manager breaks memory up into a series of pools. Each pool is broken up into a number of cells with identical sizes. The number of pools, the size of cells in each pool, and the number of
cells in each pool extent is set using the _C_Quickpool_Init() function. Environment variables can also be used to configure this support (reference section “Environment Variables” on page 586).

Suppose that a user wants to define four pools, each of which contains 64 cells. The first pool will have cells which are 16 bytes in size; the second pool will have cells which are 256 bytes in size; the third pool will have cells which are 1024 bytes in size; and the fourth pool will have cells which are 2048 bytes in size. When a request for storage is made, the memory manager assigns the request to a pool first. The memory manager compares the size of storage in the request with the size of the cells in a given pool.

In this example, the first pool satisfies requests between 1 and 16 bytes in size; the second pool satisfies requests between 17 and 256 bytes in size; the third pool satisfies requests between 257 and 1024 bytes in size, and the fourth pool satisfies requests between 1025 and 2048 bytes in size. Any requests larger than the largest cell size are allocated through the default memory manager.

After the pool has been assigned, the free queue for the pool is examined. Each pool has a free queue that contains cells that have been freed and have not yet been reallocated. If there is a cell on the free queue, the cell is removed from the free queue and returned; otherwise, the cell is retrieved from the current extent for the pool. An extent is a collection of cells that are allocated as one block. Initially, a pool has no extents.

When the first request comes in for a pool, an extent is allocated for the pool and the request is satisfied from that extent. Later requests for the pool are also satisfied by that extent until the extent is exhausted. When an extent is exhausted, a new extent is allocated for the pool. If a new extent cannot be allocated, it assumes that a memory problem exists. An attempt will be made to allocate the storage using the default memory manager. If the attempt is not successful, the NULL value is returned.

numpools

The number of pools to use for the Quick Pool memory manager. This parameter can have a value between 1 and 64.

cell_sizes

An array of unsigned integer values. The number of entries in the array is equal to the number specified on the numpools parameter. Each entry specifies the number of bytes in a cell for a given pool. These values must be multiples of 16 bytes. If a value is specified that is not a multiple of 16 bytes, the cell size is rounded up to the next larger multiple of 16 bytes. The minimum valid value is 16 bytes and the maximum valid value is 4096 bytes.

num_cells

An array of unsigned integer values. The number of entries in the array is equal to the number specified on the numpools parameter. Each entry specifies the number of cells in a single extent for the corresponding pool. Each value can be any non-negative number, but the total size of each extent may be limited due to architecture constraints. A value of zero indicates that the implementation should choose a large value.

Here is the call to _C_Quickpool_Init() for the preceding example:

```c
unsigned int cell_sizes[4] = { 16, 256, 1024, 2048 };
unsigned int cells_per_extent[4] = { 64, 64, 64, 64 };
rc = _C_Quickpool_Init(4, /* number of pools */
    cell_sizes, /* cell sizes for each pool */
    cells_per_extent); /* extent sizes for each pool */
```

Return Value

The follow list shows the return values for the _C_Quickpool_Init() function:

0  Success

-1  An alternate memory manager has already been enabled for this activation group.

-2  Error allocating storage for control structures.
An invalid number of pools was specified.

_CI_Quickpool_Init() was called from an invalid activation group.

An unexpected exception occurred when _CI_Quickpool_Init() was running.

Example

The following example uses _CI_Quickpool_Init() to enable Quick Pool memory allocation.

```c
#include <stdlib.h>
#include <stdio.h>
int main(void) {
    char *p;
    unsigned int cell_sizes[2] = {16, 64};
    unsigned int cells_per_extent[2] = {16, 16};
    if (_C_Quickpool_Init(2, cell_sizes, cells_per_extent) {
        printf("Error initializing Quick Pool memory manager.\n");
        return -1;
    }
    if ((p = malloc(10)) == NULL) {
        printf("Error during malloc.\n");
        return -2;
    }
    free(p);
    printf("Test successful!\n");
    return 0;
}
```

Related Information

• “_C_Quickpool_Debug() — Modify Quick Pool Memory Manager Characteristics” on page 91
• “_C_Quickpool_Report() — Generate Quick Pool Memory Manager Report” on page 95
• “<stdlib.h>” on page 15
• “Heap Memory” on page 578

_C_Quickpool_Report() — Generate Quick Pool Memory Manager Report

Format

```c
#include <stdlib.h>
void _C_Quickpool_Report(void);
```

Language Level

Extended

Threadsafe

Yes

Description

The _C_Quickpool_Report() function generates a spooled file that contains a snapshot of the memory used by the Quick Pool memory manager in the current activation group. If the Quick Pool memory manager has not been enabled for the current activation group or if statistics collection has not been enabled, the report will be a message that indicates no data is collected.
If the Quick Pool memory manager has been enabled and statistics collection has been enabled, the report that is generated indicates the number of allocation attempts for each 16 bytes of memory during the time that statistics collection was enabled. In addition, the report indicates the maximum number of outstanding allocations (peak allocations) that is reached for each pool. If no storage requests are made for a given range of memory, that range of memory will not be included in the report. No output is generated for allocations larger than the maximum cell size (4096 bytes).

**Return Value**

There is no return value for the function.

**Example**

The following example uses _C_Quickpool_Init() to enable Quick Pool memory manager. It uses the _C_COLLECT_STATS flag to collect information. The collected information is printed using _C_Quickpool_Report().

```c
#include <stdlib.h>
#include <stdio.h>
int main(void) {
    char *p;
    int   i;
    unsigned int cell_sizes[2]       = { 16, 64 };  
    unsigned int cells_per_extent[2] = { 16, 16 };  
    _C_Quickpool_Debug_T  dbgVals = { _C_COLLECT_STATS, 'A', 'B' };
    if (_C_Quickpool_Init(2, cell_sizes, cells_per_extent) {
        printf("Error initializing Quick Pool memory manager.\n");
        return -1;
    }
    _C_Quickpool_Debug(&dbgVals);
    for (i = 1; i <= 64; i++) {
        p = malloc(i);
        free(p);
    }
    p = malloc(128);
    free(p);
    _C_Quickpool_Report();
    return 0;
}
```

/*Spooled File Output should be similar to:***********/
Pool 1 (16 bytes, 1 peak allocations):
  1-16 bytes: 16 allocations
Pool 2 (64 bytes, 1 peak allocations):
  17-32 bytes: 16 allocations
  33-48 bytes: 16 allocations
  49-64 bytes: 16 allocations
Remaining allocations smaller than the largest cell size (4096 bytes):
  113-128 bytes: 1 allocations
*******************************************************************************/

**Related Information**

- “_C_Quickpool_Debug() — Modify Quick Pool Memory Manager Characteristics” on page 91
- “_C_Quickpool_Init() — Initialize Quick Pool Memory Manager” on page 93
- “<stdlib.h>” on page 15
- “Heap Memory” on page 578

**ctime() — Convert Time to Character String**

**Format**

```c
#include <time.h>
char *ctime(const time_t *time);
```
Language Level
ANSI

Threadsafe
No
Use ctime_r() instead.

Locale Sensitive
The behavior of this function might be affected by the LC_TOD category of the current locale. For more information, see “Understanding CCSIDs and Locales” on page 565.

Description
The ctime() function converts the time value pointed to by time to local time in the form of a character string. A time value is usually obtained by a call to the time() function.

The string result that is produced by ctime() contains exactly 26 characters and has the format:

"%.3s %.3s%3d %.2d:%.2d:%.2d %d
"

For example:

Mon Jul 16 02:03:55 1987

The ctime() function uses a 24-hour clock format. The days are abbreviated to: Sun, Mon, Tue, Wed, Thu, Fri, and Sat. The months are abbreviated to: Jan, Feb, Mar, Apr, May, Jun, Jul, Aug, Sep, Oct, Nov, and Dec. All fields have a constant width. Dates with only one digit are preceded with a zero. The new-line character (\n) and the null character (\0) occupy the last two positions of the string.

Return Value
The ctime() function returns a pointer to the character string result. If the function is unsuccessful, it returns NULL. A call to the ctime() function is equivalent to:

asctime(localtime(&anytime))

Note: The asctime() and ctime() functions, and other time functions can use a common, statically allocated buffer to hold the return string. Each call to one of these functions might destroy the result of the previous call. The asctime_r(), ctime_r(), gmtime_r(), and localtime_r() functions do not use a common, statically allocated buffer to hold the return string. These functions can be used in place of asctime(), ctime(), gmtime(), and localtime() if reentrancy is desired.

Example
This example polls the system clock using time(). It then prints a message giving the current date and time.

```c
#include <time.h>
#include <stdio.h>

int main(void)
{
    time_t ltime;
    time(&ltime);
    printf("the time is %s", ctime(&ltime));
}
```
ctime64() — Convert Time to Character String

Format

```c
#include <time.h>
char *ctime64(const time64_t *time);
```

Language Level

ILE C Extension

Threadsafe

No

Use ctime64_r() instead.

Locale Sensitive

The behavior of this function might be affected by the LC_TOD category of the current locale. For more information, see “Understanding CCSIDs and Locales” on page 565.

Description

The ctime64() function converts the time value pointed to by time to local time in the form of a character string. A time value is usually obtained by a call to the time64() function.
The string result that is produced by the \texttt{ctime64()} function contains exactly 26 characters and has the format:

\begin{verbatim}
"%.3s %.3s%3d %.2d:.2d:.2d %d\n"
\end{verbatim}

For example:

\begin{verbatim}
Mon Jul 16 02:03:55 1987\n\0
\end{verbatim}

The \texttt{ctime64()} function uses a 24-hour clock format. The month and day abbreviations used are retrieved from the locale. All fields have a constant width. Dates with only 1 digit are preceded with a zero. The new-line character (\textbackslash{n}) and the null character (\textbackslash{0}) occupy the last two positions of the string.

\textbf{Return Value}

The \texttt{ctime64()} function returns a pointer to the character string result. If the function is unsuccessful, it returns NULL. A call to the \texttt{ctime64()} function is equivalent to:

\begin{verbatim}
asctime(localtime64(&anytime))
\end{verbatim}

\textbf{Note:} The \texttt{asctime()} and \texttt{ctime64()} functions, and other time functions can use a common, statically allocated buffer to hold the return string. Each call to one of these functions might destroy the result of the previous call. The \texttt{asctime\_r()}, \texttt{ctime64\_r()}, \texttt{gmtime64\_r()}, and \texttt{localtime64\_r()} functions do not use a common, statically allocated buffer to hold the return string. These functions can be used in place of \texttt{asctime()}, \texttt{ctime64()}, \texttt{gmtime64()}, and \texttt{localtime64()}, if reentrancy is desired.

\textbf{Example}

This example polls the system clock using \texttt{time64()}. It then prints a message that gives the current date and time.

\begin{verbatim}
#include <time.h>
#include <stdio.h>

int main(void)
{
    time64_t ltime;
    time64(&ltime);
    printf("the time is %s", ctime64(&ltime));
}
\end{verbatim}

\textbf{Related Information}

- “\texttt{asctime()} — Convert Time to Character String” on page 63
- “\texttt{asctime\_r()} — Convert Time to Character String (Restartable)” on page 65
- “\texttt{ctime()} — Convert Time to Character String” on page 96
- “\texttt{ctime64()} — Convert Time to Character String” on page 98
- “\texttt{ctime64\_r()} — Convert Time to Character String (Restartable)” on page 101
- “\texttt{gmtime()} — Convert Time” on page 183
- “\texttt{gmtime64()} — Convert Time” on page 185
- “\texttt{gmtime64\_r()} — Convert Time (Restartable)” on page 189
- “\texttt{gmtime\_r()} — Convert Time (Restartable)” on page 187
- “\texttt{localtime()} — Convert Time” on page 207
- “\texttt{localtime64()} — Convert Time” on page 208
- “\texttt{localtime64\_r()} — Convert Time (Restartable)” on page 211
- “\texttt{localtime\_r()} — Convert Time (Restartable)” on page 210
ctime_r() — Convert Time to Character String (Restartable)

**Format**

```c
#include <time.h>
char *ctime_r(const time_t *time, char *buf);
```

**Language Level**

XPG4

**Threadsafe**

Yes

**Locale Sensitive**

The behavior of this function might be affected by the LC_TOD category of the current locale. For more information, see “Understanding CCSIDs and Locales” on page 565.

**Description**

This function is the restartable version of the ctime() function.

The ctime_r() function converts the time value pointed to by time to local time in the form of a character string. A time value is usually obtained by a call to the time() function.

The string result that is produced by the ctime_r() function contains exactly 26 characters and has the format:

```
%.3s %3d %2d:%2d:%2d %d
```

For example:

```
Mon Jul 16 02:03:55 1987
```

The ctime_r() function uses a 24-hour clock format. The days are abbreviated to: Sun, Mon, Tue, Wed, Thu, Fri, and Sat. The months are abbreviated to: Jan, Feb, Mar, Apr, May, Jun, Jul, Aug, Sep, Oct, Nov, and Dec. All fields have a constant width. Dates with only one digit are preceded with a zero. The new-line character (\n) and the null character (\0) occupy the last two positions of the string.

**Return Value**

The ctime_r() function returns a pointer to the character string result. If the function is unsuccessful, it returns NULL. A call to ctime_r() is equivalent to:

```
asctime_r(localtime_r(&anytime, buf2), buf)
```
where buf is a pointer to char.

**Example**

This example polls the system clock using `ctime_r()`. It then prints a message giving the current date and time.

```c
#include <time.h>
#include <stdio.h>

int main(void)
{
    time_t ltime;
    char buf[50];
    time(&ltime);
    printf("the time is %s", ctime_r(&ltime, buf));
}
```

**Related Information**

- “asctime() — Convert Time to Character String” on page 63
- “asctime_r() — Convert Time to Character String (Restartable)” on page 65
- “ctime() — Convert Time to Character String” on page 96
- “ctime64() — Convert Time to Character String” on page 98
- “ctime64_r() — Convert Time to Character String (Restartable)” on page 101
- “gmtime() — Convert Time” on page 183
- “gmtime64() — Convert Time” on page 185
- “gmtime64_r() — Convert Time (Restartable)” on page 189
- “gmtime_r() — Convert Time (Restartable)” on page 187
- “localtime() — Convert Time” on page 207
- “localtime64() — Convert Time” on page 208
- “localtime64_r() — Convert Time (Restartable)” on page 211
- “localtime_r() — Convert Time (Restartable)” on page 210
- “mktime() — Convert Local Time” on page 241
- “mktime64() — Convert Local Time” on page 242
- “strftime() — Convert Date/Time to String” on page 399
- “time() — Determine Current Time” on page 441
- “time64() — Determine Current Time” on page 443
- “<time.h>” on page 16

**ctime64_r() — Convert Time to Character String (Restartable)**

**Format**

```c
#include <time.h>
char *ctime64_r(const time64_t *time, char *buf);
```

**Language Level**

ILE C Extension
Threadsafe
Yes

Locale Sensitive
The behavior of this function might be affected by the LC_TOD category of the current locale. For more information, see “Understanding CCSIDs and Locales” on page 565.

Description
This function is the restartable version of the ctime64() function.
The ctime64() function converts the time value pointed to by time to local time in the form of a character string. A time value is usually obtained by a call to the time64() function.
The string result that is produced by the ctime64_r() function 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\n0
```
The ctime64_r() function uses a 24-hour clock format. The month and day abbreviation used are retrieved from the locale. All fields have a constant width. Dates with only 1 digit are preceded with a zero. The new-line character (\n) and the null character (\0) occupy the last two positions of the string.

Return Value
The ctime64_r() function returns a pointer to the character string result. If the function is unsuccessful, it returns NULL. A call to the ctime64_r() function is equivalent to:

```
asctime_r(localtime64_r(&anytime, buf2), buf)
```

Example
This example polls the system clock using time64(). It then prints a message, giving the current date and time.

```c
#include <time.h>
#include <stdio.h>

int main(void)
{
  time64_t ltime;
  char buf[50];

  time64(&ltime);
  printf("the time is %s", ctime64_r(&ltime, buf));
}
```

Related Information
- “asctime() — Convert Time to Character String” on page 63
- “asctime_r() — Convert Time to Character String (Restartable)” on page 65
- “ctime() — Convert Time to Character String” on page 96
- “ctime64() — Convert Time to Character String” on page 98
- “ctime_r() — Convert Time to Character String (Restartable)” on page 100
- “gmtime() — Convert Time” on page 183
- “gmtime64() — Convert Time” on page 185
_C_TS_malloc_debug() — Determine amount of teraspace memory used (with optional dumps and verification)

Format

```
#include <mallocinfo.h>
int _C_TS_malloc_debug(unsigned int dump_level, unsigned int verify_level,
                      struct _C_mallinfo_t *output_record, size_t sizeofoutput);
```

Language Level

Extended

Threadsafe

Yes

Description

The _C_TS_malloc_debug() function determines the amount of teraspace memory used and returns the information within the given output_record structure. If the given dump_level parameter is greater than 0, it also dumps the internal memory structures used to stdout. If the given verify_level parameter is greater than 0, it also performs verification checks for the internal memory structures. If a verification fails, a message is generated to stdout indicating the failure. If both the dump_level and verify_level parameters are 0, this function provides the same behavior as the _C_TS_malloc_info function.

The following macros are defined within the <mallocinfo.h> include file to be specified for the dump_level parameter:

<table>
<thead>
<tr>
<th>Macro</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>_C_NO_DUMPS</td>
<td>No information is dumped</td>
</tr>
<tr>
<td>_C_DUMP_TOTALS</td>
<td>Overall totals and totals for each chunk are printed</td>
</tr>
<tr>
<td>_C_DUMP_CHUNKS</td>
<td>Additional information about each chunk is printed</td>
</tr>
<tr>
<td>_C_DUMP_NODES</td>
<td>Additional information for all nodes within each chunk is printed</td>
</tr>
<tr>
<td>_C_DUMP_TREE</td>
<td>Additional information for the cartesian tree used to track free nodes is printed</td>
</tr>
<tr>
<td>Macro</td>
<td>Description</td>
</tr>
<tr>
<td>-----------------------</td>
<td>--------------------------------------------------</td>
</tr>
<tr>
<td>_C_DUMP_ALL</td>
<td>All available information is printed</td>
</tr>
</tbody>
</table>

The following macros are defined within the `<mallocinfo.h>` include file to be specified for the `verify_level` parameter:

<table>
<thead>
<tr>
<th>Macro</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>_C_NO_CHECKS</td>
<td>No verification checks are performed</td>
</tr>
<tr>
<td>_C_CHECK_TOTALS</td>
<td>Totals are verified for correctness</td>
</tr>
<tr>
<td>_C_CHECK_CHUNKS</td>
<td>Additional verifications are performed for each chunk</td>
</tr>
<tr>
<td>_C_CHECK_NODES</td>
<td>Additional verifications are performed for all nodes within each chunk</td>
</tr>
<tr>
<td>_C_CHECK_TREE</td>
<td>Additional verifications are performed for the cartesian tree used to track free nodes</td>
</tr>
<tr>
<td>_C_CHECK_ALL</td>
<td>All verifications are performed</td>
</tr>
<tr>
<td>_C_CHECK_ALL_AND_ABORT</td>
<td>All verifications are performed, and if any verification fails, the <code>abort()</code> function is called</td>
</tr>
</tbody>
</table>

**Note:** This function is for low-level debug of teraspace memory usage within an application.

**Return Value**
If successful, the function returns 0. If an error occurs, the function returns a negative value.

**Example**
This example prints the information returned from `_C_TS_malloc_debug()` to stdout. This program is compiled with `TERASPACE(*YES *TSIFC)`.
#include <stdio.h>
#include <stdlib.h>
#include <mallocinfo.h>

int main (void)
{
    _C_mallinfo_t info;
    int rc;
    void *m;

    /* Allocate a small chunk of memory */
    m = malloc(500);
    rc = _C_TS_malloc_debug(_C_DUMP_TOTALS,
                            _C_NO_CHECKS,
                            &info, sizeof(info));

    if (rc == 0) {
        printf("_C_TS_malloc_debug successful\n");
    }
    else {
        printf("_C_TS_malloc_debug failed (rc = %d)\n", rc);
    }

    free(m);
}

/****************************************************
The output should be similar to:
total_bytes        = 524288
allocated_bytes    = 688
unallocated_bytes  = 523600
allocated_blocks   = 1
unallocated_blocks = 1
requested_bytes    = 500
pad_bytes          = 12
overhead_bytes     = 176
Number of memory chunks  = 1
Total bytes              = 524288
Total allocated bytes    = 688
Total unallocated bytes  = 523600
Total allocated blocks   = 1
Total unallocated blocks = 1
Total requested bytes    = 500
Total pad bytes          = 12
Total overhead bytes      = 176
_C_TS_malloc_debug successful
****************************************************

Related Information
• "_C_TS_malloc_info() — Determine amount of teraspace memory used" on page 105
• calloc() — Reserve and Initialize Storage” on page 80
• “free() — Release Storage Blocks” on page 151
• "malloc() — Reserve Storage Block” on page 217
• "realloc() — Change Reserved Storage Block Size” on page 291
• "<mallocinfo.h>” on page 6
• "Heap Memory” on page 578

_C_TS_malloc_info() — Determine amount of teraspace memory used

Format
#include <mallocinfo.h>
int _C_TS_malloc_info(struct _C_mallinfo_t *output_record, size_t sizeofoutput);
Language Level
Extended

Threadsafe
Yes

Description
The _C_TS_malloc_info() function determines the amount of teraspace memory used and returns the information within the given output_record structure.

Note: This function is for low-level debug of teraspace memory usage within an application.

Return Value
If successful, the function returns 0. If an error occurs, the function returns a negative value.

Example
This example prints the information returned from _C_TS_malloc_info() to stdout. This program is compiled with TERASPACE(*YES *TSIFC).
```c
#include <stdio.h>
#include <stdlib.h>
#include <mallocinfo.h>

int main (void)
{
    _C_mallinfo_t info;
    int           rc;
    void         *m;

    /* Allocate a small chunk of memory */
    m = malloc(500);

    rc = _C_TS_malloc_info(&info, sizeof(info));
    if (rc == 0) {
        printf("Total bytes              = %llu\n", info.total_bytes);
        printf("Total allocated bytes    = %llu\n", info.allocated_bytes);
        printf("Total unallocated bytes  = %llu\n", info.unallocated_bytes);
        printf("Total allocated blocks   = %llu\n", info.allocated_blocks);
        printf("Total unallocated blocks = %llu\n", info.unallocated_blocks);
        printf("Total requested bytes    = %llu\n", info.requested_bytes);
        printf("Total pad bytes          = %llu\n", info.pad_bytes);
        printf("Total overhead bytes     = %llu\n", info.overhead_bytes);
    } else {
        printf("_C_TS_malloc_info failed (rc = %d)\n", rc);
    }
    free(m);
}

/****************************
The output should be similar to:
Total bytes              = 524288
Total allocated bytes    = 688
Total unallocated bytes  = 523600
Total allocated blocks   = 1
Total unallocated blocks = 1
Total requested bytes    = 500
Total pad bytes          = 12
Total overhead bytes     = 176
****************************

Related Information
• "_C_TS_malloc_debug() — Determine amount of teraspace memory used (with optional dumps and verification)" on page 103
• "calloc() — Reserve and Initialize Storage” on page 80
• “free() — Release Storage Blocks” on page 151
• "malloc() — Reserve Storage Block” on page 217
• “realloc() — Change Reserved Storage Block Size” on page 291
• “<mallocinfo.h>” on page 6
• “Heap Memory” on page 578
```
difftime() — Compute Time Difference

Format

```c
#include <time.h>
double difftime(time_t time2, time_t time1);
```

Language Level

ANSI

Threadsafe

Yes

Description

The `difftime()` function computes the difference in seconds between `time2` and `time1`.

Return Value

The `difftime()` function returns the elapsed time in seconds from `time1` to `time2` as a double precision number. Type `time_t` is defined in `<time.h>`.

Example

This example shows a timing application that uses `difftime()`. The example calculates how long, on average, it takes to find the prime numbers from 2 to 10 000.

```c
#include <time.h>
#include <stdio.h>
#define RUNS 1000
#define SIZE 10000
int mark[SIZE];

int main(void)
{
    time_t start, finish;
    int i, loop, n, num;
    time(&start);
    /* This loop finds the prime numbers between 2 and SIZE */
    for (loop = 0; loop < RUNS; ++loop)
    {
        for (n = 0; n < SIZE; ++n)
            mark[n] = 0;
        /* This loops marks all the composite numbers with -1 */
        for (num = 0, n = 2; n < SIZE; ++n)
            if ( ! mark[n])
                { /* There is no code here */
                    for (i = 2 * n; i < SIZE; i += n)
                        mark[i] = -1;
                    ++num;
                }
    time(&finish);
    printf("Program takes an average of %f seconds "
        "to find %d primes.\n",
        difftime(finish,start)/RUNS, num);
}

/********************  Output should be similar:  *****************/
The program takes an average of 0.106000 seconds to find 1229 primes.
```

108 IBM i: ILE C/C++ Runtime Library Functions
Related Information

• “asctime() — Convert Time to Character String” on page 63
• “asctime_r() — Convert Time to Character String (Restartable)” on page 65
• “ctime() — Convert Time to Character String” on page 96
• “ctime64() — Convert Time to Character String” on page 98
• “ctime64_r() — Convert Time to Character String (Restartable)” on page 101
• “ctime_r() — Convert Time to Character String (Restartable)” on page 100
• “difftime64() — Compute Time Difference” on page 109
• “gmtime() — Convert Time” on page 183
• “gmtime64() — Convert Time” on page 185
• “gmtime64_r() — Convert Time (Restartable)” on page 189
• “gmtime_r() — Convert Time (Restartable)” on page 187
• “localtime() — Convert Time” on page 207
• “localtime64() — Convert Time” on page 208
• “localtime64_r() — Convert Time (Restartable)” on page 211
• “localtime_r() — Convert Time (Restartable)” on page 210
• “mktime() — Convert Local Time” on page 241
• “mktime64() — Convert Local Time” on page 242
• “strftime() — Convert Date/Time to String” on page 399
• “time() — Determine Current Time” on page 441
• “time64() — Determine Current Time” on page 443
• “<time.h>” on page 16

difftime64() — Compute Time Difference

Format

```
#include <time.h>
double difftime64(time64_t time2, time64_t time1);
```

Language Level

ILE C Extension

Threatsafe

Yes

Description

The difftime64() function computes the difference in seconds between time2 and time1.

Return Value

The difftime64() function returns the elapsed time in seconds from time1 to time2 as a double precision number. Type time64_t is defined in <time.h>.
Example

This example shows a timing application that uses `difftime64()`. The example calculates how long, on average, it takes to find the prime numbers from 2 to 10 000.

```c
#include <time.h>
#include <stdio.h>

#define RUNS 1000
#define SIZE 10000

int mark[SIZE];

int main(void)
{
    time64_t start, finish;
    int i, loop, n, num;

    time64(&start);
    /* This loop finds the prime numbers between 2 and SIZE */
    for (loop = 0; loop < RUNS; ++loop)
    {
        for (n = 0; n < SIZE; ++n)
            mark[n] = 0;
        /* This loops marks all the composite numbers with -1 */
        for (num = 0, n = 2; n < SIZE; ++n)
            if (!mark[n])
                for (i = 2 * n; i < SIZE; i += n)
                    mark[i] = -1;
        ++num;
    }
    time64(&finish);
    printf("Program takes an average of %f seconds "
           "to find %d primes.\n",
           difftime64(finish,start)/RUNS, num);
}

/********************  Output should be similar:  *****************/
The program takes an average of 0.106000 seconds to find 1229 primes. */
```

Related Information

- “asctime() — Convert Time to Character String” on page 63
- “asctime_r() — Convert Time to Character String (Restartable)” on page 65
- “ctime() — Convert Time to Character String” on page 96
- “ctime64() — Convert Time to Character String” on page 98
- “ctime64_r() — Convert Time to Character String (Restartable)” on page 101
- “ctime_r() — Convert Time to Character String (Restartable)” on page 100
- “difftime() — Compute Time Difference” on page 108
- “gmtime() — Convert Time” on page 183
- “gmtime64() — Convert Time” on page 185
- “gmtime64_r() — Convert Time (Restartable)” on page 189
- “gmtime_r() — Convert Time (Restartable)” on page 187
- “localtime() — Convert Time” on page 207
- “localtime64() — Convert Time” on page 208
- “localtime64_r() — Convert Time (Restartable)” on page 211
- “localtime_r() — Convert Time (Restartable)” on page 210
- “mktime() — Convert Local Time” on page 241
- “mktime64() — Convert Local Time” on page 242
div() — Calculate Quotient and Remainder

Format

```c
#include <stdlib.h>
div_t div(int numerator, int denominator);
```

Language Level

ANSI

Threadsafe

Yes

However, only the function version is threadsafe. The macro version is NOT threadsafe.

Description

The `div()` function calculates the quotient and remainder of the division of `numerator` by `denominator`.

Return Value

The `div()` function returns a structure of type `div_t`, containing both the quotient `int quot` and the remainder `int rem`. If the return value cannot be represented, its value is undefined. If `denominator` is 0, an exception will be raised.

Example

This example uses `div()` to calculate the quotients and remainders for a set of two dividends and two divisors.
#include <stdlib.h>
#include <stdio.h>

int main(void)
{
    int num[2] = {45,-45};
    int den[2] = {7,-7};
    div_t ans; /* div_t is a struct type containing two ints:
                     'quot' stores quotient; 'rem' stores remainder */
    short i,j;

    printf("Results of division:\n");
    for (i = 0; i < 2; i++)
        for (j = 0; j < 2; j++)
        {
            ans = div(num[i],den[j]);
            printf("Dividend: %6d  Divisor: %6d", num[i], den[j]);
            printf("  Quotient: %6d  Remainder: %6d\n", ans.quot, ans.rem);
        }
}

 returned output should be similar to:

Results of 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

Example
• “ldiv() – lldiv() — Perform Long and Long Long Division” on page 201
• “<stdlib.h>” on page 15

erf() – erfc() — Calculate Error Functions

Format
#include <math.h>
double erf(double x);
double erfc(double x);

Language Level
ANSI

Threadsafe
Yes

Description
The erf() function calculates the error function of:

\[ 2\pi^{-1/2} \int_{0}^{x} e^{-t^2} dt \]

The erfc() function computes the value of 1.0 - erf(x). The erfc() function is used in place of erf() for large values of x.
Return Value
The `erf()` function returns a double value that represents the error function. The `erfc()` function returns a double value representing $1.0 - erf$.

Example
This example uses `erf()` and `erfc()` to compute the error function of two numbers.

```c
#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: \%lf\n", value);

    value = erfc(largex);      /* value = 2.088488e-45 */
    printf("Error value for 10.0: \%le\n", value);
}
/******************** Output should be similar to: ***************
Error value for 0.1: 0.112463
Error value for 10.0: 2.088488e-45
*/
```

Related Information
• “Bessel Functions” on page 75
• “`gamma()` — Gamma Function” on page 172
• “`<math.h>`” on page 6

exit() — End Program

Format
```c
#include <stdlib.h>
void exit(int status);
```

Language Level
ANSI

Threadsafe
Yes

Description
The `exit()` function returns control to the host environment from the program. It first calls all functions that are registered with the `atexit()` function, in reverse order; that is, the last one that is registered is the first one called. It deletes all buffers and closes all open files before ending the program.

The argument `status` can have a value from 0 to 255 inclusive, or be one of the macros `EXIT_SUCCESS` or `EXIT_FAILURE`. A `status` value of `EXIT_SUCCESS` or 0 indicates a normal exit; otherwise, another `status` value is returned.
Note: The exit() function cannot be called within an asynchronous signal handler. This means that when compiled with SYSIFCOPT(*ASYNCSIGNAL), exit() cannot be called in a signal handler.

Return Value

The exit() function returns both control and the value of status to the operating system.

Example

This example ends the program after deleting buffers and closing any open files if it cannot open the file myfile.

```c
#include <stdio.h>
#include <stdlib.h>

FILE *stream;

int main(void)
{
    if ((stream = fopen("mylib/myfile", "r")) == NULL)
    {
        perror("Could not open data file");
        exit(EXIT_FAILURE);
    }
}
```

Related Information

- “abort() — Stop a Program” on page 60
- “atexit() — Record Program Ending Function” on page 70
- “signal() — Handle Interrupt Signals” on page 374
- “<stdlib.h>” on page 15

exp() — Calculate Exponential Function

Format

```c
#include <math.h>
double exp(double x);
```

Language Level

ANSI

Threadsafe

Yes

Description

The exp() function calculates the exponential value of a floating-point argument \( x \) ( \( e^x \), where \( e \) equals 2.17128128...).

Return Value

If an overflow occurs, the exp() function returns HUGE_VAL. If an underflow occurs, it returns 0. Both overflow and underflow set errno to ERANGE. The value of errno can also be set to EDOM.
Example

This example calculates \( y \) as the exponential function of \( x \):

```c
#include <math.h>
#include <stdio.h>

int main(void)
{
    double x, y;
    x = 5.0;
    y = exp(x);
    printf("exp( %lf ) = %lf\n", x, y);
}

/*****************  Output should be similar to:  *****************
exp( 5.000000 ) = 148.413159
*/
```

Related Information

- “\log() — Calculate Natural Logarithm” on page 212
- “\log10() — Calculate Base 10 Logarithm” on page 213
- “\langle\text{math.h}\rangle” on page 6

\textbf{fabs()} — Calculate Floating-Point Absolute Value

\textbf{Format}

```c
#include <math.h>
double fabs(double x);
```

\textbf{Language Level}

ANSI

\textbf{Threadsafe}

Yes

\textbf{Description}

The `fabs()` function calculates the absolute value of the floating-point argument \( x \).

\textbf{Return Value}

The `fabs()` function returns the absolute value. There is no error return value.

\textbf{Example}

This example calculates \( y \) as the absolute value of \( x \):
```c
#include <math.h>
#include <stdio.h>

int main(void)
{
    double x, y;
    x = -5.6798;
    y = fabs(x);
    printf("fabs( %.1f ) = %.1f\n", x, y);
}

/*******************  Output should be similar to:  ***************
fabs( -5.679800 ) = 5.679800
*/
```

### Related Information
- “abs() — Calculate Integer Absolute Value” on page 61
- “labs() – llabs() — Calculate Absolute Value of Long and Long Long Integer” on page 199
- “<math.h>” on page 6

## fclose() — Close Stream

### Format
```c
#include <stdio.h>
int fclose(FILE *stream);
```

### Language Level
ANSI

### Threadsafe
Yes

### Description
The `fclose()` function closes a stream pointed to by `stream`. This function deletes all buffers that are associated with the stream before closing it. When it closes the stream, the function releases any buffers that the system reserved. When a binary stream is closed, the last record in the file is padded with null characters (\0) to the end of the record.

### Return Value
The `fclose()` function returns 0 if it successfully closes the stream, or EOF if any errors were detected. The value of `errno` can be set to:

<table>
<thead>
<tr>
<th>Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>ENOTOPEN</td>
<td>The file is not open.</td>
</tr>
<tr>
<td>EIOERROR</td>
<td>A non-recoverable I/O error occurred.</td>
</tr>
<tr>
<td>EIORECERR</td>
<td>A recoverable I/O error occurred.</td>
</tr>
</tbody>
</table>
ESCANFAILURE
The file was marked with a scan failure.

Note: The storage pointed to by the FILE pointer is freed by the fclose() function. After the use of the fclose() function, any attempt to use the FILE pointer is not valid.

Example
This example opens a file myfile for reading as a stream; then it closes this file.

```c
#include <stdio.h>
#define NUM_ALPHA 26

int main(void)
{
    FILE *stream;
    char buffer[NUM_ALPHA];
    if (( stream = fopen("mylib/myfile", "r"))!= NULL )
    {
        fread( buffer, sizeof( char ), NUM_ALPHA, stream );
        printf( "buffer = %s\n", buffer );
    }
    if (fclose(stream))   /* Close the stream. */
        perror("fclose error");
    else printf("File mylib/myfile closed successfully.\n");
}
```

Related Information
- “fflush() — Write Buffer to File” on page 121
- “fopen() — Open Files” on page 134
- “freopen() — Redirect Open Files” on page 153
- “<stdio.h>” on page 13

\textbf{fdopen() — Associates Stream With File Descriptor}

**Format**

```c
#include <stdio.h>
FILE *
fdopen(int handle, char *type);
```

**Language Level**

XPG4

**Threading**

Yes

**Integrated File System Interface**

This function is not available when SYSIFCOPT(*NOIFSIO) is specified on the compilation command.

**Description**

The \texttt{fdopen()} function associates an input or output stream with the file that is identified by \texttt{handle}. The \texttt{type} variable is a character string specifying the type of access that is requested for the stream. The variable contains one positional parameter that is followed by optional keyword parameters.

The possible values for the positional parameters are:
Mode
Description

r
Create a stream to read a text file. The file pointer is set to the beginning of the file.

w
Create a stream to write to a text file. The file pointer is set to the beginning of the file.

a
Create a stream to write, in append mode, at the end of the text file. The file pointer is set to the end of the file.

r+
Create a stream for reading and writing a text file. The file pointer is set to the beginning of the file.

w+
Create a stream for reading and writing a text file. The file pointer is set to the beginning of the file.

a+
Create a stream for reading or writing, in append mode, at the end of the text file. The file pointer is set to the end of the file.

rb
Create a stream to read a binary file. The file pointer is set to the beginning of the file.

wb
Create a stream to write to a binary file. The file pointer is set to the beginning of the file.

ab
Create a stream to write to a binary file in append mode. The file pointer is set to the end of the file.

r+b or rb+
Create a stream for reading and writing a binary file. The file pointer is set to the beginning of the file.

w+b or wb+
Create a stream for reading and writing a binary file. The file pointer is set to the beginning of the file.

a+b or ab+
Create a stream for reading and writing to a binary file in append mode. The file pointer is set to the end of the file.

Note: Use the w, w+, wb, wb+, and w+b modes with care; they can destroy existing files.

The specified type must be compatible with the access method you used to open the file. If the file was opened with the O_APPEND flag, the stream mode must be a, a+, ab, a+b, or ab+. To use the `fdopen()` function you need a file descriptor. To get a descriptor use the POSIX function `open()`. The O_APPEND flag is a mode for `open()`. Modes for `open()` are defined in QSYSINC/H/FCNTL. For further information see the APIs topic in the Information Center.

The keyword parameters allowed for `fdopen()` are the same as those documented in “`fopen() — Open Files`” on page 134 that are for the integrated file system.

If `fdopen()` returns NULL, use `close()` to close the file. If `fdopen()` is successful, you must use `fclose()` to close the stream and file.

Return Value
The `fdopen()` function returns a pointer to a file structure that can be used to access the open file. A NULL pointer return value indicates an error.

Example
This example opens the file sample.dat and associates a stream with the file using `fdopen()`. It then reads from the stream into the buffer.
int main(void)
{
    long length;
    int fh;
    char buffer[20];
    FILE *fp;

    printf("\nCreating sample.dat.\n");
    if ((fp = fopen("/sample.dat", "w")) == NULL) {
        perror(" File was not created: ");
        exit(1);
    }
    fputs("Sample Program", fp);
    fclose(fp);

    memset(buffer, '\0', 20); /* Initialize buffer*/
    if (-1 == (fh = open("/sample.dat", O_RDWR|O_APPEND))) {
        perror("Unable to open sample.dat");
        exit(1);
    }
    if (NULL == (fp = fdopen(fh, "r"))) {
        perror("fdopen failed");
        close(fh);
        exit(1);
    }
    if (14 != fread(buffer, 1, 14, fp)) {
        perror("fread failed");
        fclose(fp);
        exit(1);
    }
    printf("Successfully read from the stream the following:\n%s.\n", buffer);
    fclose(fp);
    return 1;

    /*****************************************************************
    * The output should be:
    *  Creating sample.dat.
    *  Successfully read from the stream the following:
    *   Sample Program.
    */
}

Related Information

- “fclose() — Close Stream” on page 116
- “fopen() — Open Files” on page 134
- “fseek() – fseeko() — Reposition File Position” on page 157
- “fsetpos() — Set File Position” on page 159
- “rewind() — Adjust Current File Position” on page 303
- “<stdio.h>” on page 13
- open API in the APIs topic in the Information Center.
- close API in the APIs topic in the Information Center.

feof() — Test End-of-File Indicator

Format

#include <stdio.h>
int feof(FILE *stream);
Language Level
ANSI

Threadsafe
Yes

Description
The `feof()` function indicates whether the end-of-file flag is set for the given stream. The end-of-file flag is set by several functions to indicate the end of the file. The end-of-file flag is cleared by calling the `rewind()`, `fsetpos()`, `fseek()`, or `clearerr()` functions for this stream.

Return Value
The `feof()` function returns a nonzero value if and only if the EOF flag is set; otherwise, it returns 0.

Example
This example scans the input stream until it reads an end-of-file character.

```c
#include <stdio.h>
#include <stdlib.h>

int main(void)
{
    char string[100];
    FILE *stream;
    memset(string, 0, sizeof(string));
    stream = fopen("qcpple/qacsrc(feof)", "r");
    fscanf(stream, "%s", string);
    while (!feof(stream))
    {
        printf("%s\n", string);
        memset(string, 0, sizeof(string));
        fscanf(stream, "%s", string);
    }
}
```

Related Information
- “`clearerr()` — Reset Error Indicators” on page 87
- “`ferror()` — Test for Read/Write Errors” on page 120
- “`fseek()` – `fseeko()` — Reposition File Position” on page 157
- “`fsetpos()` — Set File Position” on page 159
- “`perror()` — Print Error Message” on page 249
- “`rewind()` — Adjust Current File Position” on page 303
- “`<stdio.h>`” on page 13

ferror() — Test for Read/Write Errors

Format

```c
#include <stdio.h>
int ferror(FILE *stream);
```
Threadsafe
Yes

Description
The `ferror()` function tests for an error in reading from or writing to the given `stream`. If an error occurs, the error indicator for the `stream` remains set until you close `stream`, call the `rewind()` function, or call the `clearerr()` function.

Return Value
The `ferror()` function returns a nonzero value to indicate an error on the given `stream`. A return value of 0 means that no error has occurred.

Example
This example puts data out to a stream, and then checks that a write error has not occurred.

```c
#include <stdio.h>

int main(void)
{
    FILE *stream;
    char *string = "Important information";
    stream = fopen("mylib/myfile","w");
    fprintf(stream, "%s\n", string);
    if (ferror(stream))
    {
        printf("write error\n");
        clearerr(stream);
    }
    if (fclose(stream))
        perror("fclose error");
}
```

Related Information
- “clearerr() — Reset Error Indicators” on page 87
- “feof() — Test End-of-File Indicator” on page 119
- “fopen() — Open Files” on page 134
- “perror() — Print Error Message” on page 249
- “strerror() — Set Pointer to Runtime Error Message” on page 396
- “<stdio.h>” on page 13

`fflush()` — Write Buffer to File

Format

```c
#include <stdio.h>
int fflush(FILE *stream);
```

Language Level
ANSI

Threadsafe
Yes
**Description**

The `fflush()` function causes the system to empty the buffer that is associated with the specified output `stream`, if possible. If the `stream` is open for input, the `fflush()` function undoes the effect of any `ungetc()` function. The `stream` remains open after the call.

If `stream` is NULL, the system flushes all open streams.

**Note:** The system automatically deletes buffers when you close the stream, or when a program ends normally without closing the stream.

**Return Value**

The `fflush()` function returns the value 0 if it successfully deletes the buffer. It returns EOF if an error occurs.

The value of errno can be set to:

<table>
<thead>
<tr>
<th>Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>ENOTOPEN</td>
<td>The file is not open.</td>
</tr>
<tr>
<td>ERECIO</td>
<td>The file is opened for record I/O.</td>
</tr>
<tr>
<td>ESTDIN</td>
<td>stdin cannot be opened.</td>
</tr>
<tr>
<td>EIOERROR</td>
<td>A non-recoverable I/O error occurred.</td>
</tr>
<tr>
<td>EIORECERR</td>
<td>A recoverable I/O error occurred.</td>
</tr>
</tbody>
</table>

The `fflush()` function is not supported for files that are opened with type=record.

**Example**

This example deletes a stream buffer.

```c
#include <stdio.h>

int main(void)
{
    FILE *stream;
    int ch;
    unsigned int result = 0;

    stream = fopen("mylib/myfile", "r");
    while ((ch = getc(stream)) != EOF && isdigit(ch))
        result = result * 10 + ch - '0';
    if (ch != EOF)
        ungetc(ch,stream);

    fflush(stream);        /* fflush undoes the effect of ungetc function */
    printf("The result is: %d\n", result);
    if ((ch = getc(stream)) != EOF)
        printf("The character is: %c\n", ch);
}
```

**Related Information**

- “fclose() — Close Stream” on page 116
- “fopen() — Open Files” on page 134
- “setbuf() — Control Buffering” on page 364
- “ungetc() — Push Character onto Input Stream” on page 452
- “<stdio.h>” on page 13
fgetc() — Read a Character

Format

```
#include <stdio.h>
int fgetc(FILE *stream);
```

Language Level

ANSI

Threadsafe

Yes

Description

The `fgetc()` function reads a single unsigned character from the input stream at the current position and increases the associated file pointer, if any, so that it points to the next character.

Note: The `fgetc()` function is identical to `getc()`, but it is always defined as a function call; it is never replaced by a macro.

Return Value

The `fgetc()` function returns the character that is read as an integer. An EOF return value indicates an error or an end-of-file condition. Use the `feof()` or the `ferror()` function to determine whether the EOF value indicates an error or the end of the file.

The value of errno can be set to:

<table>
<thead>
<tr>
<th>Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>EBADF</td>
<td>The file pointer or descriptor is not valid.</td>
</tr>
<tr>
<td>ECONVERT</td>
<td>A conversion error occurred.</td>
</tr>
<tr>
<td>ENOTREAD</td>
<td>The file is not open for read operations.</td>
</tr>
<tr>
<td>EGETANDPUT</td>
<td>An read operation that was not allowed occurred after a write operation.</td>
</tr>
<tr>
<td>ERECIO</td>
<td>The file is open for record I/O.</td>
</tr>
<tr>
<td>ESTDIN</td>
<td>stdin cannot be opened.</td>
</tr>
<tr>
<td>EIOERROR</td>
<td>A non-recoverable I/O error occurred.</td>
</tr>
<tr>
<td>EIORECERR</td>
<td>A recoverable I/O error occurred.</td>
</tr>
</tbody>
</table>

The `fgetc()` function is not supported for files that are opened with type=record.

Example

This example gathers a line of input from a stream.
```c
#include <stdio.h>
#define MAX_LEN 80
int main(void)
{
    FILE *stream;
    char buffer[MAX_LEN + 1];
    int i, ch;
    stream = fopen("mylib/myfile","r");
    for (i = 0; (i < (sizeof(buffer)-1) &&
      ((ch = fgetc(stream)) != EOF) && (ch != '\n')); i++)
      buffer[i] = ch;
    buffer[i] = '\0';
    if (fclose(stream))
      perror("fclose error");
    printf("line: %s\n", buffer);
}
Ini: If FILENAME contains: one two three
      The output should be:
      line: one two three
**********************************************************************
```

### Related Information

- "feof() — Test End-of-File Indicator” on page 119
- “ferror() — Test for Read/Write Errors” on page 120
- “fgetwc() — Read Wide Character from Stream ” on page 127
- “fputc() — Write Character” on page 142
- “getc() – getchar() — Read a Character” on page 174
- “getwc() — Read Wide Character from Stream” on page 179
- “getwchar() — Get Wide Character from stdin” on page 181
- “<stdio.h>” on page 13

### fgetpos() — Get File Position

**Format**

```c
#include <stdio.h>
int fgetpos(FILE *stream, fpos_t *pos);
```

**Language Level**

ANSI

**Threadsafe**

Yes

**Description**

The `fgetpos()` function stores the current position of the file pointer that is 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`. 
Return Value
The `fgetpos()` function returns 0 if successful; on error, it returns nonzero and sets errno to a nonzero value.

The value of errno can be set to:

<table>
<thead>
<tr>
<th>Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>EBADF</td>
<td>The file pointer or descriptor is not valid.</td>
</tr>
<tr>
<td>EBADSEEK</td>
<td>Bad offset for a seek operation.</td>
</tr>
<tr>
<td>ENODEV</td>
<td>Operation was attempted on a wrong device.</td>
</tr>
<tr>
<td>ENOTOPEN</td>
<td>The file is not open.</td>
</tr>
<tr>
<td>ERECIO</td>
<td>The file is open for record I/O.</td>
</tr>
<tr>
<td>ESTDERR</td>
<td><code>stderr</code> cannot be opened.</td>
</tr>
<tr>
<td>ESTDIN</td>
<td><code>stdin</code> cannot be opened.</td>
</tr>
<tr>
<td>ESTDOUT</td>
<td><code>stdout</code> cannot be opened.</td>
</tr>
<tr>
<td>EIOERROR</td>
<td>A non-recoverable I/O error occurred.</td>
</tr>
<tr>
<td>EIORECERR</td>
<td>A recoverable I/O error occurred.</td>
</tr>
</tbody>
</table>

The `fgetpos()` function is not supported for files that are opened with type=record.

Example
This example opens the file `myfile` for reading and stores the current file pointer position into the variable pos.

```c
#include <stdio.h>

FILE *stream;

int main(void)
{
    int retcode;
    fpos_t pos;
    stream = fopen("mylib/myfile", "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
- “fseek() – fseeko() — Reposition File Position” on page 157
- “fsetpos() — Set File Position” on page 159
- “ftell() – ftello() — Get Current Position” on page 161
fgets() — Read a String

Format

```c
#include <stdio.h>
char *fgets (char *string, int n, FILE *stream);
```

Language Level

ANSI

Threadsafe

Yes

Description

The `fgets()` function reads characters from the current `stream` position up to and including the first new-line character (`\n`), up to the end of the stream, or until the number of characters read is equal to `n-1`, 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 new-line character, if read. If `n` is equal to 1, the `string` is empty.

Return Value

The `fgets()` function returns a pointer to the `string` buffer if successful. A NULL return value indicates an error or an end-of-file condition. Use the `feof()` or `ferror()` functions to determine whether the NULL value indicates an error or the end of the file. In either case, the value of the string is unchanged.

The `fgets()` function is not supported for files that are opened with type=record.

The value of `errno` can be set to:

<table>
<thead>
<tr>
<th>Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>EBADF</td>
<td>The file pointer or descriptor is not valid.</td>
</tr>
<tr>
<td>ECONVERT</td>
<td>A conversion error occurred.</td>
</tr>
<tr>
<td>ENOTREAD</td>
<td>The file is not open for read operations.</td>
</tr>
<tr>
<td>EGETANDPUT</td>
<td>An read operation that was not allowed occurred after a write operation.</td>
</tr>
<tr>
<td>ERECIO</td>
<td>The file is open for record I/O.</td>
</tr>
<tr>
<td>ESTDIN</td>
<td>stdin cannot be opened.</td>
</tr>
<tr>
<td>EIOERROR</td>
<td>A non-recoverable I/O error occurred.</td>
</tr>
<tr>
<td>EIORECERR</td>
<td>A recoverable I/O error occurred.</td>
</tr>
</tbody>
</table>
Example

This example gets a line of input from a data stream. The example reads no more than MAX_LEN - 1 characters, or up to a new-line character from the stream.

```c
#include <stdio.h>
#define MAX_LEN 100
int main(void)
{
  FILE *stream;
  char line[MAX_LEN], *result;
  stream = fopen("mylib/myfile","rb");
  if ((result = fgets(line,MAX_LEN,stream)) != NULL)
    printf("The string is %s\n", result);
  if (fclose(stream))
    perror("fclose error");
}
```

Related Information

- “feof() — Test End-of-File Indicator” on page 119
- “ferror() — Test for Read/Write Errors” on page 120
- “fgetws() — Read Wide-Character String from Stream ” on page 129
- “fputs() — Write String” on page 145
- “gets() — Read a Line” on page 178
- “puts() — Write a String” on page 265
- “<stdio.h>” on page 13

fgetwc() — Read Wide Character from Stream

Format

```c
#include <wchar.h>
#include <stdio.h>
wint_t fgetwc(FILE *stream);
```

Language Level

ANSI

Threadsafe

Yes

Locale Sensitive

The behavior of this function might be affected by the LC_CTYPE category of the current locale. The behavior might also be affected by the LC_UNI_CTYPE category of the current locale if LOCALETYPE(*LOCALEUCS2) or LOCALETYPE(*LOCALEUTF) is specified on the compilation command. This function is not available when LOCALETYPE(*CLD) is specified on the compilation command. For more information, see “Understanding CCSIDs and Locales” on page 565.

Integrated File System Interface

This function is not available when SYSIFCOPT(*NOIFSIO) is specified on the compilation command.
Wide Character Function

See “Wide Characters” on page 568 for more information.

Description

The `fgetwc()` reads 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).

Using non-wide-character functions with `fgetwc()` on the same stream results in undefined behavior. After calling `fgetwc()`, flush the buffer or reposition the stream pointer before calling a write function for the stream, unless EOF has been reached. After a write operation on the stream, flush the buffer or reposition the stream pointer before calling `fgetwc()`.

Note: If the current locale is changed between subsequent read operations on the same stream, undefined results can occur.

Return Value

The `fgetwc()` function 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 the `fgetwc()` function returns WEOF. If an encoding error occurs (an error converting the multibyte character into a wide character), the `fgetwc()` function sets errno to EILSEQ and returns WEOF.

Use the `ferror()` and `feof()` functions to distinguish between a read error and an EOF. 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.

The value of errno can be set to:

<table>
<thead>
<tr>
<th>Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>EBADF</td>
<td>The file pointer or descriptor is not valid.</td>
</tr>
<tr>
<td>ENOTREAD</td>
<td>The file is not open for read operations.</td>
</tr>
<tr>
<td>EGETANDPUT</td>
<td>An read operation that was not allowed occurred after a write operation.</td>
</tr>
<tr>
<td>ERECIO</td>
<td>The file is open for record I/O.</td>
</tr>
<tr>
<td>ESTDIN</td>
<td>stdin cannot be opened.</td>
</tr>
<tr>
<td>EIOERROR</td>
<td>A non-recoverable I/O error occurred.</td>
</tr>
<tr>
<td>EIORECERR</td>
<td>A recoverable I/O error occurred.</td>
</tr>
<tr>
<td>EILSEQ</td>
<td>An invalid multibyte character sequence was encountered.</td>
</tr>
<tr>
<td>ECONVERT</td>
<td>A conversion error occurred.</td>
</tr>
</tbody>
</table>

Example

This example opens a file, reads in each wide character, and prints out the characters.
```c
#include <stdio.h>
#include <stdlib.h>
#include <wchar.h>
#include <errno.h>

int main(void)
{
    FILE   *stream;
    wchar_t wc;

    if (NULL == (stream = fopen("fgetwc.dat", "r"))) {
        printf("Unable to open: \"fgetwc.dat\"\n");
        exit(1);
    }

    errno = 0;
    while (WEOF != (wc = fgetwc(stream)))
        printf("wc = %lc\n", wc);

    if (EILSEQ == errno) {
        printf("An invalid wide character was encountered.\n");
        exit(1);
    }
    fclose(stream);
    return 0;
}
```

**Related Information**

- “fgetc() — Read a Character” on page 123
- “fputwc() — Write Wide Character” on page 146
- “fgetws() — Read Wide-Character String from Stream” on page 129
- “getc() – getchar() — Read a Character” on page 174
- “getwc() — Read Wide Character from Stream” on page 179
- “getwchar() — Get Wide Character from stdin” on page 181
- “<stdio.h>” on page 13
- “<wchar.h>” on page 16

### fgetws() — Read Wide-Character String from Stream

**Format**

```c
#include <wchar.h>
#include <stdio.h>
wchar_t *fgetws(wchar_t *wcs, int n, FILE *stream);
```

**Language Level**

ANSI

**Threadsafe**

Yes

**Locale Sensitive**

The behavior of this function might be affected by the LC_CTYPE category of the current locale. The behavior might also be affected by the LC_UNI_CTYPE category of the current locale if LOCALETYPE(*LOCALEUCS2) or LOCALETYPE(*LOCALEUTF) is specified on the compilation command. This function is not available when LOCALETYPE(*CLD) is specified on the compilation command. For more information, see “Understanding CCSIDs and Locales” on page 565.
Integrated File System Interface
This function is not available when SYSIFCOPT(*NOIFSIO) is specified on the compilation command.

Wide Character Function
See “Wide Characters” on page 568 for more information.

Description
The `fgetws()` function reads at most one less than the number of wide characters specified by `n` from the stream pointed to by `stream`. The `fgetws()` function stops reading characters after WEOF, or after it reads a new-line wide character (which is retained). It adds a null wide character immediately after the last wide character read into the array. The `fgetws()` function advances the file position unless there is an error. If an error occurs, the file position is undefined.

Using non-wide-character functions with the `fgetws()` function on the same stream results in undefined behavior. After calling the `fgetws()` function, flush the buffer or reposition the stream pointer before calling a write function for the stream, unless WEOF has been reached. After a write operation on the stream, flush the buffer or reposition the stream pointer before calling the `fgetws()` function.

**Note:** If the current locale is changed between subsequent read operations on the same stream, undefined results can occur.

Return Value
If successful, the `fgetws()` function returns a pointer to the wide-character string `wcs`. If WEOF is encountered before any wide characters have been read into `wcs`, the contents of `wcs` remain unchanged and the `fgetws()` function returns a null pointer. If WEOF is reached after data has already been read into the string buffer, the `fgetws()` function returns a pointer to the string buffer to indicate success. A subsequent call would return NULL because WEOF would be reached without any data being read.

If a read error occurs, the contents of `wcs` are indeterminate, and the `fgetws()` function returns NULL. If an encoding error occurs (in converting a wide character to a multibyte character), the `fgetws()` function sets `errno` to EILSEQ and returns NULL.

If `n` equals 1, the `wcs` buffer has only room for the ending null character, and nothing is read from the stream. (Such an operation is still considered a read operation, so it cannot immediately follow a write operation unless the buffer is flushed or the stream pointer repositioned first.) If `n` is greater than 1, the `fgetws()` function fails only if an I/O error occurs, or if WEOF is reached before data is read from the stream.

Use the `ferror()` and `feof()` functions to distinguish between a read error and a WEOF. A WEOF error 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 WEOF indicator.

For information about `errno` values for `fgetws()`, see “fgetwc() — Read Wide Character from Stream” on page 127.

Example
This example opens a file, reads in the file contents, then prints the file contents.
#include <errno.h>
#include <stdio.h>
#include <stdlib.h>
#include <wchar.h>

int main(void)
{
    FILE    *stream;
    wchar_t  wcs[100];

    if (NULL == (stream = fopen("fgetws.dat", "r"))) {
        printf("Unable to open: \"fgetws.dat\"\n");
        exit(1);
    }
    errno = 0;
    if (NULL == fgetws(wcs, 100, stream)) {
        if (EILSEQ == errno) {
            printf("An invalid wide character was encountered.\n");
            exit(1);
        }
        else if (feof(stream))
            printf("End of file reached.\n");
        else
            perror("Read error.\n");
    }
    printf("wcs = \"%ls\"\n", wcs);
    fclose(stream);
    return 0;

    /*********************************************************************************
     * Assuming the file fgetws.dat contains:
     * This test string should not return -1
     * The output should be similar to:
     * wcs = "This test string should not return -1"
     *********************************************************************************/
}

Related Information

• “fgetc() — Read a Character” on page 123
• “fgets() — Read a String” on page 126
• “fgetwc() — Read Wide Character from Stream ” on page 127
• “fputws() — Write Wide-Character String” on page 148
• “<stdio.h>” on page 13
• “<wchar.h>” on page 16

fileno() — Determine File Handle

Format

#include <stdio.h>
int fileno(FILE *stream);

Language Level

XPG4

Threatsafe

Yes
**Integrated File System Interface**

This function is not available when SYSIFCOPT(*NOIFSIO) is specified on the compilation command.

**Description**

The `fileno()` function determines the file handle that is currently associated with `stream`.

**Return Value**

If the environment variable QIBM_USE_DESCRIPTOR_STDIO is set to Yes, the `fileno()` function returns 0 for stdin, 1 for stdout, and 2 for stderr.

With QIBM_USE_DESCRIPTOR_STDIO set to No, the ILE C session files stdin, stdout, and stderr do not have a file descriptor associated with them. The `fileno()` function will return a value of -1 in this case.

The value of `errno` can be set to EBADF.

**Example**

This example determines the file handle of the stderr data stream.

```c
#include <stdio.h>
#include <math.h>

int main (void)
{
  FILE *fp;
  int result;

  fp = fopen ("stderr","w");
  result = fileno(fp);
  printf("The file handle associated with stderr is %d.\n", result);
  return 0;
}
```

**Related Information**

- “fopen() — Open Files” on page 134
- “freopen() — Redirect Open Files” on page 153
- “<stdio.h>” on page 13

**floor() — Find Integer <=Argument**

**Format**

```c
#include <math.h>
double floor(double x);
```

**Language Level**

ANSI

**Threading-safe**

Yes
**Description**

The `floor()` function calculates the largest integer that is less than or equal to `x`.

**Return Value**

The `floor()` function returns the floating-point result as a double value.

The result of `floor()` cannot have a range error.

**Example**

This example assigns `y` the value of the largest integer less than or equal to 2.8 and `z` the value of the largest integer less than or equal to -2.8.

```c
#include <math.h>
#include <stdio.h>

int main(void)
{
    double y, z;
    y = floor(2.8);
    z = floor(-2.8);
    printf("floor( 2.8 ) = %lf\n", y);
    printf("floor(-2.8 ) = %lf\n", z);
}

/************************** Output should be similar to: ******************
floor(  2.8 ) = 2.000000
floor( -2.8 ) = -3.000000
*/
```

**Related Information**

- “ceil() — Find Integer >=Argument” on page 86
- “fmod() — Calculate Floating-Point Remainder” on page 133
- “<math.h>” on page 6

---

**fmod() — Calculate Floating-Point Remainder**

**Format**

```c
#include <math.h>
double fmod(double x, double y);
```

**Language Level**

ANSI

**Threading**

Yes

**Description**

The `fmod()` function 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`. 
Return Value
The `fmod()` function returns the floating-point remainder of `x/y`. If `y` is zero or if `x/y` causes an overflow, `fmod()` returns 0. The value of errno can be set to EDOM.

Example
This example computes `z` as the remainder of `x/y`; here, `x/y` is -3 with a remainder of -1.

```c
#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( %lf, %lf) = %lf\n", x, y, z);
}

/************** Output should be similar to: **************/
fmod( -10.000000, 3.000000) = -1.000000
*/
```

Related Information
- “ceil() — Find Integer >=Argument” on page 86
- “fabs() — Calculate Floating-Point Absolute Value” on page 115
- “floor() — Find Integer <=Argument” on page 132
- “<math.h>” on page 6

**fopen() — Open Files**

Format

```c
#include <stdio.h>
FILE *fopen(const char *filename, const char *mode);
```

Language Level
ANSI

Threading
Yes

Description
The `fopen()` function opens the file that is specified by `filename`. The `mode` parameter is a character string specifying the type of access that is requested for the file. The `mode` variable contains one positional parameter followed by optional keyword parameters.

**Note:** When the program is compiled with SYSIFCOPT(*IFSIO) or SYSIFCOPT(*IFS64IO), and `fopen()` creates a file in the integrated file system, the owner of the file, the owner’s group, and public is given read, write, and execute authority to the file.

The possible values for the positional parameters are:
Mode  
Description

r  
Open a text file for reading. The file must exist.

w  
Create a text file for writing. If the given file exists, its contents are destroyed unless it is a logical file.

a  
Open a text file in append mode for writing at the end of the file. The `fopen()` function creates the file if it does not exist and is not a logical file.

r+  
Open a text file for both reading and writing. The file must exist.

w+  
Create a text file for both reading and writing. If the given file exists, its contents are cleared unless it is a logical file.

a+  
Open a text file in append mode for reading or updating at the end of the file. The `fopen()` function creates the file if it does not exist.

rb  
Open a binary file for reading. The file must exist.

wb  
Create an empty binary file for writing. If the file exists, its contents are cleared unless it is a logical file.

ab  
Open a binary file in append mode for writing at the end of the file. The `fopen()` function 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+  
Create an empty binary file for both reading and writing. If the file exists, its contents will be cleared unless it is a logical file.

a+b or ab+  
Open a binary file in append mode for writing at the end of the file. The `fopen()` function creates the file if it does not exist.

Note:
1. The `fopen()` function is not supported for files that are opened with the attributes type=record and ab+, rb+, or wb+.
2. 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 that are organized into lines. Each line ends with a new-line character, except possibly the last line, depending on the compiler. The system can insert or convert control characters in an output text stream. The `fopen()` function mode "a" and "a+" cannot be used for the QSYS.LIB file system. There are implementation restrictions when using the QSYS.LIB file system for text files in all modes. Seeking beyond the start of files cannot be relied on to work with streams opened in text mode.

Note: When you use `fopen()` to create a file in the QSYS.LIB file system, specifying a library name of *LIBL or blank causes the file to be created in QTEMP library.

If a text file does not exist, you can create one using the following command:

`CRTSRCPF FILE(MYLIB/MYFILE) RCDLEN(LRECL) MBR(MYMBR) SYSTEM(*FILETYPE)`
Note: Data output to a text stream might not compare as equal to the same data on input. The QSYS.LIB file system treats database files as a directory of members. The database file must exist before a member can be dynamically created when using the fopen() function.

See Large file support in the Integrated file system topic in the Information Center for the current file system limit of the integrated file system. For files in the integrated file system that are larger than 2 GB, you need to allow your application programs access to 64-bit C runtime functions. You can use the following methods to allow your program access:

- Specify SYSIFCOPT(*IFS64IO) on a compilation command, which causes the native C compiler to define _IFS64_IO_. This causes the macros _LARGE_FILES and _LARGE_FILE_API to be defined.
- Define the macro _LARGE_FILES, either in the program source or by specifying DEFINE(_LARGE_FILES) on a compilation command. The existing C runtime functions and the relevant data types in the code will all be automatically mapped or redefined to their 64-bit versions.
- Define the macro _LARGE_FILE_API, either in the program source or by specifying DEFINE(_LARGE_FILE_API) on a compilation command. This makes visible the set of of new 64-bit C runtime functions and data types. The application must explicitly specify the name of the C runtime functions, both existing version and 64-bit version, to use.

The 64-bit C runtime functions include the following:

- int fgetpos64()
- FILE *fopen64()
- FILE *fopen64()
- FILE *freopen64()
- FILE *wfopen64()
- int fsetpos64(FILE *, const fpos64_t *)
- FILE *tmpfile64()
- int fseeko(FILE *, off_t, int)
- int fseeko64(FILE *, off64_t, int)
- off_t ftello(FILE *)
- and off64_t ftello64().

Binary files contain a series of characters. For binary files, the system does not translate control characters on input or output.

If a binary file does not exist, you can create one using the following command:

CRTPF FILE(MYLIB/MYFILE) RCDLEN(LRECL) MBR(MYMBR) MAXMBRS(*NOMAX) SYSTEM(*FILETYPE)

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 the fseek() function or the rewind() function, the write functions move the file pointer back to the end of the file before they carry out any 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 the fseek(), fsetpos(), rewind(), or fflush(). Output can immediately follow input if the end-of-file was detected.

Keyword parameters for non-Integrated File System

blksize=value
- Specifies the maximum length, in bytes, of a physical block of records.

lrecl=value
- Specifies the length, in bytes, for fixed-length records and the maximum length for variable-length records.

recfm=value
- value can be:
  - F
    - fixed-length, deblocked records
  - FB
    - fixed-length, blocked records
  - V
    - variable-length, deblocked records
  - VB
    - variable-length, blocked records
VBS
variable-length, blocked, spanned records for tape files

VS
variable-length, deblocked, spanned records for tape files

D
variable-length, deblocked, unspanned records for ASCII D format for tape files

DB
variable-length, blocked, unspanned records for ASCII D format for tape files

U
undefined format for tape files

FA
fixed-length that uses first character forms control data for printer files

Note: If the file is created using CTLCHAR(*FCFC), the first character form control will be used. If it is created using CTLCHAR(*NONE), the first character form control will not be used.

commit=value
value can be:

N This parameter identifies that this file is not opened under commitment control. This is the default.

Y This parameter identifies that this file is opened under commitment control.

ccsid=value
If a CCSID that is not supported by the operating system is specified, it is ignored by data management.

When LOCALETYPE(*LOCALEUTF) is specified on the compilation command, the default value is the LC_CTYPE CCSID value, which is determined by your current locale setting. See “setlocale() — Set Locale” on page 366 for further information about locale settings. When LOCALETYPE(*LOCALEUTF) is not specified on the compilation command, the default value is the job CCSID value. See “File CCSID” on page 566 for further information about file CCSID values.

arrseq=value
value can be:

N This parameter identifies that this file is processed in the way it was created. This is the default.

Y This parameter identifies that this file is processed in arrival sequence.

indicator=value
value can be:

N This parameter identifies that indicators in display, ICF, or printer files are stored in the file buffer. This is the default.

Y This parameter identifies that indicators in display, ICF, or printer files are stored in a separate indicator area, not in the file buffer. A file buffer is the area the system uses to transfer data to and from the user program and the operating system when writing and reading. You must store indicators in a separate indicator area when processing ICF files.

type=value
value can be:

memory This parameter identifies this file as a memory file that is available only from C programs. This is the default.

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, the fopen() function fails. Read and write operations are done with the fread() function and the fwrite() functions.
Keyword parameters for Integrated File System only

**type=value**

*value* can be:

- **record** The file is opened for sequential record I/O. (File has to be opened as binary stream.)

**ccsid=value**

*ccsid* is converted to a code page value. The default is to use the job CCSID value as the code page. The CCSID and codepage option cannot both be specified. The CCSID option provides compatibility with the operating system and Data management based stream I/O.

**Note:** Mixed data (the data contains both single and double-byte characters) is not supported for a file data processing mode of text. Mixed data is supported for a file processing mode of binary.

If you specify the *ccsid* keyword, you cannot specify the *o_ccsid* keyword or the codepage keyword.

Because of the possible expansion or contraction of converted data, making assumptions about data size and the current file offset is dangerous. For example, a file might have a physical size of 100 bytes, but after an application has read 100 bytes from the file, the current file offset might be only 50. In order to read the whole file, the application might have to read 200 bytes or more, depending on the CCSIDs involved. Therefore, file positioning functions, such as *ftell()*, *fseek()*, *fgetpos()*, and *fsetpos()*, might not work. These functions might fail with error ENOTSUP. Read functions also will not work if buffering is on, as it is by default. To turn buffering off, use the setvbuf function with the _IONBF keyword.

The *fopen()* function might fail with the ECONVET error when all of the following three conditions occur:

- The file data processing mode is text.
- The code page is not specified.
- The CCSID of the job is 'mixed-data' (the data contains both single-byte and double-byte characters).

**o_ccsid=value**

When LOCALETYPE(*LOCALEUTF) is specified on the compilation command, the default value is the LC_CTYPE CCSID value, which is determined by your current locale setting. See “setlocale() — Set Locale” on page 366 for further information about locale settings. When LOCALETYPE(*LOCALEUTF) is not specified on the compilation command, the default value is the job CCSID value. See “File CCSID” on page 566 for further information about file CCSID values.

This parameter is similar to the *ccsid* parameter, except that the value specified is not converted to a code page. Also, mixed data is supported. If the file is created, it is tagged with the specified CCSID. If the file already exists, data will be converted from the CCSID of the file to the specified CCSID on read operations. On write operations, the data is assumed to be in the specified CCSID, and is converted to the CCSID of the file.

Because of the possible expansion or contraction of converted data, making assumptions about data size and the current file offset is dangerous. For example, a file might have a physical size of 100 bytes, but after an application has read 100 bytes from the file, the current file offset might be only 50. In order to read the whole file, the application might have to read 200 bytes or more, depending on the CCSIDs involved. Therefore, file positioning functions such as *ftell()*, *fseek()*, *fgetpos()*, and *fsetpos()*, will not work. These functions will fail with ENOTSUP. Read functions also will not work if buffering is on, as it is by default. To turn buffering off, use the setvbuf function with the _IONBF keyword.

Example that uses *o_ccsid*

```c
/* Create a file that is tagged with CCSID 37 */
if ((fp = fopen("/MYFILE", "w, o_ccsid=37")) == NULL) {
    printf("Failed to open file with o_ccsid=37\n");
}
fclose(fp);
```

138 IBM i: ILE C/C++ Runtime Library Functions
/* Now reopen the file with CCSID 13488, because your application wants to deal with the data in UNICODE */

if ((fp = fopen("/MYFILE" , "r+, o_ccsid=13488")) == NULL) {
    printf("Failed to open file with o_ccsid=13488\n");
} /* Turn buffering off because read functions do not work when buffering is on */

if (setbuf(fp, NULL, _IONBF, 0) != 0){
    printf("Unable to turn buffering off\n");
} /* Because you opened with o_ccsid = 13488, you must provide all input data as unicode. If this program is compiled with LOCALETYPE(*LOCALEUCS2), L constants will be unicode. */

funcreturn = fputws(L"ABC", fp); /* Write a unicode ABC to the file. */

if (funcreturn < 0) {
    printf("Error with 'fputws' on line %d\n", __LINE__);
} /* Because the file was tagged with CCSID 37, the unicode ABC was converted to EBCDIC ABC when it was written to the file. */

codepage=value
The code page that is specified by value is used.

If you specify the codepage keyword, you cannot specify the ccsid keyword or the o_ccsid keyword.

If the file to be opened does not exist, and the open mode specifies that the file should be created, the file is created and tagged with the calculated code page. If the file already exists, the data read from the file is converted from the file's code page to the calculated code page during the read operation. Data written to the file is assumed to be in the calculated code page and is converted to the code page of the file during the write operation.

crln=value
value can be:

Y The line terminator to be used is carriage return [CR], new line [NL] combination. When data is read, all carriage returns [CR] are stripped for string functions. When data is written to a file, carriage returns [CR] are added before each new line [NL] character. Line terminator processing only occurs when a file is open with text mode. This is the default.

N The line terminator to be used is new line [NL] only.

The keyword parameters are not case sensitive and should be separated by a comma.
The fopen() function generally fails if parameters are mismatched.

Return Value
The fopen() function returns a pointer to a FILE structure type that can be used to access the open file.

Note: To use stream files (type = record) with record I/O functions, you must cast the FILE pointer to an RFILE pointer.

A NULL pointer return value indicates an error.
The value of errno can be set to:

Value
Meaning
EBADMODE
The file mode that is specified is not valid.

EBADNAME
The file name that is specified is not valid.

ECONEVRT
Conversion error.
ENOENT
   No file or library.

ENOMEM
   Storage allocation request failed.

ENOTOPEN
   The file is not open.

EIOERROR
   A non-recoverable I/O error occurred.

EIORECERR
   A recoverable I/O error occurred.

ESCANFAILURE
   The file was marked with a scan failure.

If the mode string passed to fopen() is correct, fopen() will not set errno to EBADMODE, regardless of the file type.

If the mode string that is passed to fopen() is not valid, fopen() will set errno to EBADMODE, regardless of the file type.

If the mode string passed to fopen() is correct, but is invalid to that specific type of file, fopen() will set errno to ENOTOPEN, EIOERROR, or EIORECERR, regardless of the file type.

Example
This example attempts to open a file for reading.

```c
#include <stdio.h>
#define  MAX_LEN  60

int main(void)
{
   FILE *stream;
   fpos_t pos;
   char line1[MAX_LEN];
   char line2[MAX_LEN];
   char *result;
   char ch;
   int num;
   /* The following call opens a text file for reading. */
   if ((stream = fopen("mylib/myfile", "r")) == NULL)
      printf("Could not open data file\n");
   else if ((result = fgets(line1,MAX_LEN,stream)) != NULL)
   {
      printf("The string read from myfile: %s\n", result);
      fclose(stream);
   }

   /* The following call opens a fixed record length file */
   /* for reading and writing. */
   if ((stream = fopen("mylib/myfile2", "rb+, lrecl=80, \\n      blksize=240, recfm=f") ) == NULL)
      printf("Could not open data file\n");
   else {
      fgetpos(stream, &pos);
      if (!fread(line2,sizeof(line2),1,stream))
         perror("fread error");
      else printf("1st record read from myfile2: %s\n", line2);

      fsetpos(stream, &pos); /* Reset pointer to start of file */
      fputs(result, stream); /* The line read from myfile is */
      /* written to myfile2. */
      fclose(stream);
   }
}
```

Related Information
• “fclose() — Close Stream” on page 116
fprintf() — Write Formatted Data to a Stream

Format

```c
#include <stdio.h>
int fprintf(FILE *stream, const char *format-string, argument-list);
```

Language Level

ANSI

Threading Safe

Yes

Locale Sensitive

The behavior of this function might be affected by the LC_CTYPE and LC_NUMERIC categories of the current locale. The behavior might also be affected by the LC_UNI_CTYPE category of the current locale if LOCALETYPE(LC_LOCALEUCS2) or LOCALETYPE(LC_LOCALEUTF) is specified on the compilation command. For more information, see “Understanding CCSIDs and Locales” on page 565.

Description

The `fprintf()` function formats and writes a series of characters and values to the output stream. The `fprintf()` function converts each entry in argument-list, if any, and writes to the stream according to the corresponding format specification in the format-string.

The format-string has the same form and function as the format-string argument for the `printf()` function.

Return Value

The `fprintf()` function returns the number of bytes that are printed or a negative value if an output error occurs.

For information about errno values for `fprintf()`, see “printf() — Print Formatted Characters” on page 251.

Example

This example sends a line of asterisks for each integer in the array count to the file myfile. The number of asterisks that are printed on each line corresponds to an integer in the array.
```c
#include <stdio.h>

int count[10] = {1, 5, 8, 3, 0, 3, 5, 6, 8, 10};

int main(void)
{
    int i, j;
    FILE *stream;

    stream = fopen("mylib/myfile", "w");
    /* Open the stream for writing */
    for (i = 0; i < sizeof(count) / sizeof(count[0]); i++)
    {
        for (j = 0; j < count[i]; j++)
            fprintf(stream,"*");
        /* Print asterisk */
        fprintf(stream,"
");
        /* Move to the next line */
    }
    fclose (stream);
}
```

Related Information

- “fscanf() — Read Formatted Data” on page 155
- “fwprintf() — Format Data as Wide Characters and Write to a Stream” on page 165
- “printf() — Print Formatted Characters” on page 251
- “printf() — Print Formatted Data to Buffer” on page 379
- “vprintf() — Print Argument Data” on page 464
- “vsprintf() — Print Argument Data to Buffer” on page 468
- “<stdio.h>” on page 13

**fputc() — Write Character**

**Format**

```c
#include <stdio.h>
int fputc(int c, FILE *stream);
```

**Language Level**

ANSI

**Threadsafe**

Yes
Description
The `fputc()` function converts `c` to an unsigned char and then writes `c` to the output stream at the current position and advances the file position appropriately. If the stream is opened with one of the append modes, the character is appended to the end of the stream.

The `fputc()` function is identical to `putc()`; it always is defined as a function call; it is never replaced by a macro.

Return Value
The `fputc()` function returns the character that is written. A return value of EOF indicates an error.

The value of errno can be set to:

<table>
<thead>
<tr>
<th>Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>ECONVERT</td>
<td>A conversion error occurred.</td>
</tr>
<tr>
<td>ENOTWRITE</td>
<td>The file is not open for write operations.</td>
</tr>
<tr>
<td>EPUTANDGET</td>
<td>A write operation that was not permitted occurred after a read operation.</td>
</tr>
<tr>
<td>ERECIO</td>
<td>The file is open for record I/O.</td>
</tr>
<tr>
<td>ESTDERR</td>
<td>stderr cannot be opened.</td>
</tr>
<tr>
<td>ESTDOUT</td>
<td>stdout cannot be opened.</td>
</tr>
<tr>
<td>EIOERROR</td>
<td>A non-recoverable I/O error occurred.</td>
</tr>
<tr>
<td>EIORECERR</td>
<td>A recoverable I/O error occurred.</td>
</tr>
</tbody>
</table>

The `fputc()` function is not supported for files that are opened with type=record.

Example
This example writes the contents of buffer to a file that is called `myfile`.

Note: Because the output occurs as a side effect within the second expression of the for statement, the statement body is null.
```c
#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("mylib/myfile", "w"))!= NULL )
    {
        /* Put buffer into file */
        for ( i = 0; ( i < sizeof(buffer) ) &&
             ((ch = fputc( buffer[i], stream)) != EOF ); ++i );
        fclose( stream );
    } else
        perror( "Error opening myfile" );
}
```

**Related Information**

- “fgetc() — Read a Character” on page 123
- “putc() — putchar() — Write a Character” on page 263
- “<stdio.h>” on page 13

### _fputchar() — Write Character

**Format**

```c
#include <stdio.h>
int _fputchar(int c);
```

**Language Level**

Extension

**Threading**

Yes

**Description**

_fputchar writes the single character c to the stdout stream at the current position. It is equivalent to the following fputc call:

```c
fputc(c, stdout);
```

For portability, use the ANSI/ISO fputc function instead of _fputchar.

**Return Value**

_fputchar returns the character written. A return value of EOF indicates that a write error has occurred. Use _ferror and _feof to tell whether this is an error condition or the end of the file.

For information about errno values for _fputchar, see “fputc() — Write Character” on page 142.

**Example**

This example writes the contents of buffer to stdout:
```c
#include <stdio.h>
int main(void)
{
    char buffer[80];
    int i, ch = 1;
    for (i = 0; i < 80; i++)
        buffer[i] = 'c';
    for (i = 0; (i < 80) && (ch != EOF); i++)
        ch = _fputchar(buffer[i]);
    printf("\n");
    return 0;
}
```

The output should be similar to:

```
ccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccc
```

Related Information
- “getc() – getchar() — Read a Character” on page 174
- “fputc() — Write Character” on page 142
- “putc() – putchar() — Write a Character” on page 263
- “<stdio.h>” on page 13

### fputs() — Write String

**Format**

```
#include <stdio.h>
int fputs(const char *string, FILE *stream);
```

**Language Level**

ANSI

**Threadsafe**

Yes

**Description**

The fputs() function copies string to the output stream at the current position. It does not copy the null character (\0) at the end of the string.

**Return Value**

The fputs() function returns EOF if an error occurs; otherwise, it returns a non-negative value.

The fputs() function is not supported for files that are opened with type=record.

For information about errno values for fputs(), see “fputc() — Write Character” on page 142.

**Example**

This example writes a string to a stream.
```c
#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("mylib/myfile", "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 */
        {
            perror( "fputs failed" );
        }
    }
    else   /* fputs failed */
    {
        perror( "Error opening myfile" );
    }
}
```

Related Information

- “fgets() — Read a String” on page 126
- “fputws() — Write Wide-Character String” on page 148
- “gets() — Read a Line” on page 178
- “puts() — Write a String” on page 265
- “<stdio.h>” on page 13

### fputwc() — Write Wide Character

**Format**

```c
#include <wchar.h>
#include <stdio.h>
wint_t fputwc(wint_t wc, FILE *stream);
```

**Language Level**

ANSI

**Threadsafe**

Yes

**Locale Sensitive**

The behavior of this function might be affected by the LC_CTYPE category of the current locale. It might also be affected by the LC_UNI_CTYPE category of the current locale if LOCALETYPE(*LOCALEUCS2) or LOCALETYPE(*LOCALEUTF) is specified on the compilation command. This function is not available when LOCALETYPE(*CLD) is specified on the compilation command. For more information, see “Understanding CCSIDs and Locales” on page 565.

**Integrated File System Interface**

This function is not available when SYSIFCOPT(*NOIFSIO) is specified on the compilation command.
Wide Character Function
See “Wide Characters” on page 568 for more information.

Description
The fputwc() function writes the wide character wc to the output stream pointed to by stream at the current position. It also advances the file position indicator appropriately. If the file cannot support positioning requests, or if the stream was opened with append mode, the character is appended to the stream.

Using non-wide-character functions with the fputwc() function on the same stream will result in undefined behavior. After calling the fputwc() function, delete the buffer or reposition the stream pointer before calling a read function for the stream. After reading from the stream, delete the buffer or reposition the stream pointer before calling the fputwc() function, unless EOF has been reached.

Note: If the current locale is changed between subsequent operations on the same stream, undefined results can occur.

Return Value
The fputwc() function returns the wide character that is written. If a write error occurs, the error indicator for the stream is set, and the fputwc() function returns WEOF. If an encoding error occurs during conversion from wide character to a multibyte character, fputwc() sets errno to EILSEQ and returns WEOF.

For information about errno values for putwc(), see “fputc() — Write Character” on page 142.

Example
This example opens a file and uses the fputwc() function to write wide characters to the file.

```c
#include <stdio.h>
#include <stdlib.h>
#include <wchar.h>
#include <errno.h>

int main(void)
{
    FILE    *stream;
    wchar_t *wcs = L"A character string.";
    int     i;
    if (NULL == (stream = fopen("fputwc.out", "w")))
    {
        printf("Unable to open: \"fputwc.out\".\nmarkt; exit(1);
    }
    for (i = 0; wcs[i] != L'\0'; i++) {
        errno = 0;
        if (WEOF == fputwc(wcs[i], stream)) {
            printf("Unable to fputwc() the wide character.\nmarkt; wcs[\%d] = 0x%.4lx\n" , i, wcs[i]);
            if (EILSEQ == errno)
                printf("An invalid wide character was encountered.\nmarkt; exit(1);
        }
    }
    fclose(stream);
    return 0;

    /************ *************************************************/
    The output file fputwc.out should contain:
    A character string.
    **************************************************
Related Information

- “fgetwc() — Read Wide Character from Stream” on page 127
- “fputc() — Write Character” on page 142
- “fputwc() — Write Wide Character” on page 146
- “putc() – putchar() — Write a Character” on page 263
- “putwchar() — Write Wide Character to stdout” on page 268
- “putwc() — Write Wide Character” on page 266
- “<stdio.h>” on page 13
- “<wchar.h>” on page 16

**fputws() — Write Wide-Character String**

**Format**

```c
#include <wchar.h>
#include <stdio.h>
int fputws(const wchar_t *wcs, FILE *stream);
```

**Language Level**

XPG4

**Threading**

Yes

**Locale Sensitive**

The behavior of this function might be affected by the LC_CTYPE category of the current locale. It might also be affected by the LC_UNI_CTYPE category of the current locale if LOCALETYPE(*LOCALEUCS2) or LOCALETYPE(*LOCALEUTF) is specified on the compilation command. This function is not available when LOCALETYPE(*CLD) is specified on the compilation command. For more information, see “Understanding CCSIDs and Locales” on page 565.

**Integrated File System Interface**

This function is not available when SYSIFCOPT(*NOIFSIO) is specified on the compilation command.

**Wide Character Function**

See “Wide Characters” on page 568 for more information.

**Description**

The `fputws()` function writes the wide-character string `wcs` to a `stream`. It does not write the ending null wide characters.

Using non-wide-character functions with the `fputws()` function on the same stream will result in undefined behavior. After calling the `fputws()` function, flush the buffer or reposition the stream pointer before calling a read function for the stream. After a read operation, flush the buffer or reposition the stream pointer before calling the `fputws()` function, unless EOF has been reached.

**Note:** If the current locale is changed between subsequent operations on the same stream, undefined results can occur.
Return Value

The `fputws()` function returns a non-negative value if successful. If a write error occurs, the error indicator for the stream is set, and the `fputws()` function returns -1. If an encoding error occurs when converting the wide characters to multibyte characters, the `fputws()` function sets `errno` to `EILSEQ` and returns -1.

For information about `errno` values for `fputws()`, see "fputc() — Write Character" on page 142.

Example

This example opens a file and writes a wide-character string to the file using the `fgetws()` function.

```c
#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 return -1";
    if (NULL == (stream = fopen("fputws.out", "w"))) {
        printf("Unable to open: \"fputws.out\".\n"); exit(1);
    }
    errno = 0;
    if (EOF == fputws(wcs, stream)) {
        printf("Unable to complete fputws() function.\n");
        if (EILSEQ == errno)
            printf("An invalid wide character was encountered.\n");
        exit(1);
    }
    fclose(stream);
    return 0;

    //%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
    //The output file fputws.out should contain:
    //This test string should not return -1
    //%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
}
```

Related Information

- "fgetws() — Read Wide-Character String from Stream " on page 129
- “fputs() — Write String” on page 145
- “fputwc() — Write Wide Character” on page 146
- “puts() — Write a String” on page 265
- “<stdio.h>” on page 13
- “<wchar.h>” on page 16

fread() — Read Items

Format

```c
#include <stdio.h>
size_t fread(void *buffer, size_t size, size_t count, FILE *stream);
```

Language Level

ANSI
Threadsafe

Yes

Description

The `fread()` function reads up to `count` items of `size` length from the input `stream` and stores them in the given `buffer`. The position in the file increases by the number of bytes read.

Return Value

The `fread()` function returns the number of full items successfully read, which can be less than `count` if an error occurs, or if the end-of-file is met before reaching `count`. If `size` or `count` is 0, the `fread()` function returns zero, and the contents of the array and the state of the stream remain unchanged.

The value of `errno` can be set to:

<table>
<thead>
<tr>
<th>Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>EGETANDPUT</td>
<td>A read operation that was not permitted occurred after a write operation.</td>
</tr>
<tr>
<td>ENOREC</td>
<td>Record is not found.</td>
</tr>
<tr>
<td>ENOTREAD</td>
<td>The file is not open for read operations.</td>
</tr>
<tr>
<td>ERECIO</td>
<td>The file is open for record I/O.</td>
</tr>
<tr>
<td>ESTDIN</td>
<td>stdin cannot be opened.</td>
</tr>
<tr>
<td>ETRUNC</td>
<td>Truncation occurred on the operation.</td>
</tr>
<tr>
<td>EIOERROR</td>
<td>A non-recoverable I/O error occurred.</td>
</tr>
<tr>
<td>EIORECERR</td>
<td>A recoverable I/O error occurred.</td>
</tr>
</tbody>
</table>

Use the `ferror()` and `feof()` functions to distinguish between a read error and an end-of-file.

When using `fread()` for record input, set `size` to 1 and `count` to the maximum expected length of the record, to obtain the number of bytes. If you do not know the record length, you should set `size` to 1 and `count` to a large value. You can read only one record at a time when using record I/O.

Example

This example attempts to read `NUM_ALPHA` characters from the file `myfile`. 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];

    if (( stream = fopen("mylib/myfile", "r"))!= NULL )
    {
        memset(buffer, 0, sizeof(buffer));
        num = fread( buffer, sizeof( char ), NUM_ALPHA, stream );
        if ( num ) {  /* fread success */
            printf( "Number of characters has been read = %i\n", num );
            printf( "buffer = %s\n", buffer );
            fclose( stream );
        }
        else {  /* fread failed */
            if ( ferror(stream) )    /* possibility 1 */
                perror( "Error reading myfile" );
            else if ( feof(stream) )  /* possibility 2 */
                perror( "EOF found " );
        }
    }
    else
        perror( "Error opening myfile" );
}

Related Information

- “feof() — Test End-of-File Indicator” on page 119
- “ferror() — Test for Read/Write Errors” on page 120
- “fopen() — Open Files” on page 134
- “fwrite() — Write Items” on page 168
- “<stdio.h>” on page 13

free() — Release Storage Blocks

Format

```
#include <stdlib.h>
void free(void *ptr);
```

Language Level

ANSI

Threading

Yes

Description

The free() function frees a block of storage. The ptr argument points to a block that is previously reserved with a call to the calloc(), malloc(), realloc(), _C_TS_calloc(), _C_TS_malloc(), _C_TS_realloc(), or _C_TS_malloc64() functions. The number of bytes freed is the number of bytes specified when you reserved (or reallocated, in the case of the realloc() function) the block of storage. If ptr is NULL, free() simply returns.

Note:
1. All heap storage is associated with the activation group of the calling function. As such, storage should be allocated, deallocated, and reallocated within the same activation group. It is not valid to allocate heap storage within one activation group and deallocate or reallocate that storage from a different activation group. For more information about activation groups, see the ILE Concepts manual.

2. Attempting to free a block of storage not allocated with calloc(), malloc(), or realloc() (or previously freed storage) can affect the subsequent reserving of storage and lead to undefined results. Storage that is allocated with the ILE bindable API CEEGTST can be freed with free().

To use teraspace storage instead of single-level store storage without changing the C source code, specify the TERASPACE(*YES *TSIFC) parameter on the compiler command. This maps the free() library function to _C_TS_free(), its teraspace storage counterpart.

**Note:** If a C2M1211 or C2M1212 message is generated from the free() function, refer to “Diagnosing C2M1211/C2M1212 Message Problems” on page 587 for more information.

**Return Value**

There is no return value.

**Example**

This example uses the calloc() function to allocate storage for x array elements, and then calls the free() function to free them.

```c
#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 = calloc( num, sizeof( long ))) != NULL )
    {
        for ( i = 0; i < num; ++i )           /* put values in array */
            *index++ = i;                      /* using pointer notation */
        free( array );                        /* deallocates array */
    }
    else /* Out of storage */
    {
        perror( "Error: out of storage" );
        abort();
    }
}
```

**Related Information**

- “calloc() — Reserve and Initialize Storage” on page 80
- “_C_Quickpool_Debug() — Modify Quick Pool Memory Manager Characteristics” on page 91
- “_C_Quickpool_Init() — Initialize Quick Pool Memory Manager” on page 93
- “_C_Quickpool_Report() — Generate Quick Pool Memory Manager Report” on page 95
- “Heap Memory” on page 578
- “malloc() — Reserve Storage Block” on page 217
- “realloc() — Change Reserved Storage Block Size” on page 291
- “<stdlib.h>” on page 15
freopen() — Redirect Open Files

Format

```c
#include <stdio.h>
FILE *freopen(const char *filename, const char *mode, FILE *stream);
```

Language Level

ANSI

Threadsafe

Yes

Description

The `freopen()` function closes the file that is currently associated with `stream` and reassigns `stream` to the file that is specified by `filename`. The `freopen()` function opens the new file associated with `stream` with the given `mode`, which is a character string specifying the type of access requested for the file. You can also use the `freopen()` function to redirect the standard stream files `stdin`, `stdout`, and `stderr` to files that you specify.

For database files, if `filename` is an empty string, the `freopen()` function closes and reopens the stream to the new open mode, rather than reassigning it to a new file or device. You can use the `freopen()` function with no file name specified to change the mode of a standard stream from text to binary without redirecting the stream, for example:

```c
fp = freopen("", "rb", stdin);
```

You can use the same method to change the mode from binary back to text.

You cannot use the `freopen()` function with `filename` as an empty string in modules created with SYSIFCOPT(*IFSIO).

Return Value

The `freopen()` function returns a pointer to the newly opened stream. If an error occurs, the `freopen()` function closes the original file and returns a NULL pointer value.

The value of `errno` can be set to:

<table>
<thead>
<tr>
<th>Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>EBADF</td>
<td>The file pointer or descriptor is not valid.</td>
</tr>
<tr>
<td>EBADMODE</td>
<td>The file mode that is specified is not valid.</td>
</tr>
<tr>
<td>EBADNAME</td>
<td>The file name that is specified is not valid.</td>
</tr>
<tr>
<td>ENOENT</td>
<td>No file or library.</td>
</tr>
<tr>
<td>ENOTOPEN</td>
<td>The file is not open.</td>
</tr>
<tr>
<td>EIOERROR</td>
<td>A non-recoverable I/O error occurred.</td>
</tr>
</tbody>
</table>
EIORECERR
A recoverable I/O error occurred.

Example
This example closes the stream1 data stream and reassgins its stream pointer. stream1 and stream2 will have the same value, but they will not necessarily have the same value as stream.

```c
#include <stdio.h>
#define  MAX_LEN  100

int main(void)
{
    FILE *stream, *stream1, *stream2;
    char line[MAX_LEN], *result;
    int  i;

    stream = fopen("mylib/myfile","r");
    if ((result = fgets(line,MAX_LEN,stream)) != NULL)
        printf("The string is %s\n", result);
    /* Change all spaces in the line to '*'. */
    for (i=0; i<=sizeof(line); i++)
        if (line[i] == ' ')
            line[i] = '*';

    stream1 = stream;
    stream2 = freopen("", "w+", stream1);
    fputs( line, stream2 );
    fclose( stream2);
}
```

Related Information
- “fclose() — Close Stream” on page 116
- “fopen() — Open Files” on page 134
- “<stdio.h>” on page 13

frexp() — Separate Floating-Point Value

Format
```
#include <math.h>
double frexp(double x, int *expptr);
```

Language Level
ANSI

Threading Safe
Yes

Description
The frexp() function breaks down the floating-point value x into a term m for the mantissa and another term n for the exponent. It is done such that x=m*2^n, and the absolute value of m is greater than or equal to 0.5 and less than 1.0 or equal to 0. The frexp() function stores the integer exponent n at the location to which expptr points.
Return Value

The `frexp()` function returns the mantissa term \( m \). If \( x \) is 0, `frexp()` returns 0 for both the mantissa and exponent. The mantissa has the same sign as the argument \( x \). The result of the `frexp()` function cannot have a range error.

Example

This example separates the floating-point value of \( x \), 16.4, into its mantissa 0.5125, and its exponent 5. It stores the mantissa in \( y \) and the exponent in \( n \).

```c
#include <math.h>
#include <stdio.h>

int main(void)
{
    double x, m;
    int n;
    x = 16.4;
    m = frexp(x, n);
    printf("The mantissa is \%.lf and the exponent is %d\n", m, n);
}

// Output should be similar to: ***************
The mantissa is 0.512500 and the exponent is 5
```

Related Information

- “ldexp() — Multiply by a Power of Two” on page 200
- “modf() — Separate Floating-Point Value” on page 244
- “<math.h>” on page 6

fscanf() — Read Formatted Data

Format

```c
#include <stdio.h>
int fscanf (FILE *stream, const char *format-string, argument-list);
```

Language Level

ANSI

Threading

Yes

Locale Sensitive

The behavior of this function might be affected by the LC_CTYPE and LC_NUMERIC categories of the current locale. The behavior might also be affected by the LC_UNI_CTYPE category of the current locale if LOCALETYPE("LOCALEUCS2") or LOCALETYPE("LOCALEUTF") is specified on the compilation command. For more information, see “Understanding CCSIDs and Locales” on page 565.
Description
The fscanf() function reads data from the current position of the specified stream into the locations that are given by the entries in argument-list, if any. Each entry in argument-list must be a pointer to a variable with a type that corresponds to a type specifier in format-string.

The format-string controls the interpretation of the input fields and has the same form and function as the format-string argument for the scanf() function.

Return Value
The fscanf() function returns the number of fields that it successfully converted and assigned. The return value does not include fields that the fscanf() function read but did not assign.

The return value is EOF if an input failure occurs before any conversion, or the number of input items assigned if successful.

Example
This example opens the file myfile for reading and then scans this file for a string, a long integer value, a character, and a floating-point value.

```c
#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("mylib/myfile", "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);
}

/*************** If myfile contains ************************
************** expected output is: **********************
string = abcdefghijklmnopqrstuvwxyz 343.2
************** expected output is: **********************
```

Related Information
- “fprintf() — Write Formatted Data to a Stream” on page 141
- “fwscanf() — Read Data from Stream Using Wide Character” on page 169
- “scanf() — Read Data” on page 358
- “sscanf() — Read Data” on page 382
- “swscanf() — Read Wide Character Data” on page 437
- “wscanf() — Read Data Using Wide-Character Format String” on page 539
- “<stdio.h>” on page 13
fseek() — fseeko() — Reposition File Position

Format

```c
#include <stdio.h>

int fseek(FILE *stream, long int offset, int origin);
int fseeko(FILE *stream, off_t offset, int origin);
```

Language Level

ANSI

Threadsafe

Yes

Integrated File System Interface

The fseeko() function is not available when SYSIFCOPT(*NOIFSIO) is specified on the compilation command.

Description

The fseek() and fseeko() functions change the current file position that is associated with stream to a new location within the file. The next operation on stream takes place at the new location. On a stream open for update, the next operation can be either a reading or a writing operation.

The fseeko() function is identical to fseek() except that the offset argument is of type off_t.

The origin must be one of the following constants that are defined in <stdio.h>:

<table>
<thead>
<tr>
<th>Origin</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>SEEK_SET</td>
<td>Beginning of file</td>
</tr>
<tr>
<td>SEEK_CUR</td>
<td>Current position of file pointer</td>
</tr>
<tr>
<td>SEEK_END</td>
<td>End of file</td>
</tr>
</tbody>
</table>

For a binary stream, you can also change the position beyond the end of the file. An attempt to position before the beginning of the file causes an error. If successful, the fseek() or fseeko() function clears the end-of-file indicator, even when origin is SEEK_END, and undoes the effect of any preceding the ungetc() function on the same stream.

Note: For streams opened in text mode, the fseek() and fseeko() functions have limited use because some system translations (such as those between carriage-return-line-feed and new line) can produce unexpected results. The only fseek() and fseeko() operations that can be relied upon to work on streams opened in text mode are seeking with an offset of zero relative to any of the origin values, or seeking from the beginning of the file with an offset value returned from a call to the ftell() or ftello() functions. Calls to the ftell() and ftello() functions are subject to their restrictions.

Return Value

The fseek() or fseeko() function returns 0 if it successfully moves the pointer. A nonzero return value indicates an error. On devices that cannot seek, such as terminals and printers, the return value is nonzero.

The value of errno can be set to:
<table>
<thead>
<tr>
<th>Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>EBADF</td>
<td>The file pointer or descriptor is invalid.</td>
</tr>
<tr>
<td>EBADSEEK</td>
<td>Bad offset for a seek operation.</td>
</tr>
<tr>
<td>ENODEV</td>
<td>Operation was attempted on a wrong device.</td>
</tr>
<tr>
<td>ENOTOPEN</td>
<td>The file is not open.</td>
</tr>
<tr>
<td>ERECIO</td>
<td>The file is open for record I/O.</td>
</tr>
<tr>
<td>ESTDERR</td>
<td>stderr cannot be opened.</td>
</tr>
<tr>
<td>ESTDIN</td>
<td>stdin cannot be opened.</td>
</tr>
<tr>
<td>ESTDOUT</td>
<td>stdout cannot be opened.</td>
</tr>
<tr>
<td>EIOERROR</td>
<td>A non-recoverable I/O error occurred.</td>
</tr>
<tr>
<td>EIORECERR</td>
<td>A recoverable I/O error occurred.</td>
</tr>
</tbody>
</table>

The `fseek()` and `fseeko()` functions are not supported for files that are opened with type=record.

**Example**

This example opens a file `myfile` for reading. After performing input operations, `fseek()` moves the file pointer to the beginning of the file.

```c
#include <stdio.h>
#define MAX_LEN 10

int main(void)
{
    FILE *stream;
    char buffer[MAX_LEN + 1];
    int result;
    int i;
    char ch;

    stream = fopen("mylib/myfile", "r");
    for (i = 0; (i < (sizeof(buffer) - 1) &&
        ((ch = fgetc(stream)) != EOF) && (ch != '\n')); i++)
        buffer[i] = ch;

    result = fseek(stream, 0L, SEEK_SET); /* moves the pointer to the */
    /* beginning of the file   */
    if (result == 0)
        printf("Pointer successfully moved to the beginning of the file.\n");
    else
        printf("Failed moving pointer to the beginning of the file.\n");
}
```

**Related Information**

- “ftell() – ftello() — Get Current Position” on page 161
- “fgetpos() — Get File Position” on page 124
- “fsetpos() — Set File Position” on page 159
- “rewind() — Adjust Current File Position” on page 303
fsetpos() — Set File Position

Format

```c
#include <stdio.h>
int fsetpos(FILE *stream, const fpos_t *pos);
```

Language Level

ANSI

Threadsafe

Yes

Description

The `fsetpos()` function moves any file position that is associated with `stream` to a new location within the file according to the value pointed to by `pos`. The value of `pos` was obtained by a previous call to the `fgetpos()` library function.

If successful, `fsetpos()` clears the end-of-file indicator, and undoes the effect of any previous `ungetc()` function on the same stream.

After the `fsetpos()` call, the next operation on a stream in update mode can be input or output.

Return Value

If `fsetpos()` successfully changes the current position of the file, it returns 0. A nonzero return value indicates an error.

The value of errno can be set to:

<table>
<thead>
<tr>
<th>Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>EBADF</td>
<td>The file pointer or descriptor is invalid.</td>
</tr>
<tr>
<td>EBADPOS</td>
<td>The position that is specified is not valid.</td>
</tr>
<tr>
<td>EINVAL</td>
<td>The value specified for the argument is not correct. You might receive this errno when you compile your program with *IFSIO, and you are working with a file in the QSYS file system. For example, &quot;/qsys.lib/qtemp.lib/myfile.file/mymem.mbr&quot;.</td>
</tr>
<tr>
<td>ENODEV</td>
<td>Operation was attempted on a wrong device.</td>
</tr>
<tr>
<td>ENOPOS</td>
<td>No record at the specified position.</td>
</tr>
<tr>
<td>ERECIO</td>
<td>The file is open for record I/O.</td>
</tr>
<tr>
<td>ESTDERR</td>
<td>stderr cannot be opened.</td>
</tr>
</tbody>
</table>
ESTDIN
std in cannot be opened.

ESTDOUT
std out cannot be opened.

EIOERROR
A non-recoverable I/O error occurred.

EIORECERR
A recoverable I/O error occurred.

The `fsetpos()` function cannot be used for files that are opened with type=record. Also, the `fsetpos()` function can only support setting the position to the beginning of the file if:

- your program is compiled with "IFSIO, and
- you are working on a file in the QSYS file system.

Example
This example opens a file `mylib/myfile` for reading. After performing input operations, `fsetpos()` moves the file pointer to the beginning of the file and rereads the first byte.

```c
#include <stdio.h>
FILE *stream;
int main(void)
{
    int retcode;
    fpos_t pos;
    char ptr[20]; /* existing file 'mylib/myfile' has 20 byte records */
    int i;
    /* Open file, get position of file pointer, and read first record */
    stream = fopen("mylib/myfile", "rb");
    fgetpos(stream,&pos);
    if (!fread(ptr,sizeof(ptr),1,stream))
        perror("fread error");
    else printf("1st record: %s\n", ptr);
    /* Perform another read operation on the second record */
    /* - the value of 'pos' changes */
    if (!fread(ptr,sizeof(ptr),1,stream))
        perror("fread error");
    else printf("2nd record: %s\n", ptr);
    /* Re-set pointer to start of file and re-read first record */
    fsetpos(stream,&pos);
    if (!fread(ptr,sizeof(ptr),1,stream))
        perror("fread error");
    else printf("1st record again: %s\n", ptr);
    fclose(stream);
}
```

Related Information
- “fgetpos() — Get File Position” on page 124
- “fseek() – fseeko() — Reposition File Position” on page 157
- “ftell() – ftello() — Get Current Position” on page 161
- “rewind() — Adjust Current File Position” on page 303
- “<stdio.h>” on page 13
Format

```c
#include <stdio.h>
long int ftell(FILE *stream);
off_t ftello(FILE *stream);
```

Language Level

ANSI

Threadsafe

Yes

Integrated File System Interface

The `ftello()` function is not available when SYSIFCOPT(*NOIFSIO) is specified on the compilation command.

Description

The `ftell()` and `ftello()` functions find the current position of the file associated with `stream`. For a fixed-length binary file, the value that is returned is an offset relative to the beginning of the stream. For files in the QSYS library system, the `ftell()` and `ftello()` functions return a relative value for fixed-format binary files and an encoded value for other file types. This encoded value must be used in calls to the `fseek()` and `fseeko()` functions to positions other than the beginning of the file.

Return Value

The `ftell()` and `ftello()` functions return the current file position. On error, `ftell()` and `ftello()` return -1, cast to `long` and `off_t` respectively, and set `errno` to a nonzero value. The value of `errno` can be set to:

| Value   | Meaning                                                   |
|---------|---------------|----------------------------------------------------------|
| ENODEV  | Operation was attempted on a wrong device.                |
| ENOTOPEN| The file is not open.                                    |
| ENUMMBRS| The file is open for multi-member processing.              |
| ENUMRECS| Too many records.                                        |
| ERECIO  | The file is open for record I/O.                          |
| ESTDERR | stderr cannot be opened.                                 |
| ESTDIN  | stdin cannot be opened.                                  |
| ESTDOUT | stdout cannot be opened.                                 |
EIOERROR
A non-recoverable I/O error occurred.

EIORECERR
A recoverable I/O error occurred.

The ftell() and ftello() functions are not supported for files that are opened with type=record.

Example
This example opens the file mylib/myfile for reading. It reads enough characters to fill half of the buffer and prints out the position in the stream and the buffer.

```
#include <stdio.h>
#define NUM_ALPHA 26
#define NUM_CHAR 6

int main(void)
{
    FILE * stream;
    int i;
    char ch;
    char buffer[NUM_ALPHA];
    long position;

    if (( stream = fopen("mylib/myfile", "r")) != NULL )
    {
        /* read into buffer */
        for ( i = 0; ( i < NUM_ALPHA/2 ) && ((buffer[i] = fgetc(stream)) != EOF ); ++i )
            if (i==NUM_CHAR-1)  /* We want to be able to position the */
                /* file pointer to the character in   */
                /* position NUM_CHAR                  */
                position = ftell(stream);
        buffer[i] = '\0';
    }
    printf("Current file position is %d\n", position);
    printf("Buffer contains: %s\n", buffer);
}
```

Related Information
• “fseek() – fseeko() — Reposition File Position” on page 157
• “fgetpos() — Get File Position” on page 124
• “fopen() — Open Files” on page 134
• “fsetpos() — Set File Position” on page 159
• “<stdio.h>” on page 13

fwide() — Determine Stream Orientation

Format
```
#include <stdio.h>
#include <wchar.h>

int fwide(FILE *stream, int mode);
```

Language Level
ANSI

Threading
Yes

162 IBM i: ILE C/C++ Runtime Library Functions
Locale Sensitive
This function is not available when LOCALETYPE(*CLD) is specified on the compilation command.

Integrated File System Interface
This function is not available when SYSIFCOPT(*NOIFSIO) is specified on the compilation command.

Description
The fwide() function determines the orientation of the stream pointed to by stream. If mode is greater than 0, the fwide() function first attempts to make the stream wide oriented. If mode is less than 0, the fwide() function first attempts to make the stream byte oriented. Otherwise, mode is 0, and the fwide() function does not alter the orientation of the stream.

Note: If the orientation of the stream has already been determined, the fwide() function does not change it.

Return Value
If, after the call, the stream has wide orientation, the fwide() function returns a value greater than 0. If the stream has byte orientation, it returns a value less than 0. If the stream has no orientation, it returns 0.
Example

```c
#include <stdio.h>
#include <math.h>
#include <wchar.h>

void check_orientation(FILE *stream)
{
    int rc;
    rc = fwide(stream,0); /* check the orientation */
    if (rc<0) {
        printf("Stream has byte orientation.\n");
    } else if (rc>0) {
        printf("Stream has wide orientation.\n");
    } else {
        printf("Stream has no orientation.\n");
    } return;
}

int main(void)
{
    FILE *stream;
    /* Demonstrate that fwide can be used to set the orientation,
       but cannot change it once it has been set. */
    stream = fopen("test.dat","w");
    printf("After opening the file: ");
    check_orientation(stream);
    fwide(stream, -1); /* Make the stream byte oriented */
    printf("After fwide(stream, -1): ");
    check_orientation(stream);
    fwide(stream,  1); /* Try to make the stream wide oriented */
    printf("After fwide(stream, 1): ");
    check_orientation(stream);
    fclose(stream);  
    printf("Close the stream\n");
    /* Check that a wide character output operation sets the orientation
       as expected. */
    stream = fopen("test.dat","w");
    printf("After opening the file: ");
    check_orientation(stream);
    fwprintf(stream, L"pi = %.5f\n", 4* atan(1.0));
    printf("After fwprintf( ): ");
    check_orientation(stream);
    fclose(stream);
    return 0;
}
```

Related Information

- “fgetwc() — Read Wide Character from Stream” on page 127
- “fgetws() — Read Wide-Character String from Stream” on page 129
- “fputwc() — Write Wide Character” on page 146
- “fputws() — Write Wide-Character String” on page 148
- “<stdio.h>” on page 13
- “<wchar.h>” on page 16
fwprintf() — Format Data as Wide Characters and Write to a Stream

Format

```c
#include <stdio.h>
#include <wchar.h>
int fwprintf(FILE *stream, const wchar_t *format, argument-list);
```

Language Level

ANSI

Threadsafe

Yes

Locale Sensitive

The behavior of this function might be affected by the LC_CTYPE and LC_NUMERIC categories of the current locale, and might also be affected by the LC_UNI_CTYPE and LC_UNI_NUMERIC categories of the current locale if LOCALETYPE(*LOCALEUCS2) or LOCALETYPE(*LOCALEUTF) is specified on the compilation command. This function is not available when LOCALETYPE(*CLD) is specified on the compilation command. For more information, see “Understanding CCSIDs and Locales” on page 565.

Integrated File System Interface

This function is not available when SYSIFCOPT(*NOIFSIO) is specified on the compilation command.

Wide Character Function

See “Wide Characters” on page 568 for more information.

Description

The `fwprintf()` function writes output to the stream pointed to by `stream`, under control of the wide string pointed to by `format`. The format string specifies how subsequent arguments are converted for output.

The `fwprintf()` function converts each entry in `argument-list` according to the corresponding wide-character format specifier in `format`.

If insufficient arguments exist for the format, the behavior is undefined. If the format is exhausted while arguments remain, the `fwprintf()` function evaluates the excess arguments, but otherwise ignores them. The `fwprintf()` function returns when it encounters the end of the format string.

The format comprises zero or more directives: ordinary wide characters (not %) and conversion specifications. Conversion specifications are processed as if they were replaced in the format string by wide-character strings. The wide-character strings are the result of fetching zero or more subsequent arguments and then converting them, if applicable, according to the corresponding conversion specifier. The `fwprintf()` function then writes the expanded wide-character format string to the output stream.

The format for the `fwprintf()` function has the same form and function as the format string for `printf()`, with the following exceptions:

- %c (without an l prefix) converts an integer argument to wchar_t, as if by calling the `btowc()` function.
- %s (without an l prefix) converts an array of multibyte characters to an array of wchar_t, as if by calling the `mbtowc()` function. The array is written up to, but not including, the terminating null character, unless the precision specifies a shorter output.
- %ls and %S write an array of wchar_t. The array is written up to, but not including, the ending null character, unless the precision specifies a shorter output.
• Any width or precision specified for %c, %s, %ls, and %S indicates the number of characters rather than the number of bytes.

If a conversion specification is invalid, the behavior is undefined.

If any argument is, or points to, a union or an aggregate (except for an array of char type using %s conversion, an array of wchar_t type using %ls conversion, or a pointer using %p conversion), the behavior is undefined.

In no case does a nonexistent, or small field width, cause truncation of a field; if the conversion result is wider than the field width, the field is expanded to contain the conversion result.

**Note:** When you write wide characters, the file should be opened in binary mode, or opened with the **o_ccsid** or **codepage** parameters. This ensures that no conversions occur on the wide characters.

**Return Value**

The `fwprintf()` function returns the number of wide characters transmitted. If an output error occurred, it returns a negative value.

**Example**

```c
#include <stdio.h>
#include <wchar.h>
#include <locale.h>

int count[10] = {1, 5, 8, 3, 0, 3, 5, 6, 8, 10};

int main(void)
{
    int i, j;
    FILE *stream; /* Open the stream for writing */
    if (NULL == (stream = fopen("/QSYS.LIB/LIB.LIB/WCHAR.FILE/WCHAR.MBR","wb")))
        perror("fopen error");
    for (i=0; i < sizeof(count) / sizeof(count[0]); i++)
    {
        for (j = 0; j < count[i]; j++)
            fwprintf(stream, L"*"); /* Print asterisk */
        fwprintf(stream, L"\n"); /* Move to the next line */
    }
    fclose (stream);
}
/* The member WCHAR of file WCHAR will contain:
 * 
 * *****
 * **********
 * ***
 * ***
 * *****
 * **********
 * **********
 */
```

166 IBM i: ILE C/C++ Runtime Library Functions
# include <stdio.h>
# include <stdlib.h>
# include <locale.h>

/* This program is compile LOCALETYPE(*LOCALEUCS2) and */
/* SYSIFCOPT(*IFSIO) */

int main(void)
{
    FILE *stream;
    wchar_t wc = 0x0058;     /* UNICODE X */
    char c1 = 'c';
    char *s1 = "123";
    wchar_t ws[4];
    setlocale(LC_ALL,
            "/QSYS.LIB/EN_US.LOCALE"); /* a CCSID 37 locale */
    ws[0] = 0x0041;        /* UNICODE A   */
    ws[1] = (wchar_t)0x0042;        /* UNICODE B   */
    ws[2] = (wchar_t)0x0043;        /* UNICODE C   */
    ws[3] = (wchar_t)0x0000;
    stream = fopen("myfile.dat", "wb+”);

    fwprintf(stream, L"%lc %ls", wc, ws);
    /* c and s take multibyte as input and produce UNICODE */
    /* In this case c1 and s1 are CCSID 37 characters based */
    /* on the setlocale above. So the characters are */
    /* converted from CCSID 37 to UNICODE and will look */
    /* like this in hex after the following fwprintf */
    /* statement: 0063002000200020003100320033 */
    /* 0063 is a UNICODE c 0031 is a UNICODE 1 and so on */
    /* Now let's try width and precision. 6ls means write */
    /* 6 wide characters so we will pad with 3 UNICODE */
    /* blanks and %.2s means write no more then 2 wide */
    /* characters. So we get an output that looks like */
    fwprintf(stream, L"%6ls%.2s", ws, s1);
}

Related Information

• “fprintf() — Write Formatted Data to a Stream” on page 141
• “printf() — Print Formatted Characters” on page 251
• “vfprintf() — Print Argument Data to Stream” on page 457
• “vprintf() — Print Argument Data” on page 464
• “btowc() — Convert Single Byte to Wide Character” on page 78
• “mbtowc() — Convert a Multibyte Character to a Wide Character (Restartable)” on page 223
• “vfwprintf() — Format Argument Data as Wide Characters and Write to a Stream ” on page 460
• “vswprintf() — Format and Write Wide Characters to Buffer” on page 471
• “wprintf() — Format Data as Wide Characters and Print” on page 538
• “<stdarg.h>” on page 11
• “<wchar.h>” on page 16
fwrite() — Write Items

Format

```c
#include <stdio.h>
size_t fwrite(const void *buffer, size_t size, size_t count,
              FILE *stream);
```

Language Level

ANSI

Threadsafe

Yes

Description

The `fwrite()` function writes up to `count` items, each of `size` bytes in length, from `buffer` to the output `stream`.

Return Value

The `fwrite()` function returns the number of full items successfully written, which can be fewer than `count` if an error occurs.

When using `fwrite()` for record output, set `size` to 1 and `count` to the length of the record to obtain the number of bytes written. You can only write one record at a time when using record I/O.

The value of `errno` can be set to:

<table>
<thead>
<tr>
<th>Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>ECONVERT</td>
<td>A conversion error occurred.</td>
</tr>
<tr>
<td>ENOTWRITE</td>
<td>The file is not open for write operations.</td>
</tr>
<tr>
<td>EPAD</td>
<td>Padding occurred on a write operation.</td>
</tr>
<tr>
<td>EPUTANDGET</td>
<td>An illegal write operation occurred after a read operation.</td>
</tr>
<tr>
<td>ESTDERR</td>
<td><code>stderr</code> cannot be opened.</td>
</tr>
<tr>
<td>ESTDIN</td>
<td><code>stdin</code> cannot be opened.</td>
</tr>
<tr>
<td>ESTDOUT</td>
<td><code>stdout</code> cannot be opened.</td>
</tr>
<tr>
<td>ETRUNC</td>
<td>Truncation occurred on I/O operation.</td>
</tr>
<tr>
<td>EIOERROR</td>
<td>A non-recoverable I/O error occurred.</td>
</tr>
<tr>
<td>EIORECERR</td>
<td>A recoverable I/O error occurred.</td>
</tr>
</tbody>
</table>
Example
This example writes NUM long integers to a stream in binary format.

```c
#include <stdio.h>
#define NUM 100

int main(void)
{
    FILE *stream;
    long list[NUM];
    int numwritten;
    int i;
    stream = fopen("MYLIB/MYFILE", "w+b");
    /* assign values to list[] */
    for (i=0; i<=NUM; i++)
        list[i]=i;
    numwritten = fwrite(list, sizeof(long), NUM, stream);
    printf("Number of items successfully written = %d\n", numwritten);
}
```

Related Information
• “fopen() — Open Files” on page 134
• “fread() — Read Items” on page 149
• “<stdio.h>” on page 13

**fwscanf() — Read Data from Stream Using Wide Character**

**Format**

```c
#include <stdio.h>
#include <wchar.h>
int fwscanf(FILE *stream, const wchar_t *format, argument-list);
```

**Language Level**

ANSI

**Threadsafe**

Yes

**Locale Sensitive**

The behavior of this function might be affected by the LCCTYPE and LCNUMERIC categories of the current locale. It might also be affected by the LCUNICTYPE and LCUNINUMERIC categories of the current locale if LOCALETYPE(*LOCALEUCS2) or LOCALETYPE(*LOCALEUTF) is specified on the compilation command. This function is not available when LOCALETYPE(*CLD) is specified on the compilation command. For more information, see “Understanding CCSIDs and Locales” on page 565.

**Integrated File System Interface**

This function is not available when SYSIFCOPT(*NOIFSIO) is specified on the compilation command.

**Wide Character Function**

See “Wide Characters” on page 568 for more information.
The `fwscanf()` function reads input from the stream pointed to by `stream`, under control of the wide string pointed to by `format`. The format string specifies the admissible input sequences and how they are to be converted for assignment. To receive the converted input, the `fwscanf()` function uses subsequent arguments as pointers to the objects.

Each argument in `argument-list` must point to a variable with a type that corresponds to a type specifier in `format`.

If insufficient arguments exist for the format, the behavior is undefined. If the format is exhausted while arguments remain, the `fwscanf()` function evaluates the excess arguments, but otherwise ignores them.

The format consists of zero or more directives: one or more white-space wide characters; an ordinary wide character (neither % nor a white-space wide character); or a conversion specification. Each conversion specification is introduced by a %.

The format has the same form and function as the format string for the `scanf()` function, with the following exceptions:

- `%c` (with no l prefix) converts one or more `wchar_t` characters (depending on precision) to multibyte characters, as if by calling `wcrtomb()`.
- `%lc` and `%C` convert one or more `wchar_t` characters (depending on precision) to an array of `wchar_t`.
- `%s` (with no l prefix) converts a sequence of non-white-space `wchar_t` characters to multibyte characters, as if by calling the `wcrtomb()` function. The array includes the ending null character.
- `%ls` and `%S` copy an array of `wchar_t`, including the ending null wide character, to an array of `wchar_t`.

If the data is from `stdin`, and `stdin` has not been overridden, the data is assumed to be in the CCSID of the job. The data is converted as required by the format specifications. If the file that is being read is not opened with file mode `rb`, then invalid conversion can occur.

If a conversion specification is invalid, the behavior is undefined. If the `fwscanf()` function encounters end-of-file during input, conversion is ended. If end-of-file occurs before the `fwscanf()` function reads any characters matching the current directive (other than leading white space, where permitted), execution of the current directive ends with an input failure. Otherwise, unless execution of the current directive terminates with a matching failure, execution of the following directive (other than `%n`, if any) ends with an input failure.

The `fwscanf()` function leaves trailing white space (including new-line wide characters) unread, unless matched by a directive. You cannot determine the success of literal matches and suppressed assignments other than through the `%n` directive.

**Return Value**

The `fwscanf()` function returns the number of input items assigned, which can be fewer than provided for, in the event of an early matching failure.

If an input failure occurs before any conversion, the `fwscanf()` function returns EOF.

**Example**

This example opens the file `myfile.dat` for input, and then scans this file for a string, a long integer value, a character, and a floating-point value.
#include <stdio.h>
#include <wchar.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");
    /* Read data from file. */
    fwscanf(stream, L"%s", &s[0]);
    fwscanf(stream, L"%ld", &l);
    fwscanf(stream, L"%c", &c);
    fwscanf(stream, L"%f", &fp);
    printf("string = %s\n", s);
    printf("long integer = %ld\n", l);
    printf("char = %c\n", c);
    printf("float = %f\n", fp);
    return 0;
}

/**
 * If myfile.dat contains:
 * abcdefghijklmnopqrstuvwxyz 343.2.
 * The output should be:
 * string = abcdefghijklmnopqrstuvwxyz
 * long integer = 343
 * char = .
 * float = 2.000000
 */

Related Information

• "fscanf() — Read Formatted Data" on page 155
• "fwprintf() — Format Data as Wide Characters and Write to a Stream" on page 165
• "scanf() — Read Data" on page 358
gamma() — Gamma Function

Format

```c
#include <math.h>
double gamma(double x);
```

Language Level

ILE C Extension

Threadsafe

Yes

Description

The `gamma()` function computes the natural logarithm of the absolute value of G(x) (ln(|G(x)|)), where

\[ G(x) = \int_{0}^{\infty} e^{-t} \times t^{x-1} \, dt \]

The argument `x` must be a positive real value.

Return Value

The `gamma()` function returns the value of ln(|G(x)|). If `x` is a negative value, errno is set to EDOM. If the result causes an overflow, `gamma()` returns HUGE_VAL and sets errno to ERANGE.

Example

This example uses `gamma()` to calculate ln(|G(x)|), where `x` = 42.

```c
#include <math.h>
#include <stdio.h>

int main(void)
{
    double x=42, g_at_x;
    g_at_x = exp(gamma(x));       /* g_at_x = 3.345253e+49 */
    printf("The value of G(%4.2lf) is %7.2e\n", x, g_at_x);
}
```

Output should be similar to:

```
The value of G(42.00) is 3.35e+49
```

Example

• “Bessel Functions” on page 75
_gcvt() — Convert Floating-Point to String

Format

```c
#include <stdlib.h>
char *gcvt(double value, int ndec, char *buffer);
```

Note: The `gcvt` function is supported only for C++, not for C.

Threadsafe

Yes

Language Level

Extension

Locale Sensitive

The behavior of this function might be affected by the LC_CTYPE and LC_NUMERIC categories of the current locale. For more information, see “Understanding CCSIDs and Locales” on page 565.

Description

`gcvt()` converts a floating-point value to a character string pointed to by `buffer`. The `buffer` should be large enough to hold the converted value and a null character (\0) that `gcvt()` automatically adds to the end of the string. There is no provision for overflow.

`gcvt()` attempts to produce `ndec` significant digits in FORTRAN F format. Failing that, it produces `ndec` significant digits in FORTRAN E format. Trailing zeros might be suppressed in the conversion if they are insignificant.

A FORTRAN F number has the following format:

```
+ digit digit . digit
```

A FORTRAN E number has the following format:

```
+ digit . digit E digit digit
```

`gcvt` also converts infinity values to the string INFINITY.

Return Value

`gcvt()` returns a pointer to the string of digits. If it cannot allocate memory to perform the conversion, `gcvt()` returns an empty string and sets `errno` to ENOMEM.
Example
This example converts the value -3.1415e3 to a character string and places it in the character array buffer1.

```c
#include <stdio.h>
#include <stdlib.h>
int main(void)
{
    char buffer1[10];
    _gcvt(-3.1415e3, 7, buffer1);
    printf("The first result is %s \n", buffer1);
    return 0;
}
```

The output should be:

```
The first result is -3141.5
```

Related Information
- “<stdlib.h>” on page 15

getc() – getchar() — Read a Character

Format

```c
#include <stdio.h>
int getc(FILE *stream);
int getchar(void);
```

Language Level
ANSI

Threading Safe
No

```
#define getc FILE *stream);
#define getchar(void);
```

Description
The getc() function 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 difference between the getc() and fgetc() functions is that getc() can be implemented so that its arguments can be evaluated multiple times. Therefore, the stream argument to getc() should not be an expression with side effects.

Return Value
The getc() and getchar() functions return the character read. A return value of EOF indicates an error or end-of-file condition. Use ferror() or feof() to determine whether an error or an end-of-file condition occurred.

The value of errno can be set to:

Value | Meaning
--- | ---
174 | IBM i: ILE C/C++ Runtime Library Functions
EBADF
The file pointer or descriptor is not valid.

ECONVERT
A conversion error occurred.

EGETANDPUT
An illegal read operation occurred after a write operation.

EIOERROR
A non-recoverable I/O error occurred.

EIORECERR
A recoverable I/O error occurred.

The `getc()` and `getchar()` functions are not supported in record mode.

Example
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.

```c
#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 !='
' ); ++i )
        buffer[i] = ch;
    buffer[i] = '\0';  /* a string should always end with '\0' ! */
    printf( "The string is %s\n", buffer );
}
```

Related Information
- “fgetc() — Read a Character” on page 123
- “fgetwc() — Read Wide Character from Stream” on page 127
- “gets() — Read a Line” on page 178
- “getwc() — Read Wide Character from Stream” on page 179
- “getwchar() — Get Wide Character from stdin” on page 181
- “putc() – putchar() — Write a Character” on page 263
- “ungetc() — Push Character onto Input Stream” on page 452
- “<stdio.h>” on page 13
getenv() — Search for Environment Variables

Format

```
#include <stdlib.h>
char *getenv(const char *varname);
```

Language Level

ANSI

Threadsafe

Yes

Job CCSID Interface

All character data sent to this function is expected to be in the CCSID of the job. All character data returned by this function is in the CCSID of the job. See “Understanding CCSIDs and Locales” on page 565 for more information.

Description

The `getenv()` function searches the list of environment variables for an entry corresponding to `varname`.

Return Value

The `getenv()` function returns a pointer to the string containing the value for the specified `varname` in the current environment. If `getenv()` cannot find the environment string, NULL is returned, and `errno` is set to indicate the error.

Example

```
#include  <stdlib.h>
#include  <stdio.h>
/* Where the environment variable 'PATH' is set to a value. */
int main(void)
{
    char *pathvar;
    pathvar = getenv("PATH");
    printf("pathvar=%s",pathvar);
}
```

Related Information

- “<stdlib.h>” on page 15
- “putenv() — Change/Add Environment Variables” on page 264
- Environment Variable APIs in the APIs topic in the Information Center.

_GetExcData() — Get Exception Data

Format

```
#include <signal.h>
void _GetExcData(_INTRPT_Hndlr_Parms_T *pazms);
```
Language Level
ILE C Extension

Threadsafe
Yes

Job CCSID Interface
All character data sent to this function is expected to be in the CCSID of the job. All character data returned by this function is in the CCSID of the job. See “Understanding CCSIDs and Locales” on page 565 for more information.

Description
The _GetExcData() function returns information about the current exception from within a C signal handler. The caller of the _GetExcData() function must allocate enough storage for a structure of type _INTRPT_Hndlr_Parms_T. If the _GetExcData() function is called from outside a signal handler, the storage pointed to by parms is not updated.

This function is not available when SYSIFCOPT(*ASYNCSIGNAL) is specified on the compilation commands. When SYSIFCOPT(*ASYNCSIGNAL) is specified, a signal handler established with the ILE C signal() function has no way to access any exception information that might have caused the signal handler to be invoked. An extended signal handler established with the sigaction() function, however, does have access to this exception information. The extended signal handler has the following function prototype:

void func( int signo, siginfo_t *info, void *context )

The exception information is appended to the siginfo_t structure, which is then passed as the second parameter to the extended signal handler.

The siginfo_t structure is defined in signal.h. The exception-related data follows the si_sigdata field in the siginfo_t structure. You can address it from the se_data field of the sigdata_t structure.

The format of the exception data appended to the siginfo_t structure is defined by the _INTRPT_Hndlr_Parms_T structure in except.h.

Return Value
There is no return value.

Example
This example shows how exceptions from MI library functions can be monitored and handled using a signal handling function. The signal handler my_signal_handler is registered before the rslvsp() function signals a 0x2201 exception. When a SIGSEGV signal is raised, the signal handler is called. If an 0x2201 exception occurred, the signal handler calls the QUSRCRTS API to create a space.
#include <signal.h>
#include <QSYSINC/MIH/RSLVSP>
#include <QSYSINC/H/QUSCRTUS>
#include <string.h>

#define CREATION_SIZE  65500

void my_signal_handler(int sig) {
    _INTRPT_Hndlr_Parms_T excp_data;
    int error_code = 0;

    /* Check the message id for exception 0x2201 */
    _GetExcData(&excp_data);

    if (!memcmp(excp_data.Msg_Id, "MCH3401", 7))
        QUSCRTUS("MYSPACE   QTEMP     ",
                  "MYSPACE   ",
                  CREATION_SIZE,
                  "\0",
                  "*ALL      ",
                  "MYSPACE example for Programmer's Reference ",
                  "*YES      ",
                  &error_code);
}

Related Information
• “signal() — Handle Interrupt Signals” on page 374
• “<except.h>” on page 2

gets() — Read a Line

Format
#include <stdio.h>
char *gets(char *buffer);

Language Level
ANSI

Threadsafe
Yes

Description
The gets() function reads a line from the standard input stream stdin and stores it in buffer. The line consists of all characters up to but not including the first new-line character (\n) or EOF. The gets() function then replaces the new-line character, if read, with a null character (\0) before returning the line.

Return Value
If successful, the gets() function returns its argument. A NULL pointer return value indicates an error, or an end-of-file condition with no characters read. Use the ferror() function or thefeof() function to determine which of these conditions occurred. If there is an error, the value that is stored in buffer is undefined. If an end-of-file condition occurs, buffer is not changed.

Example
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("Please enter a string:\n");
    if ((result = gets(line)) != NULL)
        printf("The string is: %s\n", line);
    else if (ferror(stdin))
        perror("Error");
}

Related Information
• “fgets() — Read a String” on page 126
• “fgetws() — Read Wide-Character String from Stream ” on page 129
• “feof() — Test End-of-File Indicator” on page 119
• “ferror() — Test for Read/Write Errors” on page 120
• “fputs() — Write String” on page 145
• “getc() – getchar() — Read a Character” on page 174
• “puts() — Write a String” on page 265
• “<stdio.h>” on page 13

getc() — Read Wide Character from Stream

Format

#include <stdio.h>
#include <wchar.h>

wint_t getwc(FILE *stream);

Language Level
ANSI

Threatsafe
Yes

Locale Sensitive
The behavior of this function might be affected by the LC_CTYPE category of the current locale. It might also be affected by the LC_UNI_CTYPE category of the current locale if LOCALETYPE(*LOCALEUCS2) or LOCALETYPE(*LOCALEUTF) is specified on the compilation command. This function is not available when LOCALETYPE(*CLD) is specified on the compilation command. For more information, see “Understanding CCSIDs and Locales” on page 565.

Integrated File System Interface
This function is not available when SYSIFCOPT(*NOIFSIO) is specified on the compilation command.

Wide Character Function
See “Wide Characters” on page 568 for more information.
**Description**

The `getwc()` function reads the next multibyte character from stream, converts it to a wide character, and advances the associated file position indicator for stream.

The `getwc()` function is equivalent to the `fgetwc()` function except that, if it is implemented as a macro, it can evaluate `stream` more than once. Therefore, the argument should never be an expression with side effects.

If the current locale is changed between subsequent read operations on the same stream, undefined results can occur. Using non-wide-character functions with the `getwc()` function on the same stream results in undefined behavior.

After calling the `getwc()` function, flush the buffer or reposition the stream pointer before calling a write function for the stream, unless EOF has been reached. After a write operation on the stream, flush the buffer or reposition the stream pointer before calling the `getwc()` function.

**Return Value**

The `getwc()` function returns the next wide character from the input stream, or WEOF. If an error occurs, the `getwc()` function sets the error indicator. If the `getwc()` function encounters the end-of-file, it sets the EOF indicator. If an encoding error occurs during conversion of the multibyte character, the `getwc()` function sets errno to EILSEQ.

Use the `ferror()` or `feof()` functions to determine whether an error or an EOF condition occurred. 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.

For information about errno values for `getwc()`, see “`fgetwc()` — Read Wide Character from Stream” on page 127.
Example

```c
#include <stdio.h>
#include <stdlib.h>
#include <wchar.h>
#include <errno.h>

int main(void)
{
    FILE   *stream;
    wint_t wc;
    if (NULL == (stream = fopen("getwc.dat", "r"))) {
        printf("Unable to open: \"getwc.dat\"\n");
        exit(1);
    }
    errno = 0;
    while (WEOF != (wc = getwc(stream)))
        printf("wc = %lc\n", wc);
    if (EILSEQ == errno)
        printf("An invalid wide character was encountered.\n");
    exit(1);
    fclose(stream);
    return 0;

    /********************************************************
    Assuming the file getwc.dat contains:
    Hello world!
    The output should be similar to:
    wc = H
    wc = e
    wc = l
    wc = l
    wc = o
    :
    ********************************************************/
}
```

Related Information

- “fgetwc() — Read Wide Character from Stream ” on page 127
- “getwchar() — Get Wide Character from stdin” on page 181
- “getc() – getchar() — Read a Character” on page 174
- “putwc() — Write Wide Character” on page 266
- “ungetwc() — Push Wide Character onto Input Stream ” on page 453
- “<stdio.h>” on page 13
- “<wchar.h>” on page 16

getwchar() — Get Wide Character from stdin

Format

```c
#include <wchar.h>
wint_t getwchar(void);
```

Language Level

ANSI
Threadsafe
Yes

Locale Sensitive
The behavior of this function might be affected by the LC_CTYPE category of the current locale. It might also be affected by the LC_UNI_CTYPE category of the current locale if LOCALETYPE(*LOCALEUCS2) or LOCALETYPE(*LOCALEUTF) is specified on the compilation command. This function is not available when LOCALETYPE(*CLD) is specified on the compilation command. For more information, see “Understanding CCSIDs and Locales” on page 565.

Integrated File System Interface
This function is not available when SYSIFCOPT(*NOIFSIO) is specified on the compilation command.

Wide Character Function
See “Wide Characters” on page 568 for more information.

Description
The getwchar() function reads the next multibyte character from stdin, converts it to a wide character, and advances the associated file position indicator for stdin. A call to the getwchar() function is equivalent to a call to getwc(stdin).

If the current locale is changed between subsequent read operations on the same stream, undefined results can occur. Using non-wide-character functions with the getwchar() function on stdin results in undefined behavior.

Return Value
The getwchar() function returns the next wide character from stdin or WEOF. If the getwchar() function encounters EOF, it sets the EOF indicator for the stream and returns WEOF. If a read error occurs, the error indicator for the stream is set, and the getwchar() function returns WEOF. If an encoding error occurs during the conversion of the multibyte character to a wide character, the getwchar() function sets errno to EILSEQ and returns WEOF.

Use the ferror() or feof() functions to determine whether an error or an EOF condition occurred. 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.

For information about errno values for getwchar(), see “fgetwc() — Read Wide Character from Stream” on page 127.

Example
This example uses the getwchar() to read wide characters from the keyboard, then prints the wide characters.
```c
#include <errno.h>
#include <stdio.h>
#include <stdlib.h>
#include <wchar.h>

int main(void)
{
    wint_t wc;
    errno = 0;
    while (WEOF != (wc = getwchar()))
    {       
        printf("wc = %lc\n", wc);
    
    if (EILSEQ == errno) {
        printf("An invalid wide character was encountered.\n");
        exit(1);
    } 
    return 0;
    
    /***************************************************************
     * Assuming you enter: abcde
    The output should be:
    wc = a
    wc = b
    wc = c
    wc = d
    wc = e
    ***************************************************************/
}
```

### Related Information

- “fgetc() — Read a Character” on page 123
- “fgetwc() — Read Wide Character from Stream ” on page 127
- “fgetws() — Read Wide-Character String from Stream ” on page 129
- “getc() – getchar() — Read a Character” on page 174
- “getwc() — Read Wide Character from Stream” on page 179
- “ungetwc() — Push Wide Character onto Input Stream ” on page 453
- “<wchar.h>” on page 16

### gmtime() — Convert Time

#### Format

```c
#include <time.h>
struct tm *gmtime(const time_t *time);
```

#### Language Level

ANSI

#### Threading Safe

No

Use gmtime_r() instead.

#### Description

The `gmtime()` function breaks down the `time` value, in seconds, and stores it in a `tm` structure, defined in `<time.h>`. The value `time` is usually obtained by a call to the `time()` function.
The fields of the tm structure include:

- **tm_sec**
  - Seconds (0-61)
- **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.

**Return Value**

The `gmtime()` function returns a pointer to the resulting tm structure.

**Note:**

1. The range (0-61) for tm_sec allows for as many as two leap seconds.
2. The `gmtime()` and `localtime()` functions can use a common, statically allocated buffer for the conversion. Each call to one of these functions might alter the result of the previous call.
3. Calendar time is the number of seconds that have elapsed since EPOCH, which is 00:00:00, January 1, 1970 Universal Coordinate Time (UTC).

**Example**

This example uses the `gmtime()` function to adjust a time_t representation to a Coordinated Universal Time character string, and then converts it to a printable string using the `asctime()` function.

```c
#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 should be similar to:  **********
Coordinated Universal Time is Wed Aug 18 21:01:44 1993
**********
```

**Related Information**

- “asctime() — Convert Time to Character String” on page 63
- “asctime_r() — Convert Time to Character String (Restartable)” on page 65
gmttime64() — Convert Time

Format

```c
#include <time.h>
struct tm *gmtime64(const time64_t *time);
```

Language Level

ILE C Extension

Threadsafe

No

Use gmttime64_r() instead.

Description

The gmttime64() function breaks down the `time` value, in seconds, and stores it in a `tm` structure, defined in `<time.h>`. The value `time` is usually obtained by a call to the `time64()` function.

The fields of the `tm` structure include:

- **tm_sec**
  - Seconds (0-61)

- **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.

**Return Value**

The `gmtime64()` function returns a pointer to the resulting `tm` structure.

**Note:**

1. The range (0-61) for `tm_sec` allows for as many as two leap seconds.
2. The `gmtime64()` and `localtime64()` functions can use a common, statically allocated buffer for the conversion. Each call to one of these functions might alter the result of the previous call. The `asctime_r()`, `ctime64_r()`, `gmtime64_r()`, and `localtime64_r()` functions do not use a common statically allocated buffer to hold the return string. These functions can be used in place of the `asctime()`, `ctime64()`, `gmtime64()`, and `localtime64()` functions if reentrancy is desired.
3. Calendar time is the number of seconds that have elapsed since EPOCH, which is 00:00:00, January 1, 1970 Universal Coordinate Time (UTC).

**Example**

This example uses the `gmtime64()` function to adjust a `time64_t` representation to a Universal Coordinate Time character string and then converts it to a printable string using the `asctime()` function.

```c
#include <stdio.h>
#include <time.h>

int main(void)
{
    time64_t ltime;
    time64(&ltime);
    printf("Universal Coordinate Time is %s",
        asctime(gmtime64(&ltime)));
}
```

/* Output should be similar to: *******
Universal Coordinate Time is Wed Aug 18 21:01:44 1993 */

**Related Information**

- “asctime() — Convert Time to Character String” on page 63
- “asctime_r() — Convert Time to Character String (Restartable)” on page 65
- “ctime() — Convert Time to Character String” on page 96
- “ctime64() — Convert Time to Character String” on page 98
- “ctime64_r() — Convert Time to Character String (Restartable)” on page 101
- “ctime_r() — Convert Time to Character String (Restartable)” on page 100
- “gmtime() — Convert Time” on page 183
- “gmtime_r() — Convert Time (Restartable)” on page 187
- “gmtime64_r() — Convert Time (Restartable)” on page 189
- “localtime() — Convert Time” on page 207
gmtime_r() — Convert Time (Restartable)

Format

```c
#include <time.h>
struct tm *gmtime_r(const time_t *time, struct tm *result);
```

Language Level

XPG4

Threadsafe

Yes

Description

This function is the restartable version of gmtime().

The gmtime_r() function breaks down the time value, in seconds, and stores it in result. result is a pointer to the tm structure, defined in <time.h>. The value time is usually obtained by a call to the time() function.

The fields of the tm structure include:

- **tm_sec**
  - Seconds (0-61)

- **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.

**Return Value**
The `gmtime_r()` function returns a pointer to the resulting `tm` structure.

**Note:**
1. The range (0-61) for `tm_sec` allows for as many as two leap seconds.
2. The `gmtime()` and `localtime()` functions can use a common, statically allocated buffer for the conversion. Each call to one of these functions might alter the result of the previous call. The `asctime_r()`, `ctime_r()`, `gmtime_r()`, and `localtime_r()` functions do not use a common, statically allocated buffer to hold the return string. These functions can be used in place of the `asctime()`, `ctime()`, `gmtime()`, and `localtime()` functions if reentrancy is desired.
3. Calendar time is the number of seconds that have elapsed since EPOCH, which is 00:00:00, January 1, 1970 Universal Coordinate Time (UTC).

**Example**
This example uses the `gmtime_r()` function to adjust a `time_t` representation to a Coordinated Universal Time character string, and then converts it to a printable string using the `asctime_r()` function.

```c
#include <stdio.h>
#include <time.h>

int main(void)
{
    time_t ltime;
    struct tm mytime;
    char buf[50];
    time(&ltime)
    printf("Coordinated Universal Time is %s\n", 
        asctime_r(gmtime_r(&ltime, &mytime), buf));
}

/******************************** Output should be similar to:  **********
Coordinated Universal Time is Wed Aug 18 21:01:44 1993
*/
```

**Related Information**
- “`asctime()` — Convert Time to Character String” on page 63
- “`asctime_r()` — Convert Time to Character String (Restartable)” on page 65
- “`ctime()` — Convert Time to Character String” on page 96
- “`ctime64()` — Convert Time to Character String” on page 98
- “`ctime64_r()` — Convert Time to Character String (Restartable)” on page 101
- “`ctime_r()` — Convert Time to Character String (Restartable)” on page 100
- “`gmtime()` — Convert Time” on page 183
- “`gmtime64()` — Convert Time” on page 185
- “`gmtime64_r()` — Convert Time (Restartable)” on page 189
- “`localtime()` — Convert Time” on page 207
- “`localtime64()` — Convert Time” on page 208
- “`localtime64_r()` — Convert Time (Restartable)” on page 211
- “`localtime_r()` — Convert Time (Restartable)” on page 210
- “`mktime()` — Convert Local Time” on page 241
gmtime64_r() — Convert Time (Restartable)

Format
#include <time.h>
struct tm *gmtime64_r(const time64_t *time, struct tm *result);

Language Level
ILE C Extension

Threading Safe
Yes

Description
This function is the restartable version of gmtime64().

The gmtime64_r() function breaks down the time value, in seconds, and stores it in result. result is a
pointer to the tm structure, defined in <time.h>. The value time is usually obtained by a call to the
time64() function.

The fields of the tm structure include:

- **tm_sec**
  - Seconds (0-61)

- **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.

Return Value
The gmtime64_r() function returns a pointer to the resulting tm structure.

Note:
1. The range (0-61) for tm_sec allows for as many as two leap seconds.

2. The gmtime64() and localtime64() functions might use a common, statically allocated buffer for the conversion. Each call to one of these functions might alter the result of the previous call. The asctime_r(), ctime64_r(), gmtime64_r(), and localtime64_r() functions do not use a common, statically allocated buffer to hold the return string. These functions can be used in place of the asctime(), ctime64(), gmtime64(), and localtime64() functions if reentrancy is desired.

3. Calendar time is the number of seconds that have elapsed since EPOCH, which is 00:00:00, January 1, 1970 Universal Coordinate Time (UTC).

Example

This example uses the gmtime64_r() function to adjust a time64_t representation to a Universal Coordinate Time character string and then converts it to a printable string using the asctime_r() function.

```c
#include <stdio.h>
#include <time.h>

int main(void)
{
    time64_t ltime;
    struct tm mytime;
    char buf[50];

    time64(&ltime)
    printf("Universal Coordinate Time is %s",
        asctime_r(gmtime64_r(&ltime, &mytime), buf));
}

/************************  Output should be similar to:  **********
Universal Coordinate Time is Wed Aug 18 21:01:44 1993
*/
```

Related Information

- “asctime() — Convert Time to Character String” on page 63
- “asctime_r() — Convert Time to Character String (Restartable)” on page 65
- “ctime() — Convert Time to Character String” on page 96
- “ctime64() — Convert Time to Character String” on page 98
- “ctime64_r() — Convert Time to Character String (Restartable)” on page 101
- “ctime_r() — Convert Time to Character String (Restartable)” on page 100
- “gmtime() — Convert Time” on page 183
- “gmtime64() — Convert Time” on page 185
- “gmtime64_r() — Convert Time (Restartable)” on page 187
- “localtime() — Convert Time” on page 207
- “localtime64() — Convert Time” on page 208
- “localtime64_r() — Convert Time (Restartable)” on page 211
- “localtime_r() — Convert Time (Restartable)” on page 210
- “mktime() — Convert Local Time” on page 241
- “mktime64() — Convert Local Time” on page 242
- “time() — Determine Current Time” on page 441
- “time64() — Determine Current Time” on page 443
- “<time.h>” on page 16
hypot() — Calculate Hypotenuse

Format

```c
#include <math.h>
double hypot(double side1, double side2);
```

Language Level

ANSI

Threadsafe

Yes

Description

The `hypot()` function calculates the length of the hypotenuse of a right-angled triangle based on the lengths of two sides `side1` and `side2`. A call to the `hypot()` function is equivalent to:

```c
sqrt(side1 * side1 + side2 * side2);
```

Return Value

The `hypot()` function returns the length of the hypotenuse. If an overflow results, `hypot()` sets `errno` to `ERANGE` and returns the value `HUGE_VAL`. If an underflow results, `hypot()` sets `errno` to `ERANGE` and returns zero. The value of `errno` can also be set to `EDOM`.

Example

This example calculates the hypotenuse of a right-angled triangle with sides of 3.0 and 4.0.

```c
#include <math.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
", x, y, z);
}
```

Output should be similar to:

The hypotenuse of the triangle with sides 3.0 and 4.0 is 5.0

Related Information

- “`sqrt()` — Calculate Square Root” on page 380
- “`<math.h>`” on page 6
isalnum() – isxdigit() — Test Integer Value

Format

```c
#include <ctype.h>
int isalnum(int c);
/* Test for upper- or lowercase letters, or decimal digit */
int isalpha(int c);
/* Test for alphabetic character */
int isblank(int c);
/* Test for blank or tab character */
int iscntrl(int c);
/* Test for any control character */
int isdigit(int c);
/* Test for decimal digit */
int isgraph(int c);
/* Test for printable character excluding space */
int islower(int c);
/* Test for lowercase */
int isprint(int c);
/* Test for printable character including space */
int ispunct(int c);
/* Test for any nonalphanumeric printable character excluding space */
int isspace(int c);
/* Test for whitespace character */
int isupper(int c);
/* Test for uppercase */
int isxdigit(int c);
/* Test for hexadecimal digit */
```

Language Level

ANSI

Threadsafe

Yes

Locale Sensitive

The behavior of these functions might be affected by the LC_CTYPE category of the current locale. For more information, see “Understanding CCSIDs and Locales” on page 565.

Description

The `<ctype.h>` functions listed test a character with an integer value.

Return Value

These functions return a nonzero value if the integer satisfies the test condition, or a zero value if it does not. The integer variable `c` must be representable as an unsigned char.

Note: EOF is a valid input value.

Example

This example analyzes all characters between code 0x0 and code UPPER_LIMIT, printing A for alphabetic characters, AN for alphanumerics, B for blank or tab characters, U for uppercase, L for lowercase, D for digits, X for hexadecimal digits, S for spaces, PU for punctuation, PR for printable characters, G for graphics characters, and C for control characters. This example prints the code if printable.

The output of this example is a 256-line table showing the characters from 0 to 255 that possess 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("%3s ", isalnum(ch) ? "AN" : "
printf("%2s ", isalpha(ch) ? "A" : "
printf("%2s ", isblank(ch) ? "B" : "
printf("%2s", iscntrl(ch) ? "C" : "
printf("%2s", isdigit(ch) ? "D" : "
printf("%2s", isgraph(ch) ? "G" : "
printf("%2s", islower(ch) ? "L" : "
printf(" %c", isprint(ch) ? ch
: '
printf("%3s", ispunct(ch) ? "PU" : "
printf("%2s", isspace(ch) ? "S" : "
printf("%3s", isprint(ch) ? "PR" : "
printf("%2s", isupper(ch) ? "U" : "
printf("%2s", isxdigit(ch) ? "X" : "

}

}

");
");
");
");
");
");
");
');
");
");
");
");
");

putchar('\n');

Related Information
• “tolower() – toupper() — Convert Character Case” on page 447
• “<ctype.h>” on page 1

isascii() — Test for Character Representable as ASCII Value
Format
#include <ctype.h>
int isascii(int c);

Language Level
XPG4
Threadsafe
Yes
Locale Sensitive
The behavior of this function might be affected by the LC_CTYPE category of the current locale. This
function is not available when LOCALETYPE(*CLD) is specified on the compilation command. For more
information, see “Understanding CCSIDs and Locales” on page 565.
Description
The isascii() function tests if a given character, in the current locale, can be represented as a valid 7–
bit US-ASCII character.

Library Functions 193


Return Value
The isascii() function returns nonzero if c, in the current locale, can be represented as a character in
the 7–bit US-ASCII character set. Otherwise, it returns 0.
Example
This example tests the integers from 0x7c to 0x82, and prints the corresponding character if the integer
can be represented as a character in the 7–bit US-ASCII character set.
#include <stdio.h>
#include <ctype.h>
int main(void)
{
int ch;

}

for (ch = 0x7c; ch <= 0x82; ch++) {
printf("%#04x
", ch);
if (isascii(ch))
printf("The character is %c\n", ch);
else
printf("Cannot be represented by an ASCII character\n");
}
return 0;
/************************************************
The output should be:
0x7c
0x7d
0x7e
0x7f
0x80
0x81
0x82

The character is @
The character is '
The character is =
The character is "
Cannot be represented by an ASCII character
The character is a
The character is b

************************************************/

Related Information
• “isalnum() – isxdigit() — Test Integer Value” on page 192
• “iswalnum() – iswxdigit() — Test Wide Integer Value” on page 194
• “toascii() — Convert Character to Character Representable by ASCII” on page 446
• “tolower() – toupper() — Convert Character Case” on page 447
• “towlower() – towupper() — Convert Wide Character Case” on page 449
• “<ctype.h>” on page 1

iswalnum() – iswxdigit() — Test Wide Integer Value
Format
#include <wctype.h>
int iswalnum(wint_t wc);
int iswalpha(wint_t wc);
int iswblank(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);

194 IBM i: ILE C/C++ Runtime Library Functions


Locale Sensitive

The behavior of these functions might be affected by the LC_CTYPE category of the current locale if LOCALETYPE(*LOCALE) is specified on the compilation command. The behavior of these functions might be affected by the LC_UNI_CTYPE category of the current locale if either the LOCALETYPE(*LOCALEUCS2) option or the LOCALETYPE(*LOCALEUTF) option is specified on the compilation command. These functions are not available when LOCALETYPE(*CLD) is specified on the compilation command. For more information, see “Understanding CCSIDs and Locales” on page 565.

Wide Character Function

See “Wide Characters” on page 568 for more information.

Description

The functions listed above, which are all declared in <wchar.h>, test a given wide integer value. The value of wc must be a wide-character code corresponding to a valid character in the current locale, or must equal the value of the macro WEOF. If the argument has any other value, the behavior is undefined.

Here are descriptions of each function in this group.

iswalnum()
Test for a wide alphanumeric character.

iswalpha()
Test for a wide alphabetic character, as defined in the alpha class of the current locale.

iswblank()
Test for a wide blank or tab character, as defined in the blank class of the current locale.

iswcntrl()
Test for a wide control character, as defined in the cntrl class of the current locale.

iswdigit()
Test for a wide decimal-digit character: 0 through 9, as defined in the digit class of the current locale.

iswgraph()
Test for a wide printing character, not a space, as defined in the graph class of the current locale.

iswlower()
Test for a wide lowercase character, as defined in the lower class of the current locale or for which none of the iswcntrl(), iswdigit(), iswspace() functions are true.

iswprint()
Test for any wide printing character, as defined in the print class of the current locale.

iswpunct()
Test for a wide nonalphanumeric, nonspace character, as defined in the punct class of the current locale.

iswspace()
Test for a wide whitespace character, as defined in the space class of the current locale.

iswupper()
Test for a wide uppercase character, as defined in the upper class 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 class of the current locale.
Returned Value

These functions return a nonzero value if the wide integer satisfies the test value, or a 0 value if it does not. The value for wc must be representable as a wide unsigned char. WEOF is a valid input value.

Example

```c
#include <stdio.h>
#include <wctype.h>

int main(void)
{
    int 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", iswblank(wc)  ? "B"  : " ");
        printf("%2s", iswcntrl(wc) ? "C"  : " ");
        printf("%2s", iswdigit(wc) ? "D"  : " ");
        printf("%2s", iswgraph(wc) ? "G"  : " ");
        printf("%2s", iswlower(wc)  ? "L"  : " ");
        printf("%2s", iswprint(wc)  ? wc   : ' ');
        printf("%3s", iswpunct(wc) ? "PU" : " ");
        printf("%2s", iswspace(wc)  ? "S"  : " ");
        printf("%3s", iswxdigit(wc) ? "X"  : " ");
        putchar(\n');
    }
}
```

Related Information

- “<wctype.h>” on page 17

iswctype() — Test for Character Property

Format

```c
#include <wctype.h>
int iswctype(wint_t wc, wctype_t wc_prop);
```

Language Level

ANSI

Threatsafe

Yes

Locale Sensitive

The behavior of this function might be affected by the LC_CTYPE category of the current locale if LOCALETYPE(*LOCALE) is specified on the compilation command. The behavior of this function might be affected by the LC_UNI_CTYPE category of the current locale if either the LOCALETYPE(*LOCALEUCS2) option or the LOCALETYPE(*LOCALEUTF) option is specified on the compilation command. This function is not available when LOCALETYPE(*CLD) is specified on the compilation command. For more information, see “Understanding CCSIDs and Locales” on page 565.
Wide Character Function
See “Wide Characters” on page 568 for more information.

Description
The iswctype() function 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 characters that corresponds to a multibyte character, the behavior is undefined. If the value of wc_prop is incorrect (that is, it is not obtained by a previous call to the wctype() function, or wc_prop has been invalidated by a subsequent call to the setlocale() function), the behavior is undefined.

Return Value
The iswctype() function returns true if the value of the wide character wc has the property wc_prop.

The following strings, alnum through to xdigit are reserved for the standard character classes. The functions are shown as follows with their equivalent isw*() function:

```c
iswctype(wc, wctype("alnum"));     /* is equivalent to */        iswalnum(wc);
iswctype(wc, wctype("alpha"));     /* is equivalent to */        iswalpha(wc);
iswctype(wc, wctype("blank"));     /* is equivalent to */        iswblank(wc);
iswctype(wc, wctype("cntrl"));     /* is equivalent to */        iswcntrl(wc);
iswctype(wc, wctype("digit"));     /* is equivalent to */        iswdigit(wc);
iswctype(wc, wctype("graph"));     /* is equivalent to */        iswgraph(wc);
iswctype(wc, wctype("lower"));     /* is equivalent to */        iswlower(wc);
iswctype(wc, wctype("print"));     /* is equivalent to */        iswprint(wc);
iswctype(wc, wctype("punct"));     /* is equivalent to */        iswpunct(wc);
iswctype(wc, wctype("space"));     /* is equivalent to */        iswspace(wc);
iswctype(wc, wctype("upper"));     /* is equivalent to */        iswupper(wc);
iswctype(wc, wctype("xdigit"));    /* is equivalent to */        iswxdigit(wc);
```

Example
```c
#include <stdio.h>
#include <wctype.h>

int main(void)
{
    int wc;
    for (wc=0; wc <= 0xFF; wc++) {
        printf("%3d", wc);
        printf("%04x", wc);
        printf("%s", iswctype(wc, wctype("alnum"))  ? "AN" : " ");
        printf("%s", iswctype(wc, wctype("alpha"))  ? "A" : " ");
        printf("%s", iswctype(wc, wctype("blank"))  ? "B" : " ");
        printf("%s", iswctype(wc, wctype("cntrl"))  ? "C" : " ");
        printf("%s", iswctype(wc, wctype("digit"))  ? "D" : " ");
        printf("%s", iswctype(wc, wctype("graph"))  ? "G" : " ");
        printf("%s", iswctype(wc, wctype("lower"))  ? "L" : " ");
        printf("%c", iswctype(wc, wctype("print"))  ? wc   : ' ');
        printf("%s", iswctype(wc, wctype("punct"))  ? "PU" : " ");
        printf("%s", iswctype(wc, wctype("space"))  ? "S" : " ");
        printf("%s", iswctype(wc, wctype("print"))  ? "PR" : " ");
        printf("%s", iswctype(wc, wctype("upper"))  ? "U" : " ");
        printf("%s", iswctype(wc, wctype("xdigit"))  ? "X" : " ");
    }
    putchar('\n');
}
```

Related Information
- “wctype() — Get Handle for Character Property Classification” on page 529
- “iswalnum() – iswxdigit() — Test Wide Integer Value” on page 194
- “<wctype.h>” on page 17
_itoa() — Convert Integer to String

**Format**
```
#include <stdlib.h>
char *_itoa(int value, char *string, int radix);
```

**Note:** The _itoa function is supported only for C++, not for C.

**Language Level**
Extension

**Threadsafe**
Yes

**Description**
_itoa() converts the digits of the given value to a character string that ends with a null character and stores the result in string. The radix argument specifies the base of value; it must be in the range 2 to 36. If radix equals 10 and value is negative, the first character of the stored string is the minus sign (-).

**Note:** The space reserved for string must be large enough to hold the returned string. The function can return up to 33 bytes including the null character (\0).

**Return Value**
_itoa returns a pointer to string. There is no error return value.

When the string argument is NULL or the radix is outside the range 2 to 36, errno will be set to EINVAL.

**Example**
This example converts the integer value -255 to a decimal, a binary, and a hex number, storing its character representation in the array buffer.

```
#include <stdio.h>
#include <stdlib.h>
int main(void)
{
    char buffer[35];
    char *p;
    p = _itoa(-255, buffer, 10);
    printf("The result of _itoa(-255) with radix of 10 is %s\n", p);
    p = _itoa(-255, buffer, 2);
    printf("The result of _itoa(-255) with radix of 2 is %s\n", p);
    p = _itoa(-255, buffer, 16);
    printf("The result of _itoa(-255) with radix of 16 is %s\n", p);
    return 0;
}
```

The output should be:
```
The result of _itoa(-255) with radix of 10 is -255
The result of _itoa(-255) with radix of 2 is 11111111111111111111111100000001
The result of _itoa(-255) with radix of 16 is ffffff01
```

**Related Information**
- “_gcvt() — Convert Floating-Point to String” on page 173
- “_itoa() — Convert Integer to String” on page 198
labs() – llabs() — Calculate Absolute Value of Long and Long Long Integer

Format (labs())

```c
#include <stdlib.h>
long int labs(long int n);
```

Format (llabs())

```c
#include <stdlib.h>
long long int llabs(long long int i);
```

Language Level

ANSI

Threadsafe

Yes

Description

The `labs()` function produces the absolute value of its long integer argument `n`. The result might be undefined when the argument is equal to `LONG_MIN`, the smallest available long integer. The value `LONG_MIN` is defined in the `<limits.h>` include file.

The `llabs()` function returns the absolute value of its long long integer operand. The result might be undefined when the argument is equal to `LONG_LONG_MIN`, the smallest available long integer. The value `LONG_LONG_MIN` is defined in the `<limits.h>` include file.

Return Value

The `labs()` function returns the absolute value of `n`. There is no error return value.

The `llabs()` function returns the absolute value of `i`. There is no error return value.

Example

This example computes `y` as the absolute value of the long integer `-41567`.

```c
#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 should be similar to:

```
The absolute value of -41567 is 41567
```
Idexp() — Multiply by a Power® of Two

Format

```c
#include <math.h>
double ldexp(double x, int exp);
```

Language Level

ANSI

Threadsafe

Yes

Description

The ldexp() function calculates the value of \( x \times (2^{exp}) \).

Return Value

The ldexp() function returns the value of \( x \times (2^{exp}) \). If an overflow results, the function returns +HUGE_VAL for a large result or -HUGE_VAL for a small result, and sets errno to ERANGE.

Example

This example computes \( y \) as 1.5 times 2 to the fifth power (1.5*2^5):

```c
#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 should be similar to:  *************
1.500000 times 2 to the power of 5 is 48.000000
*/

Related Information

• “exp() — Calculate Exponential Function” on page 114
• “frexp() — Separate Floating-Point Value” on page 154
• “modf() — Separate Floating-Point Value” on page 244
• “<math.h>” on page 6
ldiv() – lldiv() — Perform Long and Long Long Division

**Format (ldiv())**

```c
#include <stdlib.h>
ldiv_t ldiv(long int numerator, long int denominator);
```

**Format (lldiv())**

```c
#include <stdlib.h>
lldiv_t lldiv(long long int numerator, long long int denominator);
```

**Language Level**

ANSI

**Threadsafe**

Yes

However, only the function version is threadsafe. The macro version is NOT threadsafe.

**Description**

The `ldiv()` function calculates the quotient and remainder of the division of `numerator` by `denominator`.

**Return Value**

The `ldiv()` function 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 return value is undefined. If `denominator` is 0, an exception is raised.

The `lldiv()` function computes the quotient and remainder of the `numerator` parameter by the `denominator` parameter.

The `lldiv()` function returns a structure of type `lldiv_t`, containing both the quotient and the remainder. The structure is defined as:

```c
struct lldiv_t
{
  long long int quot; /* quotient */
  long long int rem; /* remainder */
};
```

If the division is inexact, the sign of the resulting quotient is that of the algebraic quotient, and magnitude of the resulting quotient is the largest long long integer less than the magnitude of the algebraic quotient. If the result cannot be represented (for example, if the `denominator` is 0), the behavior is undefined.

**Example**

This example uses `ldiv()` to calculate the quotients and remainders for a set of two dividends and two divisors.
```c
#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);
        }

} /* End of main() */
```

---

**Related Information**

- "div() — Calculate Quotient and Remainder" on page 111
- "<stdlib.h>" on page 15

---

### localeconv() — Retrieve Information from the Environment

**Format**

```c
#include <locale.h>
struct lconv *localeconv(void);
```

**Language Level**

ANSI

**Threadsafe**

Yes

**Locale Sensitive**

The behavior of this function might be affected by the LC_NUMERIC and LC_MONETARY categories of the current locale. For more information, see “Understanding CCSIDs and Locales” on page 565.

**Description**

The `localeconv()` sets the components of a structure having type `struct lconv` to values appropriate for the current locale. The structure might be overwritten by another call to `localeconv()`, or by calling the `setlocale()` function.

The structure contains the following elements (defaults shown are for the C locale):

---

202 IBM i: ILE C/C++ Runtime Library Functions
<table>
<thead>
<tr>
<th>Element</th>
<th>Purpose of Element</th>
<th>Default</th>
</tr>
</thead>
<tbody>
<tr>
<td>char *decimal_point</td>
<td>Decimal-point character used to format non-monetary quantities.</td>
<td>&quot;.&quot;</td>
</tr>
<tr>
<td>char *thousands_sep</td>
<td>Character used to separate groups of digits to the left of the decimal-point character in formatted non-monetary quantities.</td>
<td>&quot;&quot;</td>
</tr>
<tr>
<td>char *grouping</td>
<td>String indicating the size of each group of digits in formatted non-monetary quantities. Each character in the string specifies the number of digits in a group. The initial character represents the size of the group immediately to the left of the decimal delimiter. The characters following this define succeeding groups to the left of the previous group. If the last character is not UCHAR_MAX, the grouping is repeated using the last character as the size. If the last character is UCHAR_MAX, grouping is only performed for the groups already in the string (no repetition). See Table 1 on page 204 for an example of how grouping works.</td>
<td>&quot;&quot;</td>
</tr>
<tr>
<td>char *int_curr_symbol</td>
<td>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.</td>
<td>&quot;&quot;</td>
</tr>
<tr>
<td>char *currency_symbol</td>
<td>Local currency symbol of the current locale.</td>
<td>&quot;&quot;</td>
</tr>
<tr>
<td>char *mon_decimal_point</td>
<td>Decimal-point character used to format monetary quantities.</td>
<td>&quot;&quot;</td>
</tr>
<tr>
<td>char *mon_thousands_sep</td>
<td>Separator for digits in formatted monetary quantities.</td>
<td>&quot;&quot;</td>
</tr>
<tr>
<td>char *mon_grouping</td>
<td>String indicating the size of each group of digits in formatted monetary quantities. Each character in the string specifies the number of digits in a group. The initial character represents the size of the group immediately to the left of the decimal delimiter. The following characters define succeeding groups to the left of the previous group. If the last character is not UCHAR_MAX, the grouping is repeated using the last character as the size. If the last character is UCHAR_MAX, grouping is only performed for the groups already in the string (no repetition). See Table 1 on page 204 for an example of how grouping works.</td>
<td>&quot;&quot;</td>
</tr>
<tr>
<td>char *positive_sign</td>
<td>String indicating the positive sign used in monetary quantities.</td>
<td>&quot;&quot;</td>
</tr>
<tr>
<td>char *negative_sign</td>
<td>String indicating the negative sign used in monetary quantities.</td>
<td>&quot;&quot;</td>
</tr>
<tr>
<td>Element</td>
<td>Purpose of Element</td>
<td>Default</td>
</tr>
<tr>
<td>-----------------------</td>
<td>------------------------------------------------------------------------------------</td>
<td>----------</td>
</tr>
<tr>
<td>char int_frac_digits</td>
<td>The number of displayed digits to the right of the decimal place for internationally formatted monetary quantities.</td>
<td>UCHAR_MAX</td>
</tr>
<tr>
<td>char frac_digits</td>
<td>Number of digits to the right of the decimal place in monetary quantities.</td>
<td>UCHAR_MAX</td>
</tr>
<tr>
<td>char p_cs_precedes</td>
<td>1 if the currency_symbol precedes the value for a nonnegative formatted monetary quantity; 0 if it does not.</td>
<td>UCHAR_MAX</td>
</tr>
<tr>
<td>char p_sep_by_space</td>
<td>1 if the currency_symbol is separated by a space from the value of a nonnegative formatted monetary quantity; 0 if it does not.</td>
<td>UCHAR_MAX</td>
</tr>
<tr>
<td>char n_cs_precedes</td>
<td>1 if the currency_symbol precedes the value for a negative formatted monetary quantity; 0 if it does not.</td>
<td>UCHAR_MAX</td>
</tr>
<tr>
<td>char n_sep_by_space</td>
<td>1 if the currency_symbol is separated by a space from the value of a negative formatted monetary quantity; 0 if it does not.</td>
<td>UCHAR_MAX</td>
</tr>
<tr>
<td>char p_sign_posn</td>
<td>Value indicating the position of the positive_sign for a nonnegative formatted monetary quantity.</td>
<td>UCHAR_MAX</td>
</tr>
<tr>
<td>char n_sign_posn</td>
<td>Value indicating the position of the negative_sign for a negative formatted monetary quantity.</td>
<td>UCHAR_MAX</td>
</tr>
</tbody>
</table>

Pointers to strings with a value of "" indicate that the value is not available in the C locale or is of zero length. Elements with char types with a value of UCHAR_MAX indicate that the value is not available in the current locale.

The n_sign_posn and p_sign_posn elements can have the following values:

<table>
<thead>
<tr>
<th>Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>The quantity and currency_symbol are enclosed in parentheses.</td>
</tr>
<tr>
<td>1</td>
<td>The sign precedes the quantity and currency_symbol.</td>
</tr>
<tr>
<td>2</td>
<td>The sign follows the quantity and currency_symbol.</td>
</tr>
<tr>
<td>3</td>
<td>The sign precedes the currency_symbol.</td>
</tr>
<tr>
<td>4</td>
<td>The sign follows the currency_symbol.</td>
</tr>
</tbody>
</table>

### Table 1. Grouping Example

<table>
<thead>
<tr>
<th>Locale Source</th>
<th>Grouping String</th>
<th>Number</th>
<th>Formatted Number</th>
</tr>
</thead>
<tbody>
<tr>
<td>-1</td>
<td>0x00</td>
<td>123456789</td>
<td>123456789</td>
</tr>
<tr>
<td>3</td>
<td>0x0300</td>
<td>123456789</td>
<td>123,456,789</td>
</tr>
<tr>
<td>3;:-1</td>
<td>0x03FF00</td>
<td>123456789</td>
<td>123456,789</td>
</tr>
</tbody>
</table>

204 IBM i: ILE C/C++ Runtime Library Functions
Table 1. Grouping Example (continued)

<table>
<thead>
<tr>
<th>Locale Source</th>
<th>Grouping String</th>
<th>Number</th>
<th>Formatted Number</th>
</tr>
</thead>
<tbody>
<tr>
<td>3;2;1</td>
<td>0x03020100</td>
<td>123456789</td>
<td>1,2,3,4,56,789</td>
</tr>
</tbody>
</table>

Table 2. Monetary Formatting Example

<table>
<thead>
<tr>
<th>Country</th>
<th>Positive Format</th>
<th>Negative Format</th>
<th>International Format</th>
</tr>
</thead>
<tbody>
<tr>
<td>Italy</td>
<td>L.1.230</td>
<td>-L.1.230</td>
<td>ITL.1.230</td>
</tr>
<tr>
<td>Netherlands</td>
<td>F 1.234,56</td>
<td>F -1.234,56</td>
<td>NLG 1.234,56</td>
</tr>
<tr>
<td>Norway</td>
<td>kr1.234,56</td>
<td>kr1.234,56-</td>
<td>NOK1.234,56</td>
</tr>
<tr>
<td>Switzerland</td>
<td>SFRs.1,234.56</td>
<td>SFrx.1,234.56C</td>
<td>CHF 1,234.56</td>
</tr>
</tbody>
</table>

The above table was generated by locales with the following monetary fields:

Table 3. Monetary Fields

<table>
<thead>
<tr>
<th></th>
<th>Italy</th>
<th>Netherlands</th>
<th>Norway</th>
<th>Switzerland</th>
</tr>
</thead>
<tbody>
<tr>
<td>int_curr_symbol</td>
<td>&quot;ITL.&quot;</td>
<td>&quot;NLG&quot;</td>
<td>&quot;NOK&quot;</td>
<td>&quot;CHF&quot;</td>
</tr>
<tr>
<td>currency_symbol</td>
<td>&quot;L.&quot;</td>
<td>&quot;F&quot;</td>
<td>&quot;kr&quot;</td>
<td>&quot;SFrs.&quot;</td>
</tr>
<tr>
<td>mon_decimal_point</td>
<td>&quot;&quot;</td>
<td>&quot;,&quot;</td>
<td>&quot;,&quot;</td>
<td>&quot;,&quot;</td>
</tr>
<tr>
<td>mon_thousands_sep</td>
<td>&quot;,&quot;</td>
<td>&quot;,&quot;</td>
<td>&quot;,&quot;</td>
<td>&quot;,&quot;</td>
</tr>
<tr>
<td>mon_grouping</td>
<td>&quot;\3&quot;</td>
<td>&quot;\3&quot;</td>
<td>&quot;\3&quot;</td>
<td>&quot;\3&quot;</td>
</tr>
<tr>
<td>positive_sign</td>
<td>&quot;&quot;</td>
<td>&quot;&quot;</td>
<td>&quot;&quot;</td>
<td>&quot;&quot;</td>
</tr>
<tr>
<td>negative_sign</td>
<td>&quot;-&quot;</td>
<td>&quot;-&quot;</td>
<td>&quot;-&quot;</td>
<td>&quot;C&quot;</td>
</tr>
<tr>
<td>int_frac_digits</td>
<td>0</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>frac_digits</td>
<td>0</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>p_cs_precedes</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>p_sep_by_space</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>n_cs_precedes</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>n_sep_by_space</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>p_sep_posn</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>n_sign_posn</td>
<td>1</td>
<td>4</td>
<td>2</td>
<td>2</td>
</tr>
</tbody>
</table>

Return Value

The localeconv() function returns a pointer to the structure.

Example that uses *CLD locale objects

This example prints out the default decimal point for your locale and then the decimal point for the LC_C_FRANCE locale.
```c
#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, LC_C_FRANCE))) {
        mylocale = localeconv();
        /* A comma is set to be the decimal point when the locale is LC_C_FRANCE*/
        printf("France's decimal point is a %s\n", mylocale->decimal_point);
    } else {
        printf("setlocale(LC_ALL, LC_C_FRANCE) returned <NULL>\n");
    }
    return 0;
}
```

### Example that uses *LOCALE objects

```
#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, "/QSYS.LIB/MYLIB.LIB/LC_FRANCE.LOCALE"))) {
        mylocale = localeconv();
        /* A comma is set to be the decimal point in the French locale */
        printf("France's decimal point is a %s\n", mylocale->decimal_point);
    } else {
        printf("setlocale(LC_ALL, "/QSYS.LIB/MYLIB.LIB/LC_FRANCE.LOCALE") \ 
    returned <NULL>\n");
    }
    return 0;
}
```

### Related Information

- “setlocale() — Set Locale” on page 366
- “<locale.h>” on page 5
localtime() — Convert Time

Format

```c
#include <time.h>
struct tm *localtime(const time_t *timeval);
```

Language Level

ANSI

Threadsafe

No

Use localtime_r() instead.

Locale Sensitive

The behavior of this function might be affected by the LC_TOD category of the current locale.

Description

The `localtime()` function converts a time value, in seconds, to a structure of type `tm`.

The `localtime()` function takes a `timeval` assumed to be Universal Coordinate Time (UTC) and converts it to job locale time. For this conversion `localtime()` checks the current locale setting for local time zone and daylight saving time (DST). If these values are not set in the current locale, `localtime()` gets the local time zone and daylight saving time (DST) settings from the current job. Once converted, the time is returned in a structure of type `tm`. If the DST is set in the locale but the time zone information is not, the DST information in the locale is ignored.

The time value is usually obtained by a call to the `time()` function.

Note:

1. The `gmtime()` and `localtime()` functions can use a common, statically allocated buffer for the conversion. Each call to one of these functions might destroy the result of the previous call. The `ctime_r()`, `gmtime_r()`, and `localtime_r()` functions do not use a common, statically allocated buffer. These functions can be used in place of the `asctime()`, `ctime()`, `gmtime()` and `localtime()` functions if reentrancy is desired.

2. Calendar time is the number of seconds that have elapsed since EPOCH, which is 00:00:00, January 1, 1970 Universal Coordinate Time (UTC).

Return Value

The `localtime()` function returns a pointer to the structure result. There is no error return value.

Example

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;

    ltime = time(&ltime);
    newtime = localtime(&ltime);
    printf("The date and time is %s", asctime(newtime));
}

Eric: If the local time is 3 p.m. February 15, 2008, the output should be:

The date and time is Fri Feb 15 15:00:00 2008

Related Information

- “asctime() — Convert Time to Character String” on page 63
- “asctime_r() — Convert Time to Character String (Restartable)” on page 65
- “ctime() — Convert Time to Character String” on page 96
- “ctime64() — Convert Time to Character String” on page 98
- “ctime64_r() — Convert Time to Character String (Restartable)” on page 101
- “ctime_r() — Convert Time to Character String (Restartable)” on page 100
- “gmtime() — Convert Time” on page 183
- “gmtime64() — Convert Time” on page 185
- “gmtime64_r() — Convert Time (Restartable)” on page 189
- “gmtime_r() — Convert Time (Restartable)” on page 187
- “localtime_r() — Convert Time (Restartable)” on page 210
- “localtime64() — Convert Time” on page 208
- “localtime64_r() — Convert Time (Restartable)” on page 211
- “mktime() — Convert Local Time” on page 241
- “mktime64() — Convert Local Time” on page 242
- “setlocale() — Set Locale” on page 366
- “time() — Determine Current Time” on page 441
- “time64() — Determine Current Time” on page 443
- “<time.h>” on page 16

localtime64() — Convert Time

Format

#include <time.h>
struct tm *localtime64(const time64_t *timeval);

Language Level

ILE C Extension

Threadsafe

No
Use `localtime64_r()` instead.

**Locale Sensitive**
The behavior of this function might be affected by the LC_TOD category of the current locale.

**Description**
The `localtime64()` function converts a time value, in seconds, to a structure of type `tm`.

The `localtime64()` function takes a `timeval` assumed to be Universal Coordinate Time (UTC) and converts it to job locale time. For this conversion, `localtime64()` checks the current locale setting for local time zone and daylight saving time (DST). If these values are not set in the current locale, `localtime64()` gets the local time zone and daylight saving time (DST) settings from the current job. Once converted, the time is returned in a structure of type `tm`. If the DST is set in the locale but the time zone information is not, the DST information in the locale is ignored.

The time value is usually obtained by a call to the `time64()` function.

**Note:**
1. The `gmtime64()` and `localtime64()` functions might use a common, statically allocated buffer for the conversion. Each call to one of these functions might alter the result of the previous call. The `asctime_r()`, `ctime64_r()`, `gmtime64_r()`, and `localtime64_r()` functions do not use a common, statically allocated buffer. These functions can be used in place of the `asctime()`, `ctime64()`, `gmtime64()`, and `localtime64()` functions if thread safety is desired.
2. Calendar time is the number of seconds that have elapsed since EPOCH, which is 00:00:00, January 1, 1970 Universal Coordinate Time (UTC).
3. The supported date and time range for this function is 01/01/0001 00:00:00 through 12/31/9999 23:59:59.

**Return Value**
The `localtime64()` function returns a pointer to the structure result. If the given `timeval` is out of range, a NULL pointer is returned and `errno` is set to EOVERFLOW.

**Example**
This example queries the system clock and displays the local time.

```c
#include <stdio.h>
#include <time.h>

int main(void)
{
    struct tm *newtime;
    time64_t ltime;

    ltime = time64(&ltime);
    newtime = localtime64(&ltime);
    printf("The date and time is %s", asctime(newtime));
}
```

```c
/************** If the local time is 3 p.m. February 15, 2008, **********
********** the output should be: ****************
The date and time is Fri Feb 15 15:00:00 2008
*/
```

**Related Information**
- “`asctime()` — Convert Time to Character String” on page 63
- “`asctime_r()` — Convert Time to Character String (Restartable)” on page 65
- “`ctime()` — Convert Time to Character String” on page 96
localtime_r() — Convert Time (Restartable)

Format

```c
#include <time.h>
struct tm *localtime_r(const time_t *timeval, struct tm *result);
```

Language Level

XPG4

Threadsafe

Yes

Locale Sensitive

The behavior of this function might be affected by the LC_TOD category of the current locale.

Description

This function is the restartable version of localtime(). It is the same as localtime() except that it passes in the place to store the returned structure result.

Return Value

The localtime_r() returns a pointer to the structure result. There is no error return value.

Example

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;
    char buf[50];
    ltime=time(&ltime);
    localtime_r(&ltime, &newtime);
    printf("The date and time is %s", asctime_r(&newtime, buf));
}

/******* If the local time is 3 p.m. February 15, 2008, **********
************* the output should be: *******************/

The date and time is Fri Feb 15 15:00:00 2008
*/
```

Related Information

- "asctime() — Convert Time to Character String" on page 63
- "asctime_r() — Convert Time to Character String (Restartable)" on page 65
- "ctime() — Convert Time to Character String" on page 96
- "ctime_r() — Convert Time to Character String (Restartable)" on page 100
- "gmtime() — Convert Time" on page 183
- "gmtime_r() — Convert Time (Restartable)" on page 187
- "localtime() — Convert Time" on page 207
- "mktime() — Convert Local Time" on page 241
- "time() — Determine Current Time" on page 441
- "<time.h>" on page 16

### localtime64_r() — Convert Time (Restartable)

**Format**

```c
#include <time.h>
struct tm *localtime64_r(const time64_t *timeval, struct tm *result);
```

**Language Level**

ILE C Extension

**Thesdsafe**

Yes

**Locale Sensitive**

The behavior of this function might be affected by the LC_TOD category of the current locale.

**Description**

This function is the restartable version of localtime64(). It is the same as localtime64() except that it passes in the place to store the returned structure result.

**Note:**
1. The `gmtime64()` and `localtime64()` functions might use a common, statically allocated buffer for the conversion. Each call to one of these functions might alter the result of the previous call. The `asctime_r()`, `ctime64_r()`, `gmtime64_r()`, and `localtime64_r()` functions do not use a common statically allocated buffer to hold the return string. These functions can be used in place of the `asctime()`, `ctime64()`, `gmtime64()`, and `localtime64()` functions if thread safety is desired.

2. Calendar time is the number of seconds that have elapsed since EPOCH, which is 00:00:00, January 1, 1970 Universal Coordinate Time (UTC).

3. The supported date and time range for this function is 01/01/0001 00:00:00 through 12/31/9999 23:59:59.

**Return Value**

The `localtime64_r()` function returns a pointer to the structure result. If the given `timeval` is out of range, a NULL pointer is returned and errno is set to EOVERFLOW.

**Example**

This example queries the system clock and displays the local time.

```c
#include <stdio.h>
#include <time.h>

int main(void)
{
    struct tm newtime;
    time64_t ltime;
    char buf[50];

    ltime = time64(&ltime);
    localtime64_r(&ltime, &newtime);
    printf("The date and time is %s\n", asctime_r(&newtime, buf));
}
```

/* *************** If the local time is 3 p.m. February 15, 2008, ***************
*********** the output should be: ****************************
The date and time is Fri Feb 15 15:00:00 2008 */

**Related Information**

- “`asctime() — Convert Time to Character String” on page 63
- “`asctime_r() — Convert Time to Character String (Restartable)” on page 65
- “`ctime64() — Convert Time to Character String” on page 98
- “`ctime64_r() — Convert Time to Character String (Restartable)” on page 101
- “`gmtime64() — Convert Time” on page 185
- “`gmtime64_r() — Convert Time (Restartable)” on page 189
- “`localtime64() — Convert Time” on page 208
- “`mktime64() — Convert Local Time” on page 242
- “`time64() — Determine Current Time” on page 443
- “<time.h>” on page 16

**log() — Calculate Natural Logarithm**

**Format**

```c
#include <math.h>
double log(double x);
```
log() — Calculate Natural Logarithm

Description
The log() function calculates the natural logarithm (base e) of x.

Return Value
The log() function returns the computed value. If x is negative, log() sets errno to EDOM and might return the value -HUGE_VAL. If x is zero, log() returns the value -HUGE_VAL, and might set errno to ERANGE.

Example
This example calculates the natural logarithm of 1000.0.

```c
#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 should be similar to:  **************
   The natural logarithm of 1000.000000 is 6.907755
   */
```

Related Information
• “exp() — Calculate Exponential Function” on page 114
• “log10() — Calculate Base 10 Logarithm” on page 213
• “pow() — Compute Power” on page 250
• “<math.h>” on page 6

log10() — Calculate Base 10 Logarithm

Format
```c
#include <math.h>
double log10(double x);
```

Language Level
ANSI

Threadsaf
Yes
Description
The `log10()` function calculates the base 10 logarithm of `x`.

Return Value
The `log10()` function returns the computed value. If `x` is negative, `log10()` sets `errno` to `EDOM` and might return the value `-HUGE_VAL`. If `x` is zero, the `log10()` function returns the value `-HUGE_VAL`, and might set `errno` to `ERANGE`.

Example
This example calculates the base 10 logarithm of 1000.0.

```c
#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);
}
```

The base 10 logarithm of 1000.000000 is 3.000000

Related Information
• “exp() — Calculate Exponential Function” on page 114
• “log() — Calculate Natural Logarithm” on page 212
• “pow() — Compute Power” on page 250
• “<math.h>” on page 6

_ltoa() — Convert Long Integer to String

Format
```c
#include <stdlib.h>
char *ltoa(long value, char *string, int radix);
```

Note: The _ltoa function is supported only for C++, not for C.

Language Level
Extension

Threadsafe
Yes

Description
_ltoa converts the digits of the given long integer `value` to a character string that ends with a null character and stores the result in `string`. The `radix` argument specifies the base of `value`; it must be in the range 2 to 36. If `radix` equals 10 and `value` is negative, the first character of the stored string is the minus sign (`-`).
Note: The space allocated for string must be large enough to hold the returned string. The function can return up to 33 bytes including the null character (\0).

Return Value

_ltoa returns a pointer to string. There is no error return value.

When the string argument is NULL or the _radix is outside the range 2 to 36, _errno will be set to EINVAL.

Example

This example converts the integer value -255L to a decimal, a binary, and a hex value, and stores its character representation in the array buffer.

```c
#include <stdio.h>
#include <stdlib.h>
int main(void)
{
    char buffer[35];
    char *p;
    p = _ltoa(-255L, buffer, 10);
    printf("The result of _ltoa(-255) with radix of 10 is %s\n", p);
    p = _ltoa(-255L, buffer, 2);
    printf("The result of _ltoa(-255) with radix of 2\n is %s\n", p);
    p = _ltoa(-255L, buffer, 16);
    printf("The result of _ltoa(-255) with radix of 16 is %s\n", p);
    return 0;
}
```

The output should be:

```
The result of _ltoa(-255) with radix of 10 is -255
The result of _ltoa(-255) with radix of 2
 is 11111111111111111100000001
The result of _ltoa(-255) with radix of 16 is ffffff01
```

Related Information

- “atol() – atoll() — Convert Character String to Long or Long Long Integer” on page 74
- “_gcvt() — Convert Floating-Point to String” on page 173
- “_itoa() — Convert Integer to String” on page 198
- “strtol() – strto11() — Convert Character String to Long and Long Long Integer” on page 430
- “_ultoa() — Convert Unsigned Long Integer to String” on page 451
- “wcstol() – wcstoll() — Convert Wide Character String to Long and Long Long Integer” on page 515
- “<stdlib.h>” on page 15

longjmp() — Restore Stack Environment

Format

```c
#include <setjmp.h>
void longjmp(jmp_buf env, int value);
```

Language Level

ANSI

Threading Safe

Yes
Description

The `longjmp()` function restores a stack environment previously saved in `env` by the `setjmp()` function. The `setjmp()` and `longjmp()` functions provide a way to perform a non-local goto. They are often used in signal handlers.

A call to the `setjmp()` function 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. Processing resumes as if the `setjmp()` call had just returned the given value.

All variables (except register variables) that are available 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 the `setjmp()` and `longjmp()` functions are also unpredictable.

**Note:** Ensure that the function that calls the `setjmp()` function does not return before you call the corresponding `longjmp()` function. Calling `longjmp()` after the function calling the `setjmp()` function returns causes unpredictable program behavior.

The `value` argument must be nonzero. If you give a zero argument for `value`, `longjmp()` substitutes 1 in its place.

Return Value

The `longjmp()` function does not use the normal function call and return mechanisms; it has no return value.

Example

This example saves the stack environment at the statement:

```c
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 the `setjmp()` function returns a 0 when it saves the environment. The program prints the message:

```
setjmp has been called
```

The subsequent call to function `p()` causes it to call the `longjmp()` function. Control is transferred to the point in the `main()` function immediately after the call to the `setjmp()` function using the environment saved in the `mark` variable. This time, the condition is TRUE because -1 is specified in the second parameter on the `longjmp()` function call as the return value to be placed on the stack. The example then performs the statements in the block, prints the message "`longjmp()` has been called", calls the `recover()` function, and leaves the program.
#include <stdio.h>
#include <setjmp.h>
#include <stdlib.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");
    printf("Calling function p()\n");
    p();
    printf("This point should never be reached\n");
}

void p(void)
{
    printf("Calling longjmp() from inside function p()\n");
    longjmp(mark, -1);
    printf("This point should never be reached\n");
}

void recover(void)
{
    printf("Performing function recover()\n");
}

/********************Output should be as follows: **********************/
setjmp has been called
Calling function p()
Calling longjmp() from inside function p()
longjmp has been called
Performing function recover()
****************************************************************************/

Related Information
• “setjmp() — Preserve Environment” on page 365
• “<setjmp.h>” on page 11

malloc() — Reserve Storage Block

Format

#include <stdlib.h>
void *malloc(size_t size);

Language Level

ANSI

Threadsafe

Yes

Description

The malloc() function reserves a block of storage of size bytes. Unlike the calloc() function, malloc() does not initialize all elements to 0. The maximum size for a non-teraspace malloc() is 16711568 bytes.
Note:

1. All heap storage is associated with the activation group of the calling function. As such, storage should be allocated, deallocated, and reallocated within the same activation group. You cannot allocate heap storage within one activation group and deallocate or reallocate that storage from a different activation group. For more information about activation groups, see the ILE Concepts manual.

2. To use teraspace storage instead of single-level store storage without changing the C source code, specify the TERASPACE(*YES *TSIFC) parameter on the compiler command. This maps the malloc() library function to _C_TS_malloc(), its teraspace storage counterpart. The maximum amount of teraspace storage that can be allocated by each call to _C_TS_malloc() is 2GB - 224, or 2147483424 bytes. If more than 2147483408 bytes are needed on a single request, call _C_TS_malloc64(unsigned long long int);.

   For more information, see the ILE Concepts manual.

3. For current statistics on the teraspace storage being used by MI programs in an activation group, call the _C_TS_malloc_info function. This function returns information including total bytes, allocated bytes and blocks, unallocated bytes and blocks, requested bytes, pad bytes, and overhead bytes. To get more detailed information about the memory structures used by the _C_TS_malloc() and _C_TS_malloc64() functions, call the _C_TS_malloc_debug() function. You can use the information this function returns to identify memory corruption problems.

4. If the Quick Pool memory manager has been enabled in the current activation group, then the storage is retrieved using Quick Pool memory manager. See “_C_Quickpool_Init() — Initialize Quick Pool Memory Manager” on page 93 for more information.

Return Value

The malloc() function returns a pointer to the reserved space. The storage space to which the return value points is suitably aligned for storage of any type of object. The return value is NULL if not enough storage is available, or if size was specified as zero.

Example

This example prompts 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.
```c
#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 */
        perror( "Out of storage" );
        abort();
    }
}

/*************** Output should be similar to: **************
Enter the size of the array
array[ 0 ] = 0
array[ 1 ] = 1
array[ 2 ] = 2
array[ 3 ] = 3
array[ 4 ] = 4
*/
```

### Related Information

- “calloc() — Reserve and Initialize Storage” on page 80
- “_C_Quickpool_Debug() — Modify Quick Pool Memory Manager Characteristics” on page 91
- “_C_Quickpool_Init() — Initialize Quick Pool Memory Manager” on page 93
- “_C_Quickpool_Report() — Generate Quick Pool Memory Manager Report” on page 95
- “free() — Release Storage Blocks” on page 151
- “realloc() — Change Reserved Storage Block Size” on page 291
- “Heap Memory” on page 578
- “<stdlib.h>” on page 15

---

### mblen() — Determine Length of a Multibyte Character

**Format**

```c
#include <stdlib.h>
int mblen(const char *string, size_t n);
```

**Language Level**

ANSI

**Threading Safe**

No
Use `mbrlen()` instead.

**Locale Sensitive**

The behavior of this function might be affected by the LC_CTYPE category of the current locale. This function might be affected by the LC_UNI_CTYPE category of the current locale if LOCALETYPE(*LOCALEUCS2) or LOCALETYPE(*LOCALEUTF) is specified on the compilation command. For more information, see “Understanding CCSIDs and Locales” on page 565.

**Description**

The `mblen()` function determines the length in bytes of the multibyte character pointed to by `string`. `n` represents the maximum number of bytes examined.

**Return Value**

If `string` is NULL, the `mblen()` function returns:
- Non-zero if the active locale allows mixed-byte strings. The function initializes the state variable.
- 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.
- -1 if `string` does not point to a valid multibyte character.

**Note:** The `mblen()`, `mbtowc()`, and `wctomb()` functions use their own statically allocated storage and are therefore not restartable. However, `mbrlen()`, `mbrtowc()`, and `wcrtomb()` are restartable.

**Example**

This example uses `mblen()` and `mbtowc()` to convert a multibyte character into a single wide character.

```c
#include <stdio.h>
#include <stdlib.h>

int length, temp;
char string[6] = "w";
wchar_t arr[6];

int main(void)
{
    /* Initialize internal state variable */
    length = mblen(NULL, MB_CUR_MAX);
    /* Set string to point to a multibyte character */
    length = mblen(string, MB_CUR_MAX);
    temp = mbtowc(arr,string,length);
    arr[1] = L'\0';
    printf("wide character string: %ls\n", arr);
}
```

**Related Information**

- “`mbrlen()` — Determine Length of a Multibyte Character (Restartable)” on page 221
- “`mbtowc()` — Convert Multibyte Character to a Wide Character” on page 233
- “`mbstowcs()` — Convert a Multibyte String to a Wide Character String” on page 229
- “`strlen()` — Determine String Length” on page 404
- “`wcslen()` — Calculate Length of Wide-Character String” on page 494
- “`wctomb()` — Convert Wide Character to Multibyte Character” on page 527
- “`<stdlib.h>`” on page 15

220 IBM i: ILE C/C++ Runtime Library Functions
**mbrlen() — Determine Length of a Multibyte Character (Restartable)**

**Format**

```c
#include <wchar.h>
size_t mbrlen (const char *s, size_t n, mbstate_t *ps);
```

**Language Level**

ANSI

**Threadsafe**

Yes, if `ps` is not NULL.

**Locale Sensitive**

The behavior of this function might be affected by the LC_CTYPE category of the current locale. This function might also be affected by the LC_UNI_CTYPE category of the current locale if LOCALETYPE("LOCALEUCS2") or LOCALETYPE("LOCALEUTF") is specified on the compilation command. This function is not available when LOCALETYPE("CLD") is specified on the compilation command. For more information, see “Understanding CCSIDs and Locales” on page 565.

**Description**

This function is the restartable version of mblen().

The `mbrlen()` function determines the length of a multibyte character.

`n` is the number of bytes (at most) of the multibyte string to examine.

This function differs from its corresponding internal-state multibyte character function in that it has an extra parameter, `ps` of type pointer to mbstate_t that points to an object that can completely describe the current conversion state of the associated multibyte character sequence. If `ps` is a NULL pointer, `mbrlen()` behaves like `mblen()`.

`mbrlen()` is a restartable version of `mblen()`. In other words, shift-state information is passed as one of the arguments (`ps` represents the initial shift) 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.

**Return Value**

If `s` is a null pointer and if the active locale allows mixed-byte strings, the `mbrlen()` function returns nonzero. If `s` is a null pointer and if the active locale does not allow mixed-byte strings, zero will be returned.

If `s` is not a null pointer, the `mbrlen()` function returns one of the following:

- **0**
  - If `s` is a NULL string (`s` points to the NULL character).

- **positive**
  - If the next `n` or fewer bytes comprise a valid multibyte character. The value returned is the number of bytes that comprise the multibyte character.

- **(size_t)-1**
  - If `s` does not point to a valid multibyte character.

- **(size_t)-2**
  - If the next `n` or fewer bytes contribute to an incomplete but potentially valid character and all `n` bytes have been processed.
Example

/* This program is compiled with LOCALETYPE(*LOCALE) and */
/* SYSSIFCOPT(*IFSIO)                                       */

#include <stdio.h>
#include <stdlib.h>
#include <locale.h>
#include <wchar.h>
#include <errno.h>
#define LOCNAME     "/qsys.lib/JA_JP.locale"
#define LOCNAME_EN  "/qsys.lib/EN_US.locale"

int main(void)
{
    int length, sl = 0;
    char string[10];
    mbstate_t ps = 0;
    memset(string, '\0', 10);
    string[0] = 0xC1;
    string[1] = 0x0E;
    string[2] = 0x41;
    string[3] = 0x71;
    string[4] = 0x41;
    string[5] = 0x72;
    string[6] = 0x0F;
    string[7] = 0xC2;

    /* In this first example we will find the length of */
    /* a multibyte character when the CCSID of locale */
    /* associated with LC_CTYPE is 37.                        */
    /* For single byte cases the state will always */
    /* remain in the initial state 0                         */
    if (setlocale(LC_ALL, LOCNAME_EN) == NULL)
        printf("setlocale failed.\n");
    length = mbrlen(string, MB_CUR_MAX, &ps);

    /* In this case length is 1, which is always the case for */
    /* single byte CCSID */
    printf("length = %d, state = %d\n", length, ps);
    printf("MB_CUR_MAX: %d\n", MB_CUR_MAX);

    sl += length;
    length = mbrlen(string + sl, MB_CUR_MAX, &ps);

    /* The next character is a mixed byte. Length is 3 to */
    /* account for the shiftout 0x0e. State is */
    /* changed to double byte state. */
    printf("length = %d, state = %d\n", length, ps);
    sl += length;
    length = mbrlen(string + sl, MB_CUR_MAX, &ps);

    /* The next character is also a double byte character. */
    /* The state is changed to initial state since this was */
    /* the last double byte character. Length is 3 to */
    /* account for the ending 0x0f shiftin. */
    printf("length = %d, state = %d\n", length, ps);
}
sl += length;
length = mbstrlen(&string[sl], MB_CUR_MAX, &ps);
/* The next character is single byte so length is 1 and */
/* state remains in initial state. */
printf("length = %d, state = %d\n\n", length, ps);
}
/* The output should look like this:
length = 1, state = 0
MB_CUR_MAX: 1
length = 1, state = 0
MB_CUR_MAX: 4
length = 3, state = 2
length = 3, state = 0
length = 1, state = 0
*/

Related Information
- “mblen() — Determine Length of a Multibyte Character” on page 219
- “mbtowc() — Convert Multibyte Character to a Wide Character” on page 233
- “mbtowc() — Convert a Multibyte Character to a Wide Character (Restartable)” on page 223
- “mbstowcs() — Convert a Multibyte String to a Wide Character String (Restartable)” on page 227
- “setlocale() — Set Locale” on page 366
- “wcrtomb() — Convert a Wide Character to a Multibyte Character (Restartable)” on page 478
- “wcsrtombs() — Convert Wide Character String to Multibyte String (Restartable)” on page 505
- “<locale.h>” on page 5
- “<wchar.h>” on page 16

mbtowc() — Convert a Multibyte Character to a Wide Character (Restartable)

Format
#include <wchar.h>
size_t mbtowc (wchar_t *pwc, const char *s, size_t n, mbstate_t *ps);

Language Level
ANSI

Threading
Yes, if ps is not NULL.

Locale Sensitive
The behavior of this function might be affected by the LC_CTYPE category of the current locale. This function might also be affected by the LC_UNI_CTYPE category of the current locale if LOCALETYPE(*LOCALEUCS2) or LOCALETYPE(*LOCALEUTF) is specified on the compilation command.
This function is not available when LOCALETYPE(*CLD) is specified on the compilation command. For more information, see “Understanding CCSIDs and Locales” on page 565.

Wide Character Function

See “Wide Characters” on page 568 for more information.

Description

This function is the restartable version of the mbrtowc() function.

If s is a null pointer, the mbrtowc() 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). In this situation, the value of the pwc parameter will be ignored and the resulting shift state described will be the initial conversion state.

If s is not a null pointer, the mbrtowc() function determines the number of bytes that are in the multibyte character (and any leading shift sequences) pointed to by s, produces the value of the corresponding multibyte character and if pwc is not a null pointer, stores that value in the object pointed to by pwc. If the corresponding multibyte character is the null wide character, the resulting state will be reset to the initial conversion state.

This function differs from its corresponding internal-state multibyte character function in that it has an extra parameter, ps of type pointer to mbstate_t that points to an object that can completely describe the current conversion state of the associated multibyte character sequence. If ps is NULL, this function uses an internal static variable for the state.

At most, n bytes of the multibyte string are examined.

Return Value

If s is a null pointer, the mbrtowc() function returns the number of bytes necessary to enter the initial shift state. The value returned must be less than the MB_CUR_MAX macro.

If a conversion error occurs, errno might be set to ECONVERT.

If s is not a null pointer, the mbrtowc() function returns one of the following:

0 If the next n or fewer bytes form the multibyte character that corresponds to the null wide character.

positive If the next n or fewer bytes form a valid multibyte character. The value returned is the number of bytes that constitute the multibyte character.

(size_t)-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 the value of the MB_CUR_MAX macro.

(size_t)-1 If an encoding error occurs (when the next n or fewer bytes do not form a complete and correct multibyte character). The value of the macro EILSEQ is stored in errno, but the conversion state is unchanged.

Note: When a -2 value is returned, the string could contain redundant shift-out and shift-in characters or a partial UTF-8 character. To continue processing the multibyte string, increment the pointer by the value n, and call mbrtowc() again.

Example

/* This program is compiled with LOCALETYPE(*LOCALE) and */
/* SYSIFCOPT(*IFSIO) */
#include <stdio.h>
#include <stdlib.h>
```c
#include <locale.h>
#include <wchar.h>
#include <errno.h>

#define LOCNAME     "/qsys.lib/JA_JP.locale"
#define LOCNAME_EN  "/qsys.lib/EN_US.locale"

int main(void)
{
    int length, sl = 0;
    char string[10];
    wchar_t buffer[10];
    mbstate_t ps = 0;
    memset(string, 0, 10);
    string[0] = 0xC1;
    string[1] = 0x0E;
    string[2] = 0x41;
    string[3] = 0x71;
    string[4] = 0x41;
    string[5] = 0x72;
    string[6] = 0x0F;
    string[7] = 0xC2;

    /* In this first example we will convert */
    /* a multibyte character when the CCSID of locale */
    /* associated with LC_CTYPE is 37. */
    /* For single byte cases the state will always */
    /* remain in the initial state 0 */
    if (setlocale(LC_ALL, LOCNAME_EN) == NULL)
        printf("setlocale failed.\n");

    length = mbrtowc(buffer, string, MB_CUR_MAX, &ps);
    /* In this case length is 1, and C1 is converted 0x00C1 */
    printf("length = %d, state = %d\n", length, ps);
    printf("MB_CUR_MAX: %d\n", MB_CUR_MAX);

    if (setlocale(LC_ALL, LOCNAME) == NULL)
        printf("setlocale failed.\n");

    length = mbrtowc(buffer, string, MB_CUR_MAX, &ps);
    /* The first is single byte so length is 1 and */
    /* the state is still the initial state 0. C1 is converted*/
    /* to 0x00C1 */
    printf("length = %d, state = %d\n", length, ps);
    printf("MB_CUR_MAX: %d\n", MB_CUR_MAX);

    sl += length;

    length = mbrtowc(&buffer[1], &string[sl], MB_CUR_MAX, &ps);
    /* The next character is a mixed byte. Length is 3 to */
    /* account for the shiftout 0x0e. State is */
    /* changed to double byte state. 0x4171 is copied into */
    /* the buffer */
    printf("length = %d, state = %d\n", length, ps);
    sl += length;

    length = mbrtowc(&buffer[2], &string[sl], MB_CUR_MAX, &ps);
    /* The next character is also a double byte character. */
    /* The state is changed to initial state since this was */
    /* the last double byte character. Length is 3 to */
    /* account for the ending 0x0f shiftin. 0x4172 is copied */
    /* into the buffer. */
    printf("length = %d, state = %d\n", length, ps);
    sl += length;

    length = mbrtowc(&buffer[3], &string[sl], MB_CUR_MAX, &ps);
    /* The next character is single byte so length is 1 and */
```
/* state remains in initial state. 0xC2 is converted to 0x00C2. The buffer now has the value: */
/* 0x00C14171417200C2 */
printf("length = %d, state = %d\n", length, ps);
}
} /* The output should look like this: */
length = 1, state = 0
MB_CUR_MAX: 1
length = 1, state = 0
MB_CUR_MAX: 4
length = 3, state = 2
length = 3, state = 0
length = 1, state = 0

Related Information
• “mblen() — Determine Length of a Multibyte Character” on page 219
• “mbrlen() — Determine Length of a Multibyte Character (Restartable)” on page 221
• “mbsrtowcs() — Convert a Multibyte String to a Wide Character String (Restartable)” on page 227
• “setlocale() — Set Locale” on page 366
• “wcrtomb() — Convert a Wide Character to a Multibyte Character (Restartable)” on page 478
• “wcsrtombs() — Convert Wide Character String to Multibyte String (Restartable)” on page 505
• “<locale.h>” on page 5
• “<wchar.h>” on page 16

mbsinit() — Test State Object for Initial State

Format
#include <wchar.h>
int mbsinit (const mbstate_t *ps);

Language Level
ANSI

Threadsafe
Yes

Locale Sensitive
The behavior of this function might be affected by the LC_CTYPE category of the current locale. This function is not available when LOCALETYPER(CLD) is specified on the compilation command. For more information, see “Understanding CCSIDs and Locales” on page 565.

Description
If ps is not a null pointer, the mbsinit() function specifies whether the pointed to mbstate_t object describes an initial conversion state.
Return Value
The `mbsinit()` function returns nonzero if `ps` is a null pointer or if the pointed to object describes an initial conversion state. Otherwise, it returns zero.

Example
This example checks the conversion state to see if it is the initial state.

```c
#include <stdio.h>
#include <wchar.h>
#include <stdlib.h>

int 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
- “mbrlen() — Determine Length of a Multibyte Character (Restartable)” on page 221
- “mbrtoc() — Convert a Multibyte Character to a Wide Character (Restartable)” on page 223
- “mbsrtowcs() — Convert a Multibyte String to a Wide Character String (Restartable)” on page 227
- “setlocale() — Set Locale” on page 366
- “wcrtomb() — Convert a Wide Character to a Multibyte Character (Restartable)” on page 478
- “wcsrtombs() — Convert Wide Character String to Multibyte String (Restartable)” on page 505
- "<locale.h>” on page 5
- "<wchar.h>” on page 16

**mbsrtowcs() — Convert a Multibyte String to a Wide Character String (Restartable)**

Format
```c
#include <wchar.h>
size_t mbsrtowcs (wchar_t *dst, const char **src, size_t len, mbstate_t *ps);
```

Language Level
ANSI

Threading
Yes, if `ps` is not NULL.

Locale Sensitive
The behavior of this function might be affected by the LC_CTYPE category of the current locale. This function might also be affected by the LC_UNI_CTYPE category of the current locale if LOCALETYPE(*LOCALEUCS2) or LOCALETYPE(*LOCALEUTF) is specified on the compilation command.
This function is not available when LOCALETYPE(*CLD) is specified on the compilation command. For more information, see “Understanding CCSIDs and Locales” on page 565.

**Wide Character Function**

See “Wide Characters” on page 568 for more information.

**Description**

This function is the restartable version of mbstowcs().

The mbstowcs() function converts a sequence of multibyte characters that begins in the conversion state described by ps from the array indirectly pointed to by src into a sequence of corresponding wide characters. It then stores the converted characters into the array pointed to by dst.

Conversion continues up to and including an ending null character, which is also stored. Conversion will stop earlier in two cases: when a sequence of bytes are reached that do not form a valid multibyte character, or (if dst is not a null pointer) when len wide characters have been stored into the array pointed to by dst. Each conversion takes place as if by a call to mbrtowc() function.

If dst is not a null pointer, the pointer object pointed to by src will be assigned either a null pointer (if conversion stopped due to reaching an ending null character) or the address just past the last multibyte character converted. If conversion stopped due to reaching an ending null character, the initial conversion state is described.

**Return Value**

If the input string does not begin with a valid multibyte character, an encoding error occurs, the mbstowcs() function stores the value of the macro EILSEQ in errno, and returns (size_t) -1, but the conversion state will be unchanged. Otherwise, it returns the number of multibyte characters successfully converted, which is the same as the number of array elements modified when dst is not a null pointer.

If a conversion error occurs, errno might be set to ECONVERT.
Example

```c
#include <stdio.h>
#include <stdlib.h>
#include <wchar.h>
#include <locale.h>

#define SIZE 10

int main(void)
{
    char           mbs1[] = "abc";
    char           mbs2[] = "\x81\x41" "m" "\x81\x42";
    const char    *pmbs1 = mbs1;
    const char    *pmbs2 = mbs2;
    mbstate_t      ss1 = 0;
    mbstate_t      ss2 = 0;
    wchar_t        wcs1[SIZE], wcs2[SIZE];

    if (NULL == setlocale(LC_ALL, "/qsys.lib/locale.lib/ja_jp939.locale"))
    {
        printf("setlocale failed.\n");
        exit(EXIT_FAILURE);
    }
    mbsrtowcs(wcs1, &pmbs1, SIZE, &ss1);
    mbsrtowcs(wcs2, &pmbs2, SIZE, &ss2);
    printf("The first wide character string is %ls.\n", wcs1);
    printf("The second wide character string is %ls.\n", wcs2);
    return 0;
}
```

Also, see the examples for “mbtowc() — Convert a Multibyte Character to a Wide Character (Restartable)” on page 223.

Related Information

- “mblen() — Determine Length of a Multibyte Character” on page 219
- “mbrlen() — Determine Length of a Multibyte Character (Restartable)” on page 221
- “mbtowc() — Convert a Multibyte Character to a Wide Character (Restartable)” on page 223
- “mbstowcs() — Convert a Multibyte String to a Wide Character String” on page 229
- “setlocale() — Set Locale” on page 366
- “wcrtomb() — Convert a Wide Character to a Multibyte Character (Restartable)” on page 478
- “wcsrtombs() — Convert Wide Character String to Multibyte String (Restartable)” on page 505
- “<locale.h>” on page 5
- “<wchar.h>” on page 16

**mbstowcs() — Convert a Multibyte String to a Wide Character String**

**Format**

```c
#include <stdlib.h>
size_t mbstowcs(wchar_t *pwc, const char *string, size_t n);
```

**Language Level**

ANSI
Threadsafe
Yes

Locale Sensitive
The behavior of this function might be affected by the LC_CTYPE category of the current locale. This function might also be affected by the LC_UNI_CTYPE category of the current locale if LOCALETYP(*LOCALEUCS2) or LOCALETYP(*LOCALEUTF) is specified on the compilation command. For more information, see “Understanding CCSIDs and Locales” on page 565.

Wide Character Function
See “Wide Characters” on page 568 for more information.

Description
The mbstowcs() function determines the length of the sequence of the multibyte characters pointed to by string. It then converts the multibyte character string that begins in the initial shift state into a wide character string, and stores the wide characters into the buffer that is pointed to by pwc. A maximum of n wide characters are written.

Return Value
The mbstowcs() function returns the number of wide characters generated, not including any ending null wide characters. If a multibyte character that is not valid is encountered, the function returns (size_t)-1.
If a conversion error occurs, errno might be set to ECONVERT.
Examples

```c
/* This program is compiled with LOCALETYPE(*LOCALEUCS2) and */
/* SYSIFCOPT(*IFSID)                                           */
#include <stdio.h>
#include <stdlib.h>
#include <locale.h>
#include <wchar.h>
#include <errno.h>
#define LOCNAME     "/qsys.lib/JA_JP.locale"
#define LOCNAME_EN  "/qsys.lib/EN_US.locale"

int main(void)
{
    int length, sl = 0;
    char string[10];
    char string2[] = "ABC";
    wchar_t buffer[10];
    memset(string, '\0', 10);
    string[0] = 0xC1;
    string[1] = 0x0E;
    string[2] = 0x41;
    string[3] = 0x71;
    string[4] = 0x41;
    string[5] = 0x72;
    string[6] = 0x0F;
    string[7] = 0xC2;
    /* In this first example we will convert */
    /* a multibyte character when the CCSID of locale */
    /* associated with LC_CTYPE is 37. */
    if (setlocale(LC_ALL, LOCNAME_EN) == NULL)
        printf("setlocale failed.\n")
            printf("length = %d\n\n", length);

    length = mbstowcs(buffer, string2, 10);
    /* In this case length ABC is converted to UNICODE ABC */
    /* or 0x004100420043. Length will be 3. */
    printf("length = %d\n\n", length);
    /* Now lets try a multibyte example. We first must set the */
    /* locale to a multibyte locale. We choose a locale with */
    /* CCSID 5026 */
    if (setlocale(LC_ALL, LOCNAME) == NULL)
        printf("setlocale failed.\n")
            length = mbstowcs(buffer, string, 10);
    /* The buffer now has the value: */
    /* 0x004103A103A30042        length is 4 */
    printf("length = %d\n\n", length);
}
/* The output should look like this:
length = 3
length = 4 */
```

Library Functions 231
/* This program is compiled with LOCALETYPE(*LOCALE) and SYSIFCOPT(*IFSIO) */

#include <stdio.h>
#include <stdlib.h>
#include <locale.h>
#include <wchar.h>
#include <errno.h>

#define LOCNAME     "/qsys.lib/JA_JP.locale"
#define LOCNAME_EN  "/qsys.lib/EN_US.locale"

int main(void)
{
  int length, sl = 0;
  char string[10];
  char string2[] = "ABC";
  wchar_t buffer[10];
  memset(string, '\0', 10);
  string[0] = 0xC1;
  string[1] = 0x0E;
  string[2] = 0x41;
  string[3] = 0x71;
  string[4] = 0x41;
  string[5] = 0x72;
  string[6] = 0x0F;
  string[7] = 0xC2;
  /* In this first example we will convert */
  /* a multibyte character when the CCSID of locale */
  /* associated with LC_CTYPE is 37. */
  if (setlocale(LC_ALL, LOCNAME_EN) == NULL)
    printf("setlocale failed.\n");
  length = mbstowcs(buffer, string2, 10);
  /* In this case length ABC is converted to */
  /* 0x00C100C200C3. Length will be 3. */
  printf("length = %d\n", length);
  /* Now lets try a multibyte example. We first must set the */
  /* locale to a multibyte locale. We choose a locale with */
  /* CCSID 5026 */
  if (setlocale(LC_ALL, LOCNAME) == NULL)
    printf("setlocale failed.\n");
  length = mbstowcs(buffer, string, 10);
  /* The buffer now has the value: */
  /* 0x00C14171417200C2 */
  /* length is 4 */
  printf("length = %d\n", length);
}
/* The output should look like this:
length = 3
length = 4 */

Related Information
• “mblen() — Determine Length of a Multibyte Character” on page 219
• “mbtowc() — Convert Multibyte Character to a Wide Character” on page 233
• “setlocale() — Set Locale” on page 366
• “wcslen() — Calculate Length of Wide-Character String” on page 494
• “wcstombs() — Convert Wide-Character String to Multibyte String” on page 517
• “<locale.h>” on page 5
• “<stdlib.h>” on page 15
mbtowc() — Convert Multibyte Character to a Wide Character

Format

```c
#include <stdlib.h>
int mbtowc(wchar_t *pwc, const char *string, size_t n);
```

Language Level

ANSI

Threadsafe

No

Use mbtowc() instead.

Locale Sensitive

The behavior of this function might be affected by the LC_CTYPE category of the current locale. This function might also be affected by the LC_UNI_CTYPE category of the current locale if LOCALETYPE(*LOCALEUCS2) or LOCALETYPE(*LOCALEUTF) is specified on the compilation command. For more information, see “Understanding CCSIDs and Locales” on page 565.

Wide Character Function

See “Wide Characters” on page 568 for more information.

Description

The mbtowc() function first determines the length of the multibyte character pointed to by string. It then converts the multibyte character to a wide character as described in mbstowcs. A maximum of n bytes are examined.

Return Value

If string is NULL, the mbtowc() function returns:

- Nonzero when the active locale is mixed byte. The function initializes the state variable.
- 0 otherwise.

If string is not NULL, the mbtowc() function returns:

- 0 if string points to the null character
- The number of bytes comprising the converted multibyte character
- -1 if string does not point to a valid multibyte character.

If a conversion error occurs, errno might be set to ECONVERT.

Example

This example uses the mblen() and mbtowc() functions to convert a multibyte character into a single wide character.
#include <stdio.h>
#include <stdlib.h>
#define LOCNAME "/qsys.lib/mylib.lib/ja_jp959.locale
/*Locale created from source JA_JP and CCSID 939 */
int length, temp;
char string [] = "\x0e\x41\x71\x0f";
wchar_t arr[6];

int main(void)
{
    /* initialize internal state variable */
    temp = mbtowc(arr, NULL, 0);
    setlocale (LC_ALL, LOCNAME);
    /* Set string to point to a multibyte character. */
    length = mblen(string, MB_CUR_MAX);
    temp = mbtowc(arr,string,length);
    arr[1] = L'\0';
    printf("wide character string: %ls",arr);
}

Related Information
• “mblen() — Determine Length of a Multibyte Character” on page 219
• “mbstowcs() — Convert a Multibyte String to a Wide Character String” on page 229
• “wcslen() — Calculate Length of Wide-Character String” on page 494
• “wctomb() — Convert Wide Character to Multibyte Character” on page 527
• “<stdlib.h>” on page 15

memchr() — Search Buffer

Format

```
#include <string.h>
void *memchr(const void *buf, int c, size_t count);
```

Language Level
ANSI

Threading
Yes

Description
The memchr() function searches the first count bytes of buf for the first occurrence of c converted to an unsigned character. The search continues until it finds c or examines count bytes.

Return Value
The memchr() function returns a pointer to the location of c in buf. It returns NULL if c is not within the first count bytes of buf.

Example
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.
#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" );
    }
} /*
The string starting with x is xing */

Related Information

• “memcmp() — Compare Buffers” on page 235
• “memcpy() — Copy Bytes” on page 236
• “memmove() — Copy Bytes” on page 239
• “wmemchr() — Locate Wide Character in Wide-Character Buffer” on page 532
• “memset() — Set Bytes to Value” on page 240
• “strchr() — Search for Character” on page 386
• “<string.h>” on page 15

memcmp() — Compare Buffers

Format

#include <string.h>
int memcmp(const void *buf1, const void *buf2, size_t count);

Language Level

ANSI

Threading

Yes

Description

The memcmp() function compares the first count bytes of buf1 and buf2.

Return Value

The memcmp() function returns a value indicating the relationship between the two buffers as follows:

<table>
<thead>
<tr>
<th>Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>Less than 0</td>
<td>buf1 less than buf2</td>
</tr>
<tr>
<td>0</td>
<td>buf1 identical to buf2</td>
</tr>
<tr>
<td>Value</td>
<td>Meaning</td>
</tr>
<tr>
<td>--------------------</td>
<td>--------------------------</td>
</tr>
<tr>
<td>Greater than 0</td>
<td>buf1 greater than buf2</td>
</tr>
</tbody>
</table>

**Example**

This example compares first and second arguments passed to `main()` to determine which, if either, is greater.

```c
#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 */
            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] );
    }
}
```

Related Information

- “memchr() — Search Buffer” on page 234
- “memcpy() — Copy Bytes” on page 236
- “wmemcmp() — Compare Wide-Character Buffers” on page 533
- “memmove() — Copy Bytes” on page 239
- “memset() — Set Bytes to Value” on page 240
- “strcmp() — Compare Strings” on page 388
- “<string.h>” on page 15

**memcpy() — Copy Bytes**

**Format**

```c
#include <string.h>
void *memcpy(void *dest, const void *src, size_t count);
```
Language Level
ANSI

Threadsafe
Yes

Description
The `memcpy()` function copies `count` bytes of `src` to `dest`. The behavior is undefined if copying takes place between objects that overlap. The `memmove()` function allows copying between objects that might overlap.

Return Value
The `memcpy()` function returns a pointer to `dest`.

Example
This example copies the contents of source to target.

```c
#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 );
}
/******************** Expected output: ********************
Before memcpy, target is "This is the target string"
After memcpy, target becomes "This is the source string" +/
```

Related Information
- “`memchr()` — Search Buffer” on page 234
- “`memcmp()` — Compare Buffers” on page 235
- “`wmemcpy()` — Copy Wide-Character Buffer” on page 535
- “`memmove()` — Copy Bytes” on page 239
- “`memset()` — Set Bytes to Value” on page 240
- “`strcpy()` — Copy Strings” on page 392
- “`<string.h>`” on page 15

`memicmp()` — Compare Bytes

Format
```c
#include <string.h>    // also in <memory.h>
int memicmp(void *buf1, void *buf2, unsigned int cnt);
```

Note: The `memicmp` function is available for C++ programs. It is available for C only when the program defines the `__cplusplus__strings__` macro.
Language Level

Extension

Threadsafe

Yes

Locale Sensitive

The behavior of this function might be affected by the LC_CTYPE category of the current locale. For more information, see “Understanding CCSIDs and Locales” on page 565.

Description

The `memicmp` function compares the first `cnt` bytes of `buf1` and `buf2` without regard to the case of letters in the two buffers. The function converts all uppercase characters into lowercase and then performs the comparison.

Return Value

The return value of `memicmp` indicates the result as follows:

<table>
<thead>
<tr>
<th>Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>Less than 0</td>
<td><code>buf1</code> less than <code>buf2</code></td>
</tr>
<tr>
<td>0</td>
<td><code>buf1</code> identical to <code>buf2</code></td>
</tr>
<tr>
<td>Greater than 0</td>
<td><code>buf1</code> greater than <code>buf2</code></td>
</tr>
</tbody>
</table>

Example

This example copies two strings that each contain a substring of 29 characters that are the same except for case. The example then compares the first 29 bytes without regard to case.

```c
#include <stdio.h>
#include <string.h>
char first[100], second[100];
int main(void)
{
    int result;
    strcpy(first, "Those Who Will Not Learn From History");
    strcpy(second, "THOSE WHO WILL NOT LEARN FROM their mistakes");
    printf("Comparing the first 29 characters of two strings.\n");
    result = memicmp(first, second, 29);
    printf("The first 29 characters of String 1 are ");
    if (result < 0)
        printf("less than String 2.\n");
    else if (0 == result)
        printf("equal to String 2.\n");
    else
        printf("greater than String 2.\n");
    return 0;
}
```

The output should be:

Comparing the first 29 characters of two strings.
The first 29 characters of String 1 are equal to String 2

Related Information

- “memchr() — Search Buffer” on page 234
memmove() — Copy Bytes

Format

```c
#include <string.h>
void *memmove(void *dest, const void *src, size_t count);
```

Language Level

ANSI

Threading

Yes

Description

The `memmove()` function copies `count` bytes of `src` to `dest`. This function allows copying between objects that might overlap as if `src` is first copied into a temporary array.

Return Value

The `memmove()` function returns a pointer to `dest`.

Example

This example copies the word "shiny" from position target + 2 to position target + 8.

```c
#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 );
}
```

Expected output:

```
Before memmove, target is "a shiny white sphere"
After memmove, target becomes "a shiny shiny sphere"
```
memset() — Set Bytes to Value

Format

```c
#include <string.h>
void *memset(void *dest, int c, size_t count);
```

Language Level

ANSI

Threadsafe

Yes

Description

The `memset()` function sets the first `count` bytes of `dest` to the value `c`. The value of `c` is converted to an unsigned character.

Return Value

The `memset()` function returns a pointer to `dest`.

Example

This example sets 10 bytes of the buffer to A and the next 10 bytes to B.

```c
#include <string.h>
#include <stdio.h>
#define BUF_SIZE 20
int main(void)
{
    char buffer[BUF_SIZE + 1];
    char *string;
    memset(buffer, 0, sizeof(buffer));
    string = (char *) memset(buffer,'A', 10);
    printf("\nBuffer contents: %s\n", string);
    memset(buffer+10, 'B', 10);
    printf("\nBuffer contents: %s\n", buffer);
}

/******************** Output should be similar to: ********************
Buffer contents: AAAAAAAAAAAAAAA
Buffer contents: AAAAAAAAAABBBBBBBBB
 */
```
Related Information

- “memchr() — Search Buffer” on page 234
- “memcmp() — Compare Buffers” on page 235
- “memcpy() — Copy Bytes” on page 236
- “memmove() — Copy Bytes” on page 239
- “wmemset() — Set Wide Character Buffer to a Value” on page 537
- “<string.h>” on page 15

### mktime() — Convert Local Time

#### Format

```c
#include <time.h>
time_t mktime(struct tm *time);
```

#### Language Level

ANSI

#### Threadsafe

Yes

#### Locale Sensitive

The behavior of this function might be affected by the LC_TOD category of the current locale.

#### Description

The `mktime()` function converts a stored `tm` structure (assume to be in job local time) pointed to by `time`, into a `time_t` structure suitable for use with other time functions. After the conversion, the `time_t` structure will be considered Universal Coordinate Time (UTC). For this conversion, `mktime()` checks the current locale setting for local time zone and daylight saving time (DST). If these values are not set in the current locale, `mktime()` gets the local time zone and daylight saving time settings from the current job. If the DST is set in the locale but the time zone information is not, the DST information in the locale is ignored. `mktime()` then uses the current time zone information to determine UTC.

The values of some structure elements pointed to by `time` are not restricted to the ranges shown for `gmtime()`.

The values of `tm_wday` and `tm_yday` passed to `mktime()` are ignored and are assigned their correct values on return.

A positive or 0 value for `tm_isdst` passed to `mktime()` causes `mktime()` to presume initially that DST, respectively, is or is not in effect for the specified time. A negative value for `tm_isdst` passed to `mktime()` causes `mktime()` to attempt to determine whether DST is in effect for the specified time.

#### Return Value

The `mktime()` function returns Universal Coordinate Time (UTC) having type `time_t`. The value `(time_t)(-1)` is returned if the Universal Coordinate Time cannot be represented.

#### Example

This example prints the day of the week that is 40 days and 16 hours from the current date.
```c
#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 should be similar to:

```
40 days and 16 hours from now, it will be a Sunday
```
Threadsafe
Yes

Locale Sensitive
The behavior of this function might be affected by the LC_TOD category of the current locale.

Description
The `mktime64()` function converts a stored `tm` structure (assumed to be in job local time) pointed to by `time`, into a `time64_t` value suitable for use with other time functions. After the conversion, the `time64_t` value will be considered Universal Coordinate Time (UTC). For this conversion, `mktime64()` checks the current locale settings for the local time zone and daylight saving time (DST). If these values are not set in the current locale, `mktime64()` gets the local time zone and DST settings from the current job. If the DST is set in the locale but the time zone information is not, the DST information in the locale is ignored. The `mktime64()` function then uses the time zone information of the current job to determine UTC.

The values of some structure elements pointed to by `time` are not restricted to the ranges shown for `gmtime64()`.

The values of `tm_wday` and `tm_yday` passed to `mktime64()` are ignored and are assigned their correct values on return.

A positive or 0 value for `tm_isdst` causes `mktime()` to presume initially that DST, respectively, is or is not in effect for the specified time. A negative value for `tm_isdst` causes `mktime()` to attempt to determine whether DST is in effect for the specified time.

Note: The supported date and time range for this function is 01/01/1970 00:00:00 through 12/31/9999 23:59:59.

Return Value
The `mktime64()` function returns Universal Coordinate Time (UTC) having type `time64_t`. The value `(time_t)(-1)` is returned if the Universal Coordinate Time cannot be represented or if the given `time` is out of range. If the given `time` is out of range, `errno` is set to `EOVERFLOW`.

Example
This example prints the day of the week that is 40 days and 16 hours from the current date.

```c
#include <stdio.h>
#include <time.h>

char *wday[] = { "Sunday", "Monday", "Tuesday", "Wednesday", "Thursday", "Friday", "Saturday" };

int main(void)
{
    time64_t t1, t3;
    struct tm *t2;

    t1 = time64(NULL);
    t2 = localtime64(&t1);
    t2 -> tm_mday += 40;
    t2 -> tm_hour += 16;
    t3 = mktime64(t2);

    printf("40 days and 16 hours from now, it will be a %s \n",
           wday[(t2 -> tm_wday)];
}

/********************  Output should be similar to:  *****************/
40 days and 16 hours from now, it will be a Sunday
*/
modf() — Separate Floating-Point Value

Format

```c
#include <math.h>
double modf(double x, double *intptr);
```

Language Level

ANSI

Threading Safe

Yes

Description

The `modf()` function breaks down the floating-point value `x` into fractional and integral parts. The signed fractional portion of `x` is returned. The integer portion is stored as a double value pointed to by `intptr`. Both the fractional and integral parts are given the same sign as `x`.

Return Value

The `modf()` function returns the signed fractional portion of `x`.

Example

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 should be similar to:  *********************/
x = -14.876000
Integral part = -14.000000
Fractional part = -0.876000
*/

Related Information
• “fmod() — Calculate Floating-Point Remainder” on page 133
• “frexp() — Separate Floating-Point Value” on page 154
• “ldexp() — Multiply by a Power of Two” on page 200
• “<math.h>” on page 6

nextafter() – nextafterl() – nexttoward() – nexttowardl() — Calculate the Next Representable Floating-Point Value

Format
#include <math.h>
double nextafter(double x, double y);
long double nextafterl(long double x, long double y);
double nexttoward(double x, long double y);
long double nexttowardl(long double x, long double y);

Language Level
ANSI

Threading
Yes

Description
The nextafter(), nextafterl(), nexttoward(), and nexttowardl() functions calculate the next representable value after x in the direction of y.

Return Value
The nextafter(), nextafterl(), nexttoward(), and nexttowardl() functions return the next representable value after x in the direction of y. If x is equal to y, they return y. If x or y is NaN (Not a Number), NaN is returned and errno is set to EDOM. If x is the largest finite value and the result is infinite or not representable, HUGE_VAL is returned and errno is set to ERANGE.
Example
This example converts a floating-point value to the next greater representable value and next smaller representable value. It prints out the converted values.

```c
#include <stdio.h>
#include <math.h>
int main(void)
{
  double x, y;
  long double ld;

  x = nextafter(1.234567899, 10.0);
  printf("nextafter 1.234567899 is %#19.17g\n" x);
  ld = nextafterl(1.234567899, -10.0);
  printf("nextafterl 1.234567899 is %#19.17g\n" ld);

  x = nexttoward(1.234567899, 10.0);
  printf("nexttoward 1.234567899 is %#19.17g\n" x);
  ld = nexttowardl(1.234567899, -10.0);
  printf("nexttowardl 1.234567899 is %#19.17g\n" ld);
}
/******************** Output should be similar to: ******************
nextafter 1.234567899 is  1.2345678990000002
nextafterl 1.234567899 is 1.2345678989999997
nexttoward 1.234567899 is 1.2345678990000002
nexttowardl 1.234567899 is 1.2345678989999997
*/
```

Related Information
- “ceil() — Find Integer >=Argument” on page 86
- “floor() — Find Integer <=Argument” on page 132
- “frexp() — Separate Floating-Point Value” on page 154
- “modf() — Separate Floating-Point Value” on page 244
- “<math.h>” on page 6

nl_langinfo() — Retrieve Locale Information

Format
```
#include <langinfo.h>
#include <nl_types.h>
char *nl_langinfo(nl_item item);
```

Language Level
XPG4

Threadsafe
No

Locale Sensitive
The behavior of this function might be affected by the LC_CTYPE, LC_MESSAGES, LC_MONETARY, LC_NUMERIC, and LC_TIME categories of the current locale. This function is not available when LOCALETYPE(*CLD) is specified on the compilation command. For more information, see “Understanding CCSIDs and Locales” on page 565.
Description

The `nl_langinfo()` function retrieves from the current locale the string that describes the requested information specified by *item*.

The retrieval of the following information from the current locale is supported:

<table>
<thead>
<tr>
<th>Item</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>CODESET</td>
<td>CCSID of locale in character form</td>
</tr>
<tr>
<td>D_T_FMT</td>
<td>string for formatting date and time</td>
</tr>
<tr>
<td>D_FMT</td>
<td>date format string</td>
</tr>
<tr>
<td>T_FMT</td>
<td>time format string</td>
</tr>
<tr>
<td>T_FMT_AMPM</td>
<td>a.m. or p.m. time format string</td>
</tr>
<tr>
<td>AM_STR</td>
<td>Ante Meridian affix</td>
</tr>
<tr>
<td>PM_STR</td>
<td>Post Meridian affix</td>
</tr>
<tr>
<td>DAY_1</td>
<td>name of the first day of the week (for example, Sunday)</td>
</tr>
<tr>
<td>DAY_2</td>
<td>name of the second day of the week (for example, Monday)</td>
</tr>
<tr>
<td>DAY_3</td>
<td>name of the third day of the week (for example, Tuesday)</td>
</tr>
<tr>
<td>DAY_4</td>
<td>name of the fourth day of the week (for example, Wednesday)</td>
</tr>
<tr>
<td>DAY_5</td>
<td>name of the fifth day of the week (for example, Thursday)</td>
</tr>
<tr>
<td>DAY_6</td>
<td>name of the sixth day of the week (for example, Friday)</td>
</tr>
<tr>
<td>DAY_7</td>
<td>name of the seventh day of the week (for example, Saturday)</td>
</tr>
<tr>
<td>ABDAY_1</td>
<td>abbreviated name of the first day of the week</td>
</tr>
<tr>
<td>ABDAY_2</td>
<td>abbreviated name of the second day of the week</td>
</tr>
<tr>
<td>ABDAY_3</td>
<td>abbreviated name of the third day of the week</td>
</tr>
<tr>
<td>ABDAY_4</td>
<td>abbreviated name of the fourth day of the week</td>
</tr>
<tr>
<td>ABDAY_5</td>
<td>abbreviated name of the fifth day of the week</td>
</tr>
<tr>
<td>ABDAY_6</td>
<td>abbreviated name of the sixth day of the week</td>
</tr>
<tr>
<td>ABDAY_7</td>
<td>abbreviated name of the seventh day of the week</td>
</tr>
<tr>
<td>MON_1</td>
<td>name of the first month of the year</td>
</tr>
<tr>
<td>MON_2</td>
<td>name of the second month of the year</td>
</tr>
<tr>
<td>MON_3</td>
<td>name of the third month of the year</td>
</tr>
<tr>
<td>MON_4</td>
<td>name of the fourth month of the year</td>
</tr>
<tr>
<td>MON_5</td>
<td>name of the fifth month of the year</td>
</tr>
<tr>
<td>MON_6</td>
<td>name of the sixth month of the year</td>
</tr>
<tr>
<td>MON_7</td>
<td>name of the seventh month of the year</td>
</tr>
<tr>
<td>MON_8</td>
<td>name of the eighth month of the year</td>
</tr>
<tr>
<td>MON_9</td>
<td>name of the ninth month of the year</td>
</tr>
<tr>
<td>MON_10</td>
<td>name of the tenth month of the year</td>
</tr>
<tr>
<td>MON_11</td>
<td>name of the eleventh month of the year</td>
</tr>
<tr>
<td>Item</td>
<td>Explanation</td>
</tr>
<tr>
<td>----------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>MON_12</td>
<td>name of the twelfth month of the year</td>
</tr>
<tr>
<td>ABMON_1</td>
<td>abbreviated name of the first month of the year</td>
</tr>
<tr>
<td>ABMON_2</td>
<td>abbreviated name of the second month of the year</td>
</tr>
<tr>
<td>ABMON_3</td>
<td>abbreviated name of the third month of the year</td>
</tr>
<tr>
<td>ABMON_4</td>
<td>abbreviated name of the fourth month of the year</td>
</tr>
<tr>
<td>ABMON_5</td>
<td>abbreviated name of the fifth month of the year</td>
</tr>
<tr>
<td>ABMON_6</td>
<td>abbreviated name of the sixth month of the year</td>
</tr>
<tr>
<td>ABMON_7</td>
<td>abbreviated name of the seventh month of the year</td>
</tr>
<tr>
<td>ABMON_8</td>
<td>abbreviated name of the eighth month of the year</td>
</tr>
<tr>
<td>ABMON_9</td>
<td>abbreviated name of the ninth month of the year</td>
</tr>
<tr>
<td>ABMON_10</td>
<td>abbreviated name of the tenth month of the year</td>
</tr>
<tr>
<td>ABMON_11</td>
<td>abbreviated name of the eleventh month of the year</td>
</tr>
<tr>
<td>ABMON_12</td>
<td>abbreviated name of the twelfth month of the year</td>
</tr>
<tr>
<td>ERA</td>
<td>era description segments</td>
</tr>
<tr>
<td>ERA_D_FMT</td>
<td>era date format string</td>
</tr>
<tr>
<td>ERA_D_T_FMT</td>
<td>era date and time format string</td>
</tr>
<tr>
<td>ERA_T_FMT</td>
<td>era time format string</td>
</tr>
<tr>
<td>ALT_DIGITS</td>
<td>alternative symbols for digits</td>
</tr>
<tr>
<td>RADIXCHAR</td>
<td>radix character</td>
</tr>
<tr>
<td>THOUSEP</td>
<td>separator for thousands</td>
</tr>
<tr>
<td>YESEXPR</td>
<td>affirmative response expression</td>
</tr>
<tr>
<td>NOEXPR</td>
<td>negative response expression</td>
</tr>
<tr>
<td>YESSTR</td>
<td>affirmative response for yes/no queries</td>
</tr>
<tr>
<td>NOSTR</td>
<td>negative response for yes/no queries</td>
</tr>
<tr>
<td>CRNCYSTR</td>
<td>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</td>
</tr>
</tbody>
</table>

**Returned Value**

The `nl_langinfo()` function returns a pointer to a null-ended 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 should not be changed by the user's program.

If the item is not valid, the function returns a pointer to an empty string.

**Example**

This example retrieves the name of the codeset using the `nl_langinfo()` function.
```c
#include <langinfo.h>
#include <locale.h>
#include <nl_types.h>
#include <stdio.h>

int main(void)
{
    printf("Current codeset is %s\n", nl_langinfo(CODESET));
    return 0;
}

/**************************************************************************************************
The output should be similar to:
Current codeset is 37
****************************************************************************************/

Related Information
• “localeconv() — Retrieve Information from the Environment” on page 202
• “setlocale() — Set Locale” on page 366
• “<langinfo.h>” on page 5
• “<nl_types.h>” on page 7

perror() — Print Error Message

Format
#include <stdio.h>
void perror(const char *string);

Language Level
ANSI

Threading
Yes

Description
The perror() function prints an error message to stderr. If string is not NULL and 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 by a new-line character.

To produce accurate results, you should ensure that the perror() function is called immediately after a library function returns with an error; otherwise, subsequent calls might alter the errno value.

Return Value
There is no return value.

The value of errno can be set to:

<table>
<thead>
<tr>
<th>Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>EBADDATA</td>
<td>The message data is not valid.</td>
</tr>
</tbody>
</table>
```
EBUSY
The record or file is in use.

ENOENT
The file or library cannot be found.

EPERM
Insufficient authorization for access.

ENOREC
Record not found.

EIOERROR
A non-recoverable I/O error occurred.

EIORECERR
A recoverable I/O error occurred.

Example
This example tries to open a stream. If fopen() fails, the example prints a message and ends the program.

```c
#include <stdio.h>
#include <stdlib.h>

int main(void)
{
  FILE *fh;
  if ((fh = fopen("mylib/myfile","r")) == NULL)
  {
    perror("Could not open data file");
    abort();
  }
}
```

Related Information
- “clearerr() — Reset Error Indicators” on page 87
- “ferror() — Test for Read/Write Errors” on page 120
- “strerror() — Set Pointer to Runtime Error Message” on page 396
- “<stdio.h>” on page 13

**pow() — Compute Power**

Format
```c
#include <math.h>
double pow(double x, double y);
```

Language Level
ANSI

Threadsafe
Yes

Description
The pow() function calculates the value of x to the power of y.
Return Value

If \( y \) is 0, the \( \text{pow()} \) function returns the value 1. If \( x \) is 0 and \( y \) is negative, the \( \text{pow()} \) function sets \text{errno} to \text{EDOM} and returns 0. If both \( x \) and \( y \) are 0, or if \( x \) is negative and \( y \) is not an integer, the \( \text{pow()} \) function sets \text{errno} to \text{EDOM}, and returns 0. The \text{errno} variable can also be set to \text{ERANGE}. If an overflow results, the \( \text{pow()} \) function returns +\text{HUGE}_\text{VAL} for a large result or -\text{HUGE}_\text{VAL} for a small result.

Example

This example calculates the value of \( 2^3 \).

```c
#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 should be similar to:  *****************
2.000000 to the power of 3.000000 is 8.000000
*/
```

Related Information

- “exp() — Calculate Exponential Function” on page 114
- “log() — Calculate Natural Logarithm” on page 212
- “log10() — Calculate Base 10 Logarithm” on page 213
- “sqrt() — Calculate Square Root” on page 380
- “<math.h>” on page 6

\text{printf()} — Print Formatted Characters

Format

```c
#include <stdio.h>
int printf(const char *format-string, argument-list);
```

Language Level

ANSI

Threadsafe

Yes

Locale Sensitive

The behavior of this function might be affected by the LC\_CTYPE and LC\_NUMERIC categories of the current locale. The behavior might also be affected by the LC\_UNI\_CTYPE category of the current locale if \text{LOCALETYPE}(*\text{LOCALEUCS2}) or \text{LOCALETYPE}(*\text{LOCALEUTF}) is specified on the compilation command. For more information, see “Understanding CCSIDs and Locales” on page 565.
Description

The printf() function formats and prints a series of characters and values to the standard output stream stdout. Format specifications, beginning with a percent sign (%), determine the output format for any argument-list following the format-string. The format-string is a multibyte character string beginning and ending in its initial shift state.

The format-string is read left to right. When the first format specification is found, the value of the first argument after the format-string is converted and printed according to the format specification. The second format specification causes the second argument after the format-string to be converted and printed, 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.

A format specification has the following form:

```
%flags width.precision type
```

Conversions can be applied to the nth argument after the format-string in the argument list, rather than to the next unused argument. In this case, the conversion character % is replaced by the sequence %n$, where n is a decimal integer in the range 1 through NL_ARGMAX, giving the position of the argument in the argument list. This feature provides for the definition of format strings that select arguments in an order appropriate to specific languages.

Alternative format specification has the following form:

```
%arg-number$flags width.precision type
```

As an alternative, specific entries in the argument-list can be assigned by using the format specification outlined in the preceding diagram. This format specification and the previous format specification cannot be mixed in the same call to printf(). Otherwise, unpredictable results might occur.

The arg-number is a positive integer constant where 1 refers to the first entry in the argument-list. Arg-number cannot be greater than the number of entries in the argument-list, or else the results are undefined. Arg-number also may not be greater than NL_ARGMAX.

In format strings containing the %n$ form of conversion specifications, numbered arguments in the argument list can be referenced from the format string as many times as required.
In format strings containing the %n$ form of a conversion specification, a field width or precision may be indicated by the sequence *m$, where m is a decimal integer in the range 1 thru NL_ARGMAX giving the position in the argument list (after the format argument) of an integer argument containing the field width or precision, for example:

```c
printf("%1$d:%2$.*3$d:%4$.*3$d\n", hour, min, precision, sec);
```

The format-string can contain either numbered argument specifications (that is, %n$ and *m$), or unnumbered argument specifications (that is, % and *), but normally not both. The only exception to this is that % can be mixed with the %n$ form. The results of mixing numbered and unnumbered argument specifications in a format-string string are undefined. When numbered argument specifications are used, specifying the nth argument requires that all the leading arguments, from the first to the (n-1)th, are specified in the format string.

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).

The following optional fields control other aspects of the formatting:

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>flags</td>
<td>Justification of output and printing of signs, blanks, decimal points, octal, and hexadecimal prefixes, and the semantics for wchar_t precision unit.</td>
</tr>
<tr>
<td>width</td>
<td>Minimum number of bytes output.</td>
</tr>
<tr>
<td>precision</td>
<td>See Table 6 on page 259.</td>
</tr>
<tr>
<td>h, l, ll, L, H, D, DD</td>
<td>Size of argument expected:</td>
</tr>
<tr>
<td>h</td>
<td>A prefix with d, i, o, u, x, X, and n types that specifies that the argument is a short int or unsigned short int.</td>
</tr>
<tr>
<td>l</td>
<td>A prefix with d, i, o, u, x, X, and n types that specifies that the argument is a long int or unsigned long int.</td>
</tr>
<tr>
<td>ll</td>
<td>A prefix with d, i, o, u, x, X, and n types that specifies that the argument is a long long int or unsigned long long int.</td>
</tr>
<tr>
<td>L</td>
<td>A prefix with a, A, e, E, f, F, g, or G types that specifies that the argument is long double.</td>
</tr>
<tr>
<td>H</td>
<td>A prefix with a, A, e, E, f, F, g, or G types that specifies that the argument is _Decimal32.</td>
</tr>
<tr>
<td>D</td>
<td>A prefix with a, A, e, E, f, F, g, or G types that specifies that the argument is _Decimal64.</td>
</tr>
<tr>
<td>DD</td>
<td>A prefix with a, A, e, E, f, F, g, or G types that specifies that the argument is _Decimal128.</td>
</tr>
</tbody>
</table>

Each field of the format specification is discussed in detail below. If a percent sign (%) is followed by a character that has no meaning as a format field, the behavior is undefined. One exception to this behavior is %%. To print a percent-sign character, use %%%.

The type characters and their meanings are given in the following table:
<table>
<thead>
<tr>
<th>Character</th>
<th>Argument</th>
<th>Output Format</th>
</tr>
</thead>
</table>
| a         | Floating-point | For non decimal floating-point numbers, signed value having the form \[-\]0xh.hhhhp[sign]ddd, where h is a single hexadecimal digit, hhhh is one or more hexadecimal digits, ddd is one or more decimal digits, and sign is + or -. The number of hexadecimal digits after the decimal point is equal to the requested precision. For decimal floating-point numbers, if the precision is missing, either the f or e style formatting is used based on the following criteria. • f style formatting is used when the quantum exponent of the value is less than or equal to 0 but greater than or equal to -(n +5). The quantum exponent of a number can be determined by calling the quantexp64() function. n is the number of digits in the digit-sequence (including trailing zeros) when the decimal point is ignored. 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 the absolute value of the quantum exponent of the value. • e style formatting is used when the quantum exponent of the value does not satisfy the f style criteria. The precision is equal to \( n-1 \). The e style format of a decimal floating-point value is the same as the e style format of a non decimal floating-point value with two exceptions: a) if the value is equal to 0 then the exponent is equal to the quantum exponent of the value, and b) the exponent is always given with the minimum number of digits required (i.e., the exponent never contains a leading zero). For example:  

| 0.0000000 produces 0e-7  
| -1870 produces -1.87e+3  
<p>| If the precision modifier is present and at least as large as the precision of the value, the conversion is as if the precision modifier were missing. If the precision modifier is present and less than the precision of the value, the value is first rounded to the number of digits specified by the precision modifier. The result is then converted as if the precision modifier were missing. |
| A         | Floating-point | Identical to the a format except that uppercase alphabetic characters are used instead of lowercase alphabetic characters. |
| d, i      | Integer       | Signed decimal integer. |
| u         | Integer       | Unsigned decimal integer. |</p>
<table>
<thead>
<tr>
<th>Character</th>
<th>Argument</th>
<th>Output Format</th>
</tr>
</thead>
<tbody>
<tr>
<td>o</td>
<td>Integer</td>
<td>Unsigned octal integer.</td>
</tr>
<tr>
<td>x</td>
<td>Integer</td>
<td>Unsigned hexadecimal integer, using abcdedef.</td>
</tr>
<tr>
<td>X</td>
<td>Integer</td>
<td>Unsigned hexadecimal integer, using ABCDEF.</td>
</tr>
<tr>
<td>D(n,p)</td>
<td>Packed decimal</td>
<td>It has the format [-] dddd.dddd where the number of digits after the decimal point is equal to the precision of the specification. If the precision is missing, the default is p; if the precision is zero, and the # flag is not specified, no decimal point character appears. If the n and the p are *, an argument from the argument list supplies the value. n and p must precede the value being formatted in the argument list. At least one character appears before a decimal point. The value is rounded to the appropriate number of digits.</td>
</tr>
<tr>
<td>f</td>
<td>Floating-point</td>
<td>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.</td>
</tr>
<tr>
<td>F</td>
<td>Floating-point</td>
<td>Identical to the f format except that uppercase alphabetic characters are used instead of lowercase alphabetic characters.</td>
</tr>
<tr>
<td>e</td>
<td>Floating-point</td>
<td>Signed value having the form [-] dddd e[sign] 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 -.</td>
</tr>
<tr>
<td>E</td>
<td>Floating-point</td>
<td>Identical to the e format except that uppercase alphabetic characters are used instead of lowercase alphabetic characters.</td>
</tr>
<tr>
<td>g</td>
<td>Floating-point</td>
<td>Signed value printed 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 precision. Trailing zeros are truncated, and the decimal point appears only if one or more digits follow it.</td>
</tr>
<tr>
<td>G</td>
<td>Floating-point</td>
<td>Identical to the g format except that uppercase alphabetic characters are used instead of lowercase alphabetic characters.</td>
</tr>
<tr>
<td>c</td>
<td>Character (byte)</td>
<td>Single character.</td>
</tr>
<tr>
<td>s</td>
<td>String</td>
<td>Characters (bytes) printed up to the first null character (\0) or until precision is reached.</td>
</tr>
<tr>
<td>n</td>
<td>Pointer to integer</td>
<td>Number of characters (bytes) successfully written so far to the stream or buffer; this value is stored in the integer whose address is given as the argument.</td>
</tr>
</tbody>
</table>
### Table 5. Type characters (continued)

<table>
<thead>
<tr>
<th>Character</th>
<th>Argument</th>
<th>Output Format</th>
</tr>
</thead>
</table>
| p         | Pointer   | Pointer converted to a sequence of printable characters. It can be one of the following:  
• space pointer  
• system pointer  
• invocation pointer  
• procedure pointer  
• open pointer  
• suspend pointer  
• data pointer  
• label pointer  |
| lc or C   | Wide Character  | The (wchar_t) character is converted to a multibyte character as if by a call to wctomb(), and this character is printed out.\(^1\) |
| ls or S   | Wide Character  | The (wchar_t) characters up to the first (wchar_t) null character (L\0), or until precision is reached, are converted to multibyte characters, as if by a call to wcstombs(), and these characters are printed out. If the argument is a null string, (null) is printed.\(^1\) |

**Note:**

1. See the documentation for the wctomb() function or the documentation for the wcstombs() function for more information. You can also find additional information in “Wide Characters” on page 568.

2. If the H, D, or DD format size specifiers are not used, only 15 significant digits of output are guaranteed.

The following list shows the format of the printed values for IBM i pointers, and gives a brief description of the components of the printed values.

**Space pointer:** SPP:Context:Object:Offset:AG

- **Context:** type, subtype and name of the context
- **Object:** type, subtype and name of the object
- **Offset:** offset within the space
- **AG:** Activation group ID

**System pointer:** SYP:Context:Object:Auth:Index:AG

- **Context:** type, subtype and name of the context
- **Object:** type, subtype and name of the object
- **Auth:** authority
- **Index:** Index associated with the pointer
- **AG:** Activation group ID

**Invocation pointer:** IVP:Index:AG

- **Index:** Index associated with the pointer
- **AG:** Activation group ID

**Procedure pointer:** PRP:Index:AG

- **Index:** Index associated with the pointer
- **AG:** Activation group ID
Suspend pointer: SUP:Index:AG

Index: Index associated with the pointer
AG: Activation group ID

Data pointer: DTP:Index:AG

Index: Index associated with the pointer
AG: Activation group ID

Label pointer: LBP:Index:AG

Index: Index associated with the pointer
AG: Activation group ID

NULL pointer: NULL

The following restrictions apply to pointer printing and scanning on the IBM i operating system:

• If a pointer is printed out and scanned back from the same activation group, the scanned back pointer will be compared equal to the pointer printed out.

• If a scanf() family function scans a pointer that was printed out by a different activation group, the scanf() family function will set the pointer to NULL.

• If a pointer is printed out in a teraspace environment, just the hexadecimal value of the pointer is printed out. These results are the same as when using %#p.

The %#p format specifier has much better performance than the %p format specifier.

See the ILE C/C++ Programmer's Guide for more information about using IBM i pointers.

If a floating-point value of INFINITY or Not-a-Number (NaN) is formatted using the a, e, f, or g format, the output string is infinity or nan. If a floating-point value of INFINITY or Not-A-Number (NaN) is formatted using the A, E, F, or G format, the output string is INFINITY or NAN.

The flag characters and their meanings are as follows (notice that more than one flag can appear in a format specification):

<table>
<thead>
<tr>
<th>Flag</th>
<th>Meaning</th>
<th>Default</th>
</tr>
</thead>
<tbody>
<tr>
<td>-</td>
<td>Left-justify the result within the field width.</td>
<td>Right-justify.</td>
</tr>
<tr>
<td>+</td>
<td>Prefix the output value with a sign (+ or -) if the output value is of a signed type.</td>
<td>Sign appears only for negative signed values (-).</td>
</tr>
<tr>
<td>blank(‘ ’)</td>
<td>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.</td>
<td>No blank.</td>
</tr>
<tr>
<td>Flag</td>
<td>Meaning</td>
<td>Default</td>
</tr>
<tr>
<td>------</td>
<td>---------</td>
<td>---------</td>
</tr>
<tr>
<td>#</td>
<td>When used with the o, x, or X formats, the # flag prefixes any nonzero output value with 0, 0x, or 0X, respectively.</td>
<td>No prefix.</td>
</tr>
<tr>
<td></td>
<td>When used with the D(n,p), a, A, e, E, f or F formats, the # flag forces the output value to contain a decimal point in all cases.</td>
<td>Decimal point appears only if digits follow it.</td>
</tr>
<tr>
<td></td>
<td>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.</td>
<td>Decimal point appears only if digits follow it; trailing zeros are truncated.</td>
</tr>
<tr>
<td></td>
<td>When used with the lS or S format, the # flag causes precision to be measured in characters, regardless of the size of the character. For example, if single-byte characters are being printed, a precision of 4 would result in 4 bytes being printed. If double-byte characters are being printed, a precision of 4 would result in 8 bytes being printed.</td>
<td>Precision indicates the maximum number of bytes to be output.</td>
</tr>
<tr>
<td></td>
<td>When used with the p format, the # flag converts the pointer to hex digits. These hex digits cannot be converted back into a pointer, unless in a teraspace environment.</td>
<td>Pointer converted to a sequence of printable characters.</td>
</tr>
<tr>
<td>0</td>
<td>When used with the d, i, D(n,p) o, u, x, X, a, A, e, E, f, F, g, or G formats, the 0 flag causes leading 0s 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.</td>
<td>Space padding. No space padding for D(n,p).</td>
</tr>
</tbody>
</table>

The # flag should not be used with c, lc, d, i, u, or s types.

**Width** is a nonnegative decimal integer controlling the minimum number of characters printed. If the number of characters (bytes) 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.

**Width** never causes a value to be truncated; if the number of characters (bytes) in the output value is greater than the specified width, or width is not given, all characters of the value are printed (subject to the precision specification).

For the ls or S type, width is specified in bytes. If the number of bytes in the output value is less than the specified width, single-byte blanks are added on the left or the right (depending on whether the - flag is specified) until the minimum width is reached.

The width specification can be an asterisk (*), in which case an argument from the argument list supplies the value. The width argument must precede the value being formatted in the argument list.

**Precision** is a nonnegative decimal integer preceded by a period, which specifies the number of characters to be printed 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 or packed decimal value.

The precision specification can be an asterisk (*), in which case an argument from the argument list supplies the value. The precision argument must precede the value being formatted in the argument list.

The interpretation of the precision value and the default when the precision is omitted depend on the type, as shown in the following table:
Table 6. Values of Precision

<table>
<thead>
<tr>
<th>Type</th>
<th>Meaning</th>
<th>Default</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>For non decimal floating-point numbers, precision specifies the number of hexadecimal digits to be printed after the decimal point. For decimal floating-point numbers, precision specifies the number of significant digits to be printed.</td>
<td>For non decimal floating-point numbers, the default precision is sufficient for an exact representation of the value. If precision is 0, no decimal point is printed. For decimal floating-point numbers, the default precision is sufficient for an exact representation of the value. Refer to Table 5 on page 254 for more details on the format used.</td>
</tr>
<tr>
<td>i</td>
<td>Precision specifies the minimum number of digits to be printed. 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.</td>
<td>If precision is 0 or omitted entirely, or if the period (.) appears without a number following it, the precision is set to 1.</td>
</tr>
<tr>
<td>f</td>
<td>Precision specifies the number of digits to be printed after the decimal point. The last digit printed is rounded.</td>
<td>Default precision for f, F, e and E is six. Default precision for D(n,p) is p. If precision is 0 or the period appears without a number following it, no decimal point is printed.</td>
</tr>
<tr>
<td>g</td>
<td>Precision specifies the maximum number of significant digits printed.</td>
<td>All significant digits are printed. Default precision is six.</td>
</tr>
<tr>
<td>c</td>
<td>No effect.</td>
<td>The character is printed.</td>
</tr>
<tr>
<td>lc</td>
<td>No effect.</td>
<td>The wchar_t character is converted and resulting multibyte character is printed.</td>
</tr>
<tr>
<td>s</td>
<td>Precision specifies the maximum number of characters (bytes) to be printed. Characters (bytes) in excess of precision are not printed.</td>
<td>Characters are printed until a null character is encountered.</td>
</tr>
<tr>
<td>ls</td>
<td>Precision specifies the maximum number of bytes to be printed. Bytes in excess of precision are not printed; however, multibyte integrity is always preserved.</td>
<td>wchar_t characters are converted and resulting multibyte characters are printed.</td>
</tr>
</tbody>
</table>

**Return Value**

The printf() function returns the number of bytes printed. The value of errno may be set to:

<table>
<thead>
<tr>
<th>Value</th>
<th>Meaning</th>
</tr>
</thead>
</table>

EBADMODE  
The file mode that is specified is not valid.

ECONVERT  
A conversion error occurred.

EIOERROR  
A non-recoverable I/O error occurred.

EIORECERR  
A recoverable I/O error occurred.

EILSEQ  
An invalid multibyte character sequence was encountered.

EPUTANGET  
An illegal write operation occurred after a read operation.

ESTDOUT  
stdout cannot be opened.

**Note:** The radix character for the printf() function is locale sensitive. The radix character is the decimal point to be used for the # flag character of the format string parameter for the format types D(n,p), a, A, e, E, f, F, g, and G.

**Examples**

This example prints data in a variety of formats.

```c
#include <stdio.h>
#include <stdlib.h>

int main(void)
{
    char ch = 'h', *string = "computer";
    int count = 234, hex = 0x10, oct = 010, dec = 10;
    double fp = 251.7366;
    wchar_t wc = (wchar_t)0x0058;
    wchar_t ws[4];
    printf("1234567890123\n\n", &count);
    printf("Value of count should be 13; count = %d\n", count);
    printf("%10c%5c\n", ch, ch);
    printf("%25s\n%25.4s\n\n", string, string);
    printf("%f    %.2f    %e    %E\n
", fp, fp, fp, fp);
    printf("%i    %i     %i\n
", hex, oct, dec);
}
/* *******************  Output should be similar to:  *******************
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
******************************************************************/
```

Example that uses printf()

```c
#include <stdio.h>
#include <stdlib.h>
#include <locale.h>

/* This program is compiled with LOCALETYPE(*LOCALEUCS2) and */
/* SYSIFCOPT(*IFSIO) */
/* We will assume the locale setting is the same as the CCSID of the */
/* job. We will also assume any files involved have a CCSID of */
```

260 IBM i: ILE C/C++ Runtime Library Functions
int main(void)
{
    wchar_t wc = 0x0058;     /* UNICODE X */
    wchar_t ws[4];
    setlocale(LC_ALL,
            "/QSYS.LIB/EN_US.LOCALE"); /* a CCSID 37 locale */
    ws[0] = 0x0041;        /* UNICODE A   */
    ws[1] = (wchar_t)0x0042;        /* UNICODE B   */
    ws[2] = (wchar_t)0x0043;        /* UNICODE C   */
    ws[3] = (wchar_t)0x0000;
    /* The output displayed is CCSID 37 */
    printf("%lc   %ls

",wc,ws);
    printf("%lc   %.2ls

",wc,ws);

    setlocale(LC_ALL,
            "/QSYS.LIB/JA_JP.LOCALE"); /* a CCSID 5026 locale */
    /* big A means an A that takes up 2 bytes on the screen */
    /* It will look bigger then single byte A */
    ws[0] = (wchar_t)0xFF21;        /* UNICODE big A   */
    ws[1] = (wchar_t)0xFF22;        /* UNICODE big B   */
    ws[2] = (wchar_t)0xFF23;        /* UNICODE big C   */
    ws[3] = (wchar_t)0x0000;
    wc = 0xff11;                    /* UNICODE big 1   */
    printf("%lc   %ls

",wc,ws);

    printf("%lc %ls\n\n",wc,ws);
    printf("%lc %.4ls\n\n",wc,ws);
    printf("%lc %#.2ls\n\n",wc,ws);
}

Example that uses printf()
ws[3] = (wchar_t)0x0000;
/* The output displayed is CCSID 37 */
printf("%lc  %ls
\n",wc,ws);
/* Now let's try a mixed-byte CCSID example */
/* You would need a device that can handle mixed bytes to */
/* display this correctly. */
setlocale(LC_ALL,
"/QSYS.LIB/JA_JP.LOCALE"); /* a CCSID 5026 locale */
/* big A means an A that takes up 2 bytes on the screen */
/* it will look bigger than single byte A */
ws[0] = (wchar_t)0x42C1;        /* big A */
ws[1] = (wchar_t)0x42C2;        /* big B */
ws[2] = (wchar_t)0x42C3;        /* big C */
ws[3] = (wchar_t)0x0000;        /* big 1 */
wc = 0x42F1;                    /* big 1 */
printf("%lc  %ls
\n",wc,ws);
/* The output of this printf is not shown below and it */
/* will differ depending on the device you display it on, */
/* but if you looked at the string in hex it would look */
/* like this: 0E42F10F40404040E42C142C242C30F */
/* 0E is shift out, 0F is shift in, and 42F1 is the */
/* big 1 in CCSID 5026 */
printf("%.4ls\n",wc,ws);
/* The output of this printf is not shown below either. */
/* The hex would look like: */
/* 0E42F10F40404040E42C10F */
/* Since the precision is in bytes we only get 4 bytes */
/* of the string. */
printf("%.2ls\n",wc,ws);
/* The output of this printf is not shown below either. */
/* The hex would look like: */
/* 0E42F10F40404040E42C120F */
/* The # means precision is in characters regardless */
/* of size. So we get 2 characters of the string. */
}
/
/******************** Output should be similar to: *******************/

D  ABC

/*********************/

Related Information

• “fprintf() — Write Formatted Data to a Stream” on page 141
• “fscanf() — Read Formatted Data” on page 155
• “quantexpd32() - quantexpd64() - quantexpd128() — Compute the Quantum Exponent” on page 270
• “scanf() — Read Data” on page 358
• “sprintf() — Print Formatted Data to Buffer” on page 379
• “sscanf() — Read Data” on page 382
• “vfprintf() — Print Argument Data to Stream” on page 457
• “vprintf() — Print Argument Data” on page 464
• “vsscanf() — Print Argument Data to Buffer” on page 468
• “wprintf() — Format Data as Wide Characters and Print” on page 538
• “<stdio.h>” on page 13
putc() – putchar() — Write a Character

Format

```c
#include <stdio.h>
int putc(int c, FILE *stream);
int putchar(int c);
```

Language Level

ANSI

Threading Safe

No

```c
#undef putc
#undef putchar
```

allows the `putc` or `putchar` function to be called instead of the macro version of these functions. The functions are threadsafe.

Description

The `putc()` function converts `c` to unsigned char and then writes `c` to the output `stream` at the current position. The `putchar()` is equivalent to `putc(c, stdout)`.

The `putc()` function can be defined as a macro so the argument can be evaluated multiple times.

The `putc()` and `putchar()` functions are not supported for files opened with type=record.

Return Value

The `putc()` and `putchar()` functions return the character written. A return value of EOF indicates an error.

The value of `errno` may be set to:

<table>
<thead>
<tr>
<th>Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>ECONVERT</td>
<td>A conversion error occurred.</td>
</tr>
<tr>
<td>EPUTANDGET</td>
<td>An illegal write operation occurred after a read operation.</td>
</tr>
<tr>
<td>EIOERROR</td>
<td>A non-recoverable I/O error occurred.</td>
</tr>
<tr>
<td>EIORECERR</td>
<td>A recoverable I/O error occurred.</td>
</tr>
</tbody>
</table>

Example

This example writes the contents of a buffer to a data stream. In this example, the body of the for statement is null because the example carries out the writing operation in the test expression.
```c
#include <stdio.h>
#include <string.h>
#define  LENGTH 80

int main(void)
{
    FILE *stream = stdout;
    int i, ch;
    char buffer[LENGTH + 1] = "Hello world";
    /* This could be replaced by using the fwrite function */
    for ( i = 0;
         (i < strlen(buffer)) && ((ch = putc(buffer[i], stream)) != EOF);
            ++i);
}
/********************  Expected output:  **************************
Hello world
*/
```

**Related Information**

- “fputc() — Write Character” on page 142
- “fwrite() — Write Items” on page 168
- “getc() – getchar() — Read a Character” on page 174
- “puts() — Write a String” on page 265
- “putwc() — Write Wide Character” on page 266
- “putwchar() — Write Wide Character to stdout ” on page 268
- “<stdio.h>” on page 13

---

**putenv() — Change/Add Environment Variables**

**Format**

```c
#include <stdlib.h>
int putenv(const char *varname);
```

**Language Level**

XPG4

**Threadsafe**

Yes

**Job CCSID Interface**

All character data sent to this function is expected to be in the CCSID of the job. All character data returned by this function is in the CCSID of the job. See “Understanding CCSIDs and Locales” on page 565 for more information.

**Description**

The `putenv()` function sets the value of an environment variable by altering an existing variable or creating a new one. The `varname` parameter points to a string of the form `var=x`, where x is the new value for the environment variable `var`. 
The name cannot contain a blank or an equal (=) symbol. For example,

```
PATH NAME=/my_lib/joe_user
```

is not valid because of the blank between PATH and NAME. Similarly,

```
PATH=NAME=/my_lib/joe_user
```

is not valid because of the equal symbol between PATH and NAME. The system interprets all characters following the first equal symbol as being the value of the environment variable.

**Return Value**

The `putenv()` function returns 0 if successful. If `putenv()` fails then -1 is returned and `errno` is set to indicate the error.

**Example**

```
#include <stdio.h>
#include <stdlib.h>

int main(void)
{
    char *pathvar;
    if (-1 == putenv("PATH=/:/home/userid")) {
        printf("putenv failed \n");
        return EXIT_FAILURE;
    }
    /* getting and printing the current environment path */
    pathvar = getenv("PATH");
    printf("The current path is: %s\n", pathvar);
    return 0;
}
```

The output should be:

```
The current path is: /:/home/userid
```

**Related Information**

- “`getenv()` — Search for Environment Variables” on page 176
- “`<stdlib.h>`” on page 15

**puts() — Write a String**

**Format**

```
#include <stdio.h>
int puts(const char *string);
```

**Language Level**

ANSI

**Threadsafe**

Yes
Description
The puts() function writes the given string to the standard output stream stdout; it also appends a new-line character to the output. The ending null character is not written.

Return Value
The puts() function returns EOF if an error occurs. A nonnegative return value indicates that no error has occurred.

The value of errno may be set to:

<table>
<thead>
<tr>
<th>Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>ECONVERT</td>
<td>A conversion error occurred.</td>
</tr>
<tr>
<td>EPUTANDGET</td>
<td>An illegal write operation occurred after a read operation.</td>
</tr>
<tr>
<td>EIOERROR</td>
<td>A non-recoverable I/O error occurred.</td>
</tr>
<tr>
<td>EIORECERR</td>
<td>A recoverable I/O error occurred.</td>
</tr>
</tbody>
</table>

Example
This example writes Hello World to stdout.

```c
#include <stdio.h>

int main(void)
{
    if ( puts("Hello World") == EOF )
        printf( "Error in puts
" );
}

/************************  Expected output:  ******************
Hello World
*/
```

Related Information
- “fputs() — Write String” on page 145
- “fputws() — Write Wide-Character String” on page 148
- “fgets() — Read a Line” on page 178
- “putc() — putchar() — Write a Character” on page 263
- “putwc() — Write Wide Character” on page 266
- “<stdio.h>” on page 13

putwc() — Write Wide Character

Format

```c
#include <stdio.h>
#include <wchar.h>
wint_t putwc(wint_t wc, FILE *stream);
```
Language Level
ANSI

Threadsafe
Yes

Locale Sensitive
The behavior of this function might be affected by the LC_CTYPE category of the current locale. This behavior might also be affected by the LC_UNI_CTYPE category of the current locale if LOCALETYPE(*LOCALEUCS2) or LOCALETYPE(*LOCALEUTF) is specified on the compilation command. This function is not available when LOCALETYPE(*CLD) is specified on the compilation command. For more information, see “Understanding CCSIDs and Locales” on page 565.

Integrated File System Interface
This function is not available when SYSIFCOPT(*NOIFSIO) is specified on the compilation command.

Wide Character Function
See “Wide Characters” on page 568 for more information.

Description
The putwc() function writes the wide character wc to the stream at the current position. It also advances the file position indicator for the stream appropriately. The putwc() function is equivalent to the fputwc() function except that some platforms implement putwc() as a macro. Therefore, for portability, the stream argument to putwc() should not be an expression with side effects.

Using a non-wide-character function with the putwc() function on the same stream results in undefined behavior. After calling the putwc() function, flush the buffer or reposition the stream pointer before calling a write function for the stream, unless EOF has been reached. After a write operation on the stream, flush the buffer or reposition the stream pointer before calling the putwc() function.

Return Value
The putwc() function returns the wide character written. If a write error occurs, it sets the error indicator for the stream and returns WEOF. If an encoding error occurs when a wide character is converted to a multibyte character, the putwc() function sets errno to EILSEQ and returns WEOF.

For information about errno values for putwc(), see “fputc() — Write Character” on page 142.

Example
The following example uses the putwc() function to convert the wide characters in wcs to multibyte characters and write them to the file putwc.out.
```c
#include <stdio.h>
#include <stdlib.h>
#include <wchar.h>
#include <errno.h>

int main(void)
{
    FILE    *stream;
    wchar_t wcs = L"A character string.";
    int     i;

    if (NULL == (stream = fopen("putwc.out", "w"))) {
        printf("Unable to open: \"putwc.out\".\n")
        exit(1);
    }

    for (i = 0; wcs[i] != L'\0'; i++) {
        errno = 0;
        if (WEOF == putwc(wcs[i], stream)) {
            printf("Unable to putwc() the wide character.\n"
"wcs[%d] = 0x%lx\n", i, wcs[i]);
            if (EILSEQ == errno)
                printf("An invalid wide character was encountered.\n"n);
            exit(1);
        }
    }

    fclose(stream);
    return 0;
}
```

Related Information

- “fputc() — Write Character” on page 142
- “fputwc() — Write Wide Character” on page 146
- “fputws() — Write Wide-Character String” on page 148
- “getwc() — Read Wide Character from Stream” on page 179
- “putc() – putchar() — Write a Character” on page 263
- “putwchar() — Write Wide Character to stdout” on page 268
- “<stdio.h>” on page 13
- “<wchar.h>” on page 16

**putwchar() — Write Wide Character to stdout**

**Format**

```c
#include <wchar.h>
wint_t putwchar(wint_t wc);
```

**Language Level**

ANSI

**Threadsafe**

Yes
Locale Sensitive

The behavior of this function might be affected by the LC_CTYPE category of the current locale. This behavior might also be affected by the LC_UNI_CTYPE category of the current locale if LOCALETYPE(*LOCALEUCS2) or LOCALETYPE(*LOCALEUTF) is specified on the compilation command. This function is not available when LOCALETYPE(*CLD) is specified on the compilation command. For more information, see “Understanding CCSIDs and Locales” on page 565.

Integrated File System Interface

This function is not available when SYSIFCOPT(*NOIFSIO) is specified on the compilation command.

Wide Character Function

See “Wide Characters” on page 568 for more information.

Description

The putwchar() function converts the wide character wc to a multibyte character and writes it to stdout. A call to the putwchar() function is equivalent to putwc(wc, stdout).

Using a non-wide-character function with the putwchar() function on the same stream results in undefined behavior. After calling the putwchar() function, flush the buffer or reposition the stream pointer before calling a write function for the stream, unless EOF has been reached. After a write operation on the stream, flush the buffer or reposition the stream pointer before calling the putwchar() function.

Return Value

The putwchar() function returns the wide character written. If a write error occurs, the putwchar() function sets the error indicator for the stream and returns WEOF. If an encoding error occurs when a wide character is converted to a multibyte character, the putwchar() function sets errno to EILSEQ and returns WEOF.

For information about errno values for putwc(), see “fputc() — Write Character” on page 142.

Example

This example uses the putwchar() function to write the string in wcs.

```c
#include <stdio.h>
#include <wchar.h>
#include <errno.h>
#include <stdlib.h>

int main(void)
{
    wchar_t *wcs = L"A character string."
    int i;
    for (i = 0; wcs[i] != L'\0'; i++) {
        errno = 0;
        if (WEOF == putwchar(wcs[i])) {
            printf("Unable to putwchar() the wide character.\n"");
            printf("wcs[%d] = 0x%lx\n", i, wcs[i]);
            if (EILSEQ == errno)
                printf("An invalid wide character was encountered.\n");
            exit(EXIT_FAILURE);
        }
    }
    return 0;
}
```

The output should be similar to:

```
A character string.
```

Library Functions 269
quantexpd32() - quantexpd64() - quantexpd128() — Compute the Quantum Exponent

Format
```c
#define __STDC_WANT_DEC_FP__
#include <math.h>
int quantexpd32(_Decimal32 x);
int quantexpd64(_Decimal64 x);
int quantexpd128(_Decimal128 x);
```

Language Level
ANSI

Threadsafe
Yes

Description
The quantexpd32(), quantexpd64(), and quantexpd128() functions compute the quantum exponent of a finite argument. The numerical value of a finite number is given by: \((-1)^{\text{sign}} \times \text{coefficient} \times 10^{\text{exponent}}\). The quantum of a finite number is given by \(1 \times 10^{\text{exponent}}\) and represents the value of a unit in the least significant position of the coefficient of a finite number. The quantum exponent is the exponent of the quantum (represented by \(\text{exponent}\) above).

Return Value
The quantexpd32(), quantexpd64(), and quantexpd128() functions return the quantum exponent of \(x\). If \(x\) is infinite or NaN, errno is set to EDOM and the value INT_MIN is returned.

Example
This example illustrates the use of the quantexpd128() function:
```c
#define __STDC_WANT_DEC_FP__
#include <stdio.h>
#include <math.h>
int main(void)
{
    _Decimal128 x;
    int y;
    x = 4.56DL;
    y = quantexpd128(x);
    printf("quantexpd128(%DDa) = %d\n", x, y);
    return 0;
}
/******************** Output should be similar to: *******************/
quantexpd128(4.56) = -2
/**/```

**Related Information**

- “quantized32() - quantized64() - quantized128() — Set the Quantum Exponent of X to the Quantum Exponent of Y” on page 271
- “samequantumd32() - samequantumd64() - samequantumd128() — Determine if Quantum Exponents X and Y are the Same” on page 356

---

**quantized32() - quantized64() - quantized128() — Set the Quantum Exponent of X to the Quantum Exponent of Y**

### Format

```c
#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);
```

### Language Level

ANSI

### Threading

Yes

### Description

The `quantized32()`, `quantized64()`, and `quantized128()` functions set the quantum exponent of argument `x` to the quantum exponent of argument `y`, while trying to keep the value the same. If the quantum exponent is being increased, the value is correctly rounded according to the rounding mode Round to Nearest, Ties to Even. If the result does not have the same value as `x`, the “inexact” floating-point exception is raised. If the quantum exponent is being decreased, and the significand of the result has more digits than the type would allow, the result is NaN and the “invalid” floating-point exception is raised.

If one or 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 infinity and the sign is the same as `x`.

The `quantized32()`, `quantized64()`, and `quantized128()` functions do not signal underflow or overflow.
Return Value
The quantized32(), quantized64(), and quantized128() functions return the number which is equal in value (except for any rounding) and sign to x, and which has a quantum exponent equal to the quantum exponent of y.

Example
This example illustrates the use of the quantized128() function:

```c
#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 = %DDa\n"
           "tax = %DDa (price * rate = %DDa)\n"
           "total = %DDa\n", price, tax, price * rate, total );
  return 0;
}
/** Output should be similar to: **************
 price = 64999.99
 tax = 6418.75 (price * rate = 6418.7490125)
 total = 71418.74
 */
```

Related Information
• “quantexpd32() - quantexpd64() - quantexpd128() — Compute the Quantum Exponent” on page 270
• “samequantumd32() - samequantumd64() - samequantumd128() — Determine if Quantum Exponents X and Y are the Same” on page 356

qsort() — Sort Array

Format
```
#include <stdlib.h>
void qsort(void *base, size_t num, size_t width,
int(*compare)(const void *key, const void *element));
```

Language Level
ANSI

Threadsafe
Yes

Description
The `qsort()` function sorts an array of `num` elements, each of `width` bytes in size. The `base` pointer is a pointer to the array to be sorted. The `qsort()` function overwrites this array with the sorted elements.

The `compare` argument is a pointer to a function you must supply that takes a pointer to the `key` argument and to an array `element`, in that order. The `qsort()` function calls this function one or more times during the search. The function must compare the `key` and the `element` and return one of the following values:
Table 7. Return values of the qsort() compare function

<table>
<thead>
<tr>
<th>Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>Less than 0</td>
<td>key less than element</td>
</tr>
<tr>
<td>0</td>
<td>key equal to element</td>
</tr>
<tr>
<td>Greater than 0</td>
<td>key greater than element</td>
</tr>
</tbody>
</table>

The sorted array elements are stored in ascending order, as defined by your compare function. You can sort in reverse order by reversing the sense of “greater than” and “less than” in compare. The order of the elements is unspecified when two elements compare equally.

Return Value

There is no return value.

Example

This example sorts the arguments (argv) in ascending lexical sequence, using the comparison function compare() supplied in the example.

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
/* Declaration of compare() as a function */
int compare(const void *, const void *);

int main (int argc, char *argv[])
{
    int i;
    argv++;  // Copy argv[1] into argv[0]
    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));
}
```

/* If the program is passed the arguments: **************
************* 'Does' 'this' 'really' 'sort' 'the' 'arguments' 'correctly?' ****
************** then the expected output is: *******************/

arguments
correctly?
really
sort
the
this
Does
*/

Related Information

- “bsearch() — Search Arrays” on page 76
- “<stdlib.h>” on page 15
QXXCHGDA() — Change Data Area

Format

```c
#include <xxdtaa.h>

void QXXCHGDA(_DTAA_NAME_T dtaname, short int offset, short int len,
              char *dtaptr);
```

Language Level

ILE C Extension

Threadsafe

Yes

Job CCSID Interface

All character data sent to this function is expected to be in the CCSID of the job. All character data returned by this function is in the CCSID of the job. See “Understanding CCSIDs and Locales” on page 565 for more information.

Description

The QXXCHGDA() function allows you to change the data area specified by dtaname, starting at position offset, with the data in the user buffer pointed to by dtaptr of length len. The structure dtaname contains the names of the data area and the library that contains the data area. The values that can be specified for the data area name are:

*LDA
  Specifies that the contents of the local data area are to be changed. The library name dtaa_lib must be blank.

*GDA
  Specifies that the contents of the group data area are to be changed. The library name dtaa_lib must be blank.

data-area-name
  Specifies that the contents of the data area created using the Create Data Area (CRTDTAARA) CL command are to be changed. The library name dtaa_lib must be either *LIBL, *CURLIB, or the name of the library where the data area (data-area-name) is located. The data area is locked while it is being changed.

QXXCHGDA can only be used to change character data.
Example

```c
#include <stdio.h>
#include <xxdtaa.h>
#define START   1
#define LENGTH  8

int main(void)
{
    char newdata[LENGTH] = "new data";
    /* The local data area will be changed         */
    _DTAA_NAME_T dtaname = {"LDA      ", "          "};
    /* Use function to change the local data area. */
    QXXCHGDA(dtaname, START, LENGTH, newdata);
    /* The first 8 characters in the local data area*/
    /* are: new data                                 */
}
```

Related Information

- “QXXRTVDA() — Retrieve Data Area” on page 280

**QXXDTOP() — Convert Double to Packed Decimal**

Format

```c
#include <xxcvt.h>
void QXXDTOP(unsigned char *pptr, int digits, int fraction,
             double value);
```

Language Level

ILE C Extension

Threadsafe

Yes

Description

The QXXDTOP function converts the double value specified in value to a packed decimal number with digits total digits, and fraction fractional digits. The result is stored in the array pointed to by pptr.

Example

```c
#include <xxcvt.h>
#include <stdio.h>

int main(void)
{
    unsigned char pptr[10];
    int digits = 8, fraction = 6;
    double value = 3.141593;
    QXXDTOP(pptr, digits, fraction, value);
}
```

Related Information

- “QXXDTOZ() — Convert Double to Zoned Decimal” on page 276
- “QXXITOP() — Convert Integer to Packed Decimal” on page 277
QXXDTOZ() — Convert Double to Zoned Decimal

Format

```c
#include <xxcvt.h>
void QXXDTOZ(unsigned char *zptr, int digits, int fraction, double value);
```

Language Level

ILE C Extension

Threading

Yes

Description

The QXXDTOZ function converts the `double` value specified in `value` to a zoned decimal number with `digits` total digits, and `fraction` fractional digits. The result is stored in the array pointed to by `zptr`.

Example

```c
#include <xxcvt.h>
#include <stdio.h>

int main(void)
{
    unsigned char zptr[10];
    int digits = 8, fraction = 6;
    double value = 3.141593;
    QXXDTOZ(zptr, digits, fraction, value);
    /* Zoned value is : 03141593 */
}
```

Related Information

- “QXXDTOP() — Convert Double to Packed Decimal” on page 275
- “QXXITOP() — Convert Integer to Packed Decimal” on page 277
- “QXXITOZ() — Convert Integer to Zoned Decimal” on page 277
- “QXXPTOD() — Convert Packed Decimal to Double” on page 278
- “QXXPTOI() — Convert Packed Decimal to Integer” on page 279
- “QXXZTOD() — Convert Zoned Decimal to Double” on page 281
- “QXXZTOI() — Convert Zoned Decimal to Integer” on page 282
QXXITOP() — Convert Integer to Packed Decimal

**Format**

```c
#include <xxcvt.h>
void QXXITOP(unsigned char *pptr, int digits, int fraction, int value);
```

**Language Level**

ILE C Extension

**Threadsafe**

Yes

**Description**

The QXXITOP function converts the integer specified in `value` to a packed decimal number with `digits` total digits, and `fraction` fractional digits. The result is stored in the array pointed to by `pptr`.

**Example**

```c
#include <xxcvt.h>
#include <stdio.h>

int main(void)
{
    unsigned char pptr[10];
    int digits = 3, fraction = 0;
    int value = 116;
    QXXITOP(pptr, digits, fraction, value);
}
```

**Related Information**

- “QXXDTOP() — Convert Double to Packed Decimal” on page 275
- “QXXDTOZ() — Convert Double to Zoned Decimal” on page 276
- “QXXITOZ() — Convert Integer to Zoned Decimal” on page 277
- “QXXPTOD() — Convert Packed Decimal to Double” on page 278
- “QXXPTOI() — Convert Packed Decimal to Integer” on page 279
- “QXXZTOD() — Convert Zoned Decimal to Double” on page 281
- “QXXZTOI() — Convert Zoned Decimal to Integer” on page 282

QXXITOZ() — Convert Integer to Zoned Decimal

**Format**

```c
#include <xxcvt.h>
void QXXITOZ(unsigned char *zptr, int digits, int fraction, int value);
```

**Language Level**

ILE C Extension
**Threadsafe**
Yes

**Description**
The QXXITOZ function converts the integer specified in `value` to a zoned decimal number with `digits` total digits, and `fraction` fractional digits. The result is stored in the array pointed to by `zptr`.

**Example**

```c
#include <xxcvt.h>
#include <stdio.h>

int main(void)
{
    unsigned char zptr[10];
    int digits = 9, fraction = 0;
    int value = 111115;
    QXXITOZ(zptr, digits, fraction, value);
    /* Zoned value is : 000111115 */
}
```

**Related Information**
- “QXXDTOP() — Convert Double to Packed Decimal” on page 275
- “QXXDTOZ() — Convert Double to Zoned Decimal” on page 276
- “QXXITOP() — Convert Integer to Packed Decimal” on page 277
- “QXXPTOD() — Convert Packed Decimal to Double” on page 278
- “QXXPTOI() — Convert Packed Decimal to Integer” on page 279
- “QXXZTOD() — Convert Zoned Decimal to Double” on page 281
- “QXXZTOI() — Convert Zoned Decimal to Integer” on page 282

---

**QXXPTOD() — Convert Packed Decimal to Double**

**Format**

```c
#include <xxcvt.h>
double QXXPTOD(unsigned char *pptr, int digits, int fraction);
```

**Language Level**
ILE C Extension

**Threadsafe**
Yes

**Description**
The QXXPTOD function converts a packed decimal number to a `double`.
Example

```c
#include <xxcvt.h>
#include <stdio.h>

int main(void)
{
    unsigned char pptr[10];
    int digits = 8, fraction = 6;
    double value = 6.123456, result;
    /* First convert an integer to a packed decimal,*/
    QXXDTOP(pptr, digits, fraction, value);
    /* then convert it back to a double.            */
    result = QXXPTOD(pptr, digits, fraction);
    /* result = 6.123456                            */
}
```

Related Information

- “QXXDTOP() — Convert Double to Packed Decimal” on page 275
- “QXXDTOZ() — Convert Double to Zoned Decimal” on page 276
- “QXXITOP() — Convert Integer to Packed Decimal” on page 277
- “QXXITOZ() — Convert Integer to Zoned Decimal” on page 277
- “QXXPTOI() — Convert Packed Decimal to Integer” on page 279
- “QXXZTOD() — Convert Zoned Decimal to Double” on page 281
- “QXXZTOI() — Convert Zoned Decimal to Integer” on page 282

QXXPTOI() — Convert Packed Decimal to Integer

Format

```c
#include <xxcvt.h>
int QXXPTOI(unsigned char *pptr, int digits, int fraction);
```

Language Level

ILE C Extension

Threadsafe

Yes

Description

The QXXPTOI function converts a packed decimal number to an integer.

Example

```c
#include <xxcvt.h>
#include <stdio.h>

int main(void)
{
    unsigned char pptr[10];
    int digits = 3, fraction = 0, value = 104, result;
    /* First convert an integer to a packed decimal,*/
    QXXITOP(pptr, digits, fraction, value);
    /* then convert it back to an integer.            */
    result = QXXPTOI(pptr, digits, fraction);
    /* result = 104                                 */
}
```
Related Information

- “QXXDTOP() — Convert Double to Packed Decimal” on page 275
- “QXXDTOZ() — Convert Double to Zoned Decimal” on page 276
- “QXXITOP() — Convert Integer to Packed Decimal” on page 277
- “QXXITOZ() — Convert Integer to Zoned Decimal” on page 277
- “QXXPTOD() — Convert Packed Decimal to Double” on page 278
- “QXXZTOD() — Convert Zoned Decimal to Double” on page 281
- “QXXZTOI() — Convert Zoned Decimal to Integer” on page 282

**QXXRTVDA() — Retrieve Data Area**

**Format**

```c
#include <xxdtaa.h>

void QXXRTVDA(_DTAA_NAME_T dtaname, short int offset,
              short int len, char *dtaptr);
```

**Language Level**

ILE C Extension

**Threadsafe**

Yes

**Job CCSID Interface**

All character data sent to this function is expected to be in the CCSID of the job. All character data returned by this function is in the CCSID of the job. See “Understanding CCSIDs and Locales” on page 565 for more information.

**Description**

The following typedef definition is included in the <xxdtaa.h> header file. The character arrays are not null-ended strings so they must be blank filled.

```c
typedef struct _DTAA_NAME_T {
    char dtaa_name[10]; /* name of data area */
    char dtaa_lib[10]; /* library that contains data area */
}_DTAA_NAME_T;
```

The QXXRTVDA() function retrieves a copy of the data area specified by `dtaname` starting at position `offset` with a length of `len`. The structure `dtaname` contains the names of the data area and the library that contains the data area. The values that can be specified for the data area name are:

**LDA**

The contents of the local data area are to be retrieved. The library name `dtaa_lib` must be blank.

**GDA**

The contents of the group data area are to be retrieved. The library name `dtaa_lib` must be blank.

**PDA**

Specifies that the contents of the program initialization parameters (PIP) data area are to be retrieved. The PIP data area is created for each pre-started job and is a character area up to 2000 characters in length. You cannot retrieve the PIP data area until you have acquired the requester. The library name `dtaa_lib` must be blank.
data-area-name

Specifies that the contents of the data area created using the Create Data Area (CRTDTAARA) CL command are to be retrieved. The library name dtaa_lib must be either *LIBL, *CURLIB, or the name of the library where the data area (data-area-name) is located. The data area is locked while the data is being retrieved.

The parameter dtaptr is a pointer to the storage that receives the retrieved copy of the data area. Only character data can be retrieved using QXXRTVDA.

Example

```c
#include <stdio.h>
#include <xxdtaa.h>
#define DATA_AREA_LENGTH 30
#define START 6
#define LENGTH 7

int main(void)
{
    char uda_area[DATA_AREA_LENGTH];
    /* Retrieve data from user-defined data area currently in MYLIB */
    _DTAA_NAME_T dtaname = {"USRDDA", "MYLIB"};
    /* Use the function to retrieve some data into uda_area. */
    QXXRTVDA(dtname, START, LENGTH, uda_area);
    /* Print the contents of the retrieved subset. */
    printf("uda_area contains %7.7s\n", uda_area);
}
```

Related Information

- “QXXCHGDA() — Change Data Area” on page 274

QXXZTOD() — Convert Zoned Decimal to Double

Format

```c
#include <xxcvt.h>
double QXXZTOD(unsigned char *zptr, int digits, int fraction);
```

Language Level

ILE C Extension

Threadsafe

Yes

Description

The QXXZTOD function converts to a double, the zoned decimal number (with digits total digits, and fraction fractional digits) pointed to by zptr. The resulting double value is returned.
Example

```c
#include <xxcvt.h>
#include <stdio.h>

int main(void)
{
    unsigned char zptr[] = "06123456";
    int digits = 8, fraction = 6;
    double result;
    result = QXXZTOD(zptr, digits, fraction);
    /* result = 6.123456 */
}
```

Related Information

- “QXXDTOP() — Convert Double to Packed Decimal” on page 275
- “QXXDTOZ() — Convert Double to Zoned Decimal” on page 276
- “QXXITOP() — Convert Integer to Packed Decimal” on page 277
- “QXXITOZ() — Convert Integer to Zoned Decimal” on page 277
- “QXXPTOD() — Convert Packed Decimal to Double” on page 278
- “QXXPTOI() — Convert Packed Decimal to Integer” on page 279
- “QXXZTOI() — Convert Zoned Decimal to Integer” on page 282

### QXXZTOI() — Convert Zoned Decimal to Integer

**Format**

```c
#include <xxcvt.h>
int QXXZTOI(unsigned char *zptr, int digits, int fraction);
```

**Language Level**

ILE C Extension

**Threadsafe**

Yes

**Description**

The QXXZTOI function converts to an `integer`, the zoned decimal number (with `digits` total digits, and `fraction` fractional digits) pointed to by `zptr`. The resulting integer is returned.

**Example**

```c
#include <xxcvt.h>
#include <stdio.h>

int main(void)
{
    unsigned char zptr[] = "000111115";
    int digits = 9, fraction = 0, result;
    result = QXXZTOI(zptr, digits, fraction);
    /* result = 111115 */
}
```
Related Information

- “QXXDTOP() — Convert Double to Packed Decimal” on page 275
- “QXXDTOZ() — Convert Double to Zoned Decimal” on page 276
- “QXXITOP() — Convert Integer to Packed Decimal” on page 277
- “QXXITOZ() — Convert Integer to Zoned Decimal” on page 277
- “QXXPTOD() — Convert Packed Decimal to Double” on page 278
- “QXXPTOI() — Convert Packed Decimal to Integer” on page 279
- “QXXZTOD() — Convert Zoned Decimal to Double” on page 281

raise() — Send Signal

Format

```c
#include <signal.h>
int raise(int sig);
```

Language Level

ANSI

Threadsafe

Yes

Description

The `raise()` function sends the signal `sig` to the running program. If compiled with `SYSIFCOPT(*ASYNCSIGNAL)` on the compilation command, this function uses asynchronous signals. The asynchronous version of this function throws a signal to the process or thread.

Return Value

The `raise()` function returns 0 if successful, nonzero if unsuccessful.

Example

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.

```c
#include <signal.h>
#include <stdio.h>

void sig_hand(int); /* declaration of sig_hand() as a function */

int main(void)
{
    signal(SIGUSR1, sig_hand); /* set up handler for SIGUSR1 */
    raise(SIGUSR1); /* signal SIGUSR1 is raised */
    /* sig_hand() is called */
}

void sig_hand(int sig)
{
    static int count = 0; /* initialized only once */
    count++;
    if (count == 10) /* ignore the first 9 occurrences of this signal */
        exit(10);
```
else
    signal(SIGUSR1, sig_hand); /* set up the handler again */
}
/* This is a program fragment and not a complete program */

Related Information

- “signal() — Handle Interrupt Signals” on page 374
- “Signal Handling Action Definitions” on page 548
- “<signal.h>” on page 11
- Signal APIs in the APIs topic in the Information Center.
- POSIX thread APIs in the APIs topic in the Information Center.

rand() – rand_r() — Generate Random Number

Format

```c
#include <stdlib.h>
int rand(void);
int rand_r(unsigned int *seed);
```

Language Level

ANSI

Threadsafe

No

rand() is not threadsafe, but rand_r() is.

Description

The `rand()` function generates a pseudo-random integer in the range 0 to RAND_MAX (macro defined in `<stdlib.h>`). Use the `srand()` function before calling `rand()` to set a starting point for the random number generator. If you do not call the `srand()` function first, the default seed is 1.

**Note:** The `rand_r()` function is the restartable version of `rand()`.

Return Value

The `rand()` function returns a pseudo-random number.

Example

This example prints the first 10 random numbers generated.
```c
#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 should be similar to:
/******************
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=4886
*/

Related Information
• “srand() — Set Seed for rand() Function” on page 381
• “<stdlib.h>” on page 15

_Racquire() — Acquire a Program Device

Format
#include <recio.h>
int _Racquire(_RFILE *fp, char *dev);

Language Level
ILE C Extension

Threading
No

Job CCSID Interface
All character data sent to this function is expected to be in the CCSID of the job. All character data
returned by this function is in the CCSID of the job. See “Understanding CCSIDs and Locales” on page 565
for more information.

Description
The _Racquire() function acquires the program device specified by the dev parameter and associates it
with the file specified by fp. The dev parameter is a null-ended C string. The program device name must
be specified in uppercase. The program device must be defined to the file.

This function is valid for display and ICF files.

Return Value
The _Racquire() function returns 1 if it is successful or zero if it is unsuccessful. The value of errno may
be set to EIOERROR (a non-recoverable I/O error occurred) or EIORECERR (a recoverable I/O error
occurred).
See Table 22 on page 543 and Table 24 on page 547 for errno settings.

Example

```c
#include <stdio.h>
#include <recio.h>
#include <string.h>
#include <stdlib.h>

int main(void)
{
    _RFILE    *fp;
    _RIOFB_T  *rfb;

    /* Open the device file.                                      */
    if (( fp = _Ropen ( "MYLIB/T1677RD2", "ar+" )) == NULL )
    {
        printf ( "Could not open file\n" );
        exit ( 1 );
    }

    _Racquire ( fp,"DEVICE1" );    /* Acquire another program device. */
    /* Replace with actual device name. */
    _Rformat ( fp,"FORMAT1" );     /* Set the record format for the   */
    /* display file.                   */
    rfb = _Rwrite ( fp, "", 0 );    /* Set up the display.             */
    /* Do some processing...          */

    _Rclose ( fp);
}
```

Related Information

- “_Rrelease() — Release a Program Device” on page 341

_Rclose() — Close a File

Format

```c
#include <recio.h>

int _Rclose(_RFILE *fp);
```

Language Level

ILE C Extension

Threadsafe

Yes

Description

The _Rclose() function closes the file specified by fp. Before this file is closed, all buffers associated with it are flushed and all buffers reserved for it are released. The file is closed even if an exception occurs. The _Rclose() function applies to all types of files.

Note: The storage pointed to by the _RFILE pointer is freed by the _Rclose() function. After the use of the _Rclose() function, any attempt to use the _RFILE pointer is not valid.
Return Value
The _Rclose() function returns zero if the file is closed successfully, or EOF if the close operation failed or the file was already closed. The file is closed even if an exception occurs, and zero is returned.

The value of errno may be set to:

<table>
<thead>
<tr>
<th>Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>ENOTOPEN</td>
<td>The file is not open.</td>
</tr>
<tr>
<td>EIOERROR</td>
<td>A non-recoverable I/O error occurred.</td>
</tr>
<tr>
<td>EIORECERR</td>
<td>A recoverable I/O error occurred.</td>
</tr>
</tbody>
</table>

See Table 22 on page 543 and Table 24 on page 547 for errno settings.

Example

```c
#include <stdio.h>
#include <stdlib.h>
#include <recio.h>

int main(void)
{
    _RFILE      *fp;
    /* Open the file for processing in arrival sequence. */
    if (( fp = _Ropen ( "MYLIB/T1677RD1", "rr+, arrseq=Y" ) ) == NULL )
    {
        printf ( "Open failed\n" );
        exit ( 1 );
    } else
    /* Do some processing */;
    _Rclose ( fp );
}
```

Related Information
• “_Ropen() — Open a Record File for I/O Operations” on page 315

_Rcommit() — Commit Current Record

Format

```c
#include <recio.h>
int _Rcommit(char *cmtid);
```

Language Level
ILE C Extension

Theradsafe
No

Job CCSID Interface
All character data sent to this function is expected to be in the CCSID of the job. All character data returned by this function is in the CCSID of the job. See “Understanding CCSIDs and Locales” on page 565 for more information.
Description

The _Rcommit() function completes the current transaction for the job that calls it and establishes a new commitment boundary. All changes made since the last commitment boundary are made permanent. Any file or resource that is open under commitment control in the job is affected.

The cntid parameter is a null-ended C string used to identify the group of changes associated with a commitment boundary. It cannot be longer than 4000 bytes.

The _Rcommit() function applies to database and DDM files.

Return Value

The _Rcommit() function returns 1 if the operation is successful or zero if the operation is unsuccessful. The value of errno may be set to EIOERROR (a non-recoverable I/O error occurred) or EIORECERR (a recoverable I/O error occurred).

See Table 22 on page 543 and Table 24 on page 547 for errno settings.

Example

```c
#include <stdio.h>
#include <recio.h>
#include <stdlib.h>
#include <string.h>

int main(void)
{
    char       buf[40];
    int        rc = 1;
    _RFILE     *purf;
    _RFILE     *dailyf;

    /* Open purchase display file and daily transaction file */
    if ( ( purf = _Ropen ( "MYLIB/T1677RD3", "ar+,indicators=y" )) == NULL )
    {
        printf ( "Display file did not open.\n" );
        exit ( 1 );
    }
    if ( ( dailyf = _Ropen ( "MYLIB/T1677RDA", "wr,commit=y" ) ) == NULL )
    {
        printf ( "Daily transaction file did not open.\n" );
        exit ( 2 );
    }

    /* Select purchase record format */
    _Rformat ( purf, "PURCHASE" );
    /* Invite user to enter a purchase transaction. */
    /* The _Rwrite function writes the purchase display. */
    _Rwrite ( purf, "", 0 );
    _Rreadn ( purf, buf, sizeof(buf), _DFT );

    /* Update daily transaction file */
    rc = ((( _Rwrite ( dailyf, buf, sizeof(buf) )))->num_bytes );
    /* If the databases were updated, then commit the transaction. */
    /* Otherwise, rollback the transaction and indicate to the */
    /* user that an error has occurred and end the application. */
    if ( rc )
    {
        _Rcommit ( "Transaction complete" );
    }
    else
    {
        _Rrollbck ( );
        _Rformat ( purf, "ERROR" );
    }

    _Rclose ( purf );
    _Rclose ( dailyf );
}
```
Related Information

- "_Rrollback() — Roll Back Commitment Control Changes" on page 343

_Rdelete() — Delete a Record

Format

```c
#include <recio.h>
_RIOFB_T *Rdelete(_RFILE *fp);
```

Language Level

ILE C Extension

Threadsafe

Yes

Description

The _Rdelete() function deletes the record that is currently locked for update in the file specified by fp. After the delete operation, the record is not locked. The file must be open for update.

A record is locked for update by reading or locating to it unless __NO_LOCK is specified on the read or locate option. If the __NO_POSITION option is specified on the locate operation that locked the record, the record deleted may not be the record that the file is currently positioned to.

This function is valid for database and DDM files.

Return Value

The _Rdelete() function returns a pointer to the _RIOFB_T structure associated with fp. If the operation is successful, the num_bytes field contains 1. If the operation is unsuccessful, the num_bytes field contains zero.

The value of errno may be set to:

<table>
<thead>
<tr>
<th>Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>ENOTDLT</td>
<td>The file is not open for delete operations.</td>
</tr>
<tr>
<td>EIOERROR</td>
<td>A non-recoverable I/O error occurred.</td>
</tr>
<tr>
<td>EIORECERR</td>
<td>A recoverable I/O error occurred.</td>
</tr>
</tbody>
</table>

See Table 22 on page 543 and Table 24 on page 547 for errno settings.
Example
#include <stdio.h>
#include <stdlib.h>
#include <recio.h>

int main(void)
{
  _RFILE      *fp;
  _XXOPFB_T   *opfb;

  /* Open the file for processing in arrival sequence. */
  if (( fp = _Ropen ( "MYLIB/T1677RD1", "rr+, arrseq=Y" )) == NULL )
  {
    printf ( "Open failed\n" );
    exit ( 1 );
  }

  /* Get the library and file names of the file opened. */
  opfb = _Ropnfbk ( fp );
  printf ( "Library: %10.10s
File:    %10.10s
", opfb->library_name,
              opfb->file_name);

  /* Get the first record. */
  _Rreadf ( fp, NULL, 20, __DFT );
  printf ( "First record:  %10.10s
", *(fp->in_buf) );

  /* Delete the first record. */
  _Rdelete ( fp );

  _Rclose ( fp );
}

Related Information

• “_Rrlslck() — Release a Record Lock” on page 342

_Rdevatr() — Get Device Attributes

Format
#include <recio.h>
#include <xxfdbk.h>
_XXDEV_ATR_T *_Rdevatr(_RFILE *fp, char *dev);

Language Level
ILE C Extension

Threadsafe
No

Job CCSID Interface
All character data sent to this function is expected to be in the CCSID of the job. All character data
returned by this function is in the CCSID of the job. See “Understanding CCSIDs and Locales” on page 565
for more information.

Description
The _Rdevatr() function returns a pointer to a copy of the device attributes feedback area for the file
pointed to by fp, and the device specified by dev.

The dev parameter is a null-ended C string. The device name must be specified in uppercase.
The `_Rdevatr()` function is valid for display and ICF files.

**Return Value**

The `_Rdevatr()` function returns NULL if an error occurs.

See Table 22 on page 543 and Table 24 on page 547 for errno settings.

**Example**

```c
#include <stdio.h>
#include <recio.h>
#include <string.h>
#include <stdlib.h>

int main(int argc, char ** argv)
{
    _RFILE   *fp; /* File pointer */
    _RIOFB_T *rfb; /* Pointer to the file's feedback structure */
    _XXIOFB_T *iofb; /* Pointer to the file's feedback area */
    _XXDEV_ATR_T *dv_atr; /* Pointer to a copy of the file's device attributes feedback area */

    /* Open the device file. */
    if (( fp = _Ropen ( "MYLIB/T1677RD2", "ar+" )) == NULL )
    { 
        printf ( "Could not open file\n" );
        exit ( 1 );
    }

    dv_atr = _Rdevatr (fp, argv[1]);
    if (dv_atr == NULL)
        printf("Error occurred getting device attributes for %s.\n", argv[1]);

    _Rclose ( fp );
}
```

**Related Information**

- “`_Racquire()` — Acquire a Program Device” on page 285
- “`_Rrelease()` — Release a Program Device” on page 341

### realloc() — Change Reserved Storage Block Size

**Format**

```c
#include <stdlib.h>
void *realloc(void *ptr, size_t size);
```

**Language Level**

ANSI

**Threading safe**

Yes

**Description**

The `realloc()` function 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 necessarily give all bits of each element an initial value of 0.

If `size` is 0 and the `ptr` is not NULL, `realloc()` frees the storage allocated to `ptr` and returns NULL.

**Note:**

1. All heap storage is associated with the activation group of the calling function. As such, storage should be allocated, deallocated, and reallocated within the same activation group. You cannot allocate heap storage within one activation group and deallocate or reallocate that storage from a different activation group. For more information about activation groups, see the *ILE Concepts* manual.

2. If the Quick Pool memory manager has been enabled in the current activation group then storage is retrieved using Quick Pool memory manager. See “`_C_Quickpool_Init()` — Initialize Quick Pool Memory Manager” on page 93 for more information.

**Return Value**

The `realloc()` function returns a pointer to the reallocated storage block. The storage location of the block may be moved by the `realloc()` function. Thus, the `ptr` argument to the `realloc()` function is not necessarily the same as the return value.

If `size` is 0, the `realloc()` function returns NULL. If there is not enough storage to expand the block to the given size, the original block is unchanged and the `realloc()` function returns NULL.

The storage to which the return value points is aligned for storage of any type of object.

To use teraspace storage instead of single-level store storage without changing the C source code, specify the `TERASPACE(*YES *TSIFC)` parameter on the compiler command. This maps the `realloc()` library function to `_C_TS_realloc()`, its teraspace storage counterpart. The maximum amount of teraspace storage that can be allocated by each call to `_C_TS_realloc()` is 2GB - 240, or 214743408 bytes. For additional information about teraspace storage, see the *ILE Concepts* manual.

**Example**

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.
```c
#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 */
        perror("Out of storage");
        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 */
        perror("Out of storage");
        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] );
}

/**** If the initial value entered is 2 and the second value entered
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[ 3 ] = 2003 */
```

Related Information

- “calloc() — Reserve and Initialize Storage” on page 80
- “_C_Quickpool_Debug() — Modify Quick Pool Memory Manager Characteristics” on page 91
- “_C_Quickpool_Init() — Initialize Quick Pool Memory Manager” on page 93
- “_C_Quickpool_Report() — Generate Quick Pool Memory Manager Report” on page 95
- “Heap Memory” on page 578
- “free() — Release Storage Blocks” on page 151
- “malloc() — Reserve Storage Block” on page 217
- “<stdlib.h>” on page 15
regcomp() — Compile Regular Expression

Format

```c
#include <regex.h>
int regcomp(regex_t *preg, const char *pattern, int cflags);
```

Language Level

XPG4

Threadsafe

Yes

Locale Sensitive

The behavior of this function might be affected by the LC_CTYPE and LC_COLLATE categories of the current locale. This function is not available when LOCALETYPE(*CLD) is specified on the compilation command. For more information, see “Understanding CCSIDs and Locales” on page 565.

Description

The `regcomp()` function compiles the source regular expression pointed to by `pattern` into an executable version and stores it in the location pointed to by `preg`. You can then use the `regexec()` function to compare the regular expression to other strings.

The `cflags` flag defines the attributes of the compilation process:

<table>
<thead>
<tr>
<th>cflag</th>
<th>Description String</th>
</tr>
</thead>
<tbody>
<tr>
<td>REG_ALT_NL</td>
<td>• When LOCALETYPE(*LOCALE) is specified, the newline character of the integrated file system will be matched by regular expressions.</td>
</tr>
<tr>
<td></td>
<td>• When LOCALETYPE(*LOCALEUTF) is specified, the database newline character will be matched.</td>
</tr>
<tr>
<td></td>
<td>If the REG_ALT_NL flag is not set, the default for LOCALETYPE(*LOCALE) is to match the database newline, and the default for LOCALETYPE(*LOCALEUTF) is to match the integrated file system newline.</td>
</tr>
<tr>
<td></td>
<td><strong>Note:</strong> For UTF-8 and UTF-32, the newline character of the integrated file system and the database newline character are the same.</td>
</tr>
<tr>
<td>REG_EXTENDED</td>
<td>Support extended regular expressions.</td>
</tr>
<tr>
<td>REG_NEWLINE</td>
<td>Treat newline character as a special end-of-line character; it then establishes the line boundaries matched by the ] and $ patterns, and can only be matched within a string explicitly using \n. (If you omit this flag, the newline character is treated like any other character.)</td>
</tr>
<tr>
<td>REG_ICASE</td>
<td>Ignore case in match.</td>
</tr>
</tbody>
</table>
Regular expressions are a context-independent syntax that can represent a wide variety of character sets and character set orderings, which can be interpreted differently depending on the current locale. The functions `regcomp()`, `regerror()`, `regexec()`, and `regfree()` use regular expressions in a similar way to the UNIX awk, ed, grep, and egrep commands.

Return Value

If the `regcomp()` function is successful, it returns 0. Otherwise, it returns an error code that you can use in a call to the `regerror()` function, and the content of `preg` is undefined.

Example

```c
#include <regex.h>
#include <stdio.h>
#include <stdlib.h>

int main(void)
{
    regex_t    preg;
    char       *string = "a very simple simple simple string";
    char       *pattern = "\(sim[a-z]le\) \1";
    int        rc;
    size_t     nmatch = 2;
    regmatch_t pmatch[2];

    if (0 != (rc = regcomp(&preg, pattern, 0))) {
        printf("regcomp() failed, returning nonzero (%d)\n", rc);
        exit(EXIT_FAILURE);
    }

    if (0 != (rc = regexec(&preg, string, nmatch, pmatch, 0))) {
        printf("Failed to match '%s' with '%s', returning %d\n", string, pattern, rc);
    }
    else {
        printf("With the whole expression, "
            "a matched substring "%.*s\n" is found at position %d to %d\n",
            pmatch[0].rm_eo - pmatch[0].rm_so, &string[pmatch[0].rm_so],
            pmatch[0].rm_so, pmatch[0].rm_eo - 1);
        printf("With the sub-expression, "
            "a matched substring "%.*s\n" is found at position %d to %d\n",
            pmatch[1].rm_eo - pmatch[1].rm_so, &string[pmatch[1].rm_so],
            pmatch[1].rm_so, pmatch[1].rm_eo - 1);
    }
    regfree(&preg);
    return 0;
}
```

The output should be similar to:

```
With the whole expression, a matched substring "simple simple" is found at position 7 to 19.
With the sub-expression, a matched substring "simple" is found at position 7 to 12.
```

Library Functions 295
Related Information

- “regerror() — Return Error Message for Regular Expression” on page 296
- “regexec() — Execute Compiled Regular Expression” on page 297
- “regfree() — Free Memory for Regular Expression” on page 300
- “<regex.h>” on page 10

regerror() — Return Error Message for Regular Expression

Format

```
#include <regex.h>
size_t regerror(int errcode, const regex_t *preg,
                char *errbuf, size_t errbuf_size);
```

Language Level

XPG4

Threadsafe

Yes

Locale Sensitive

The behavior of this function might be affected by the LC_CTYPE and LC_COLLATE categories of the current locale. This function is not available when LOCALETYPE(*CLD) is specified on the compilation command. For more information, see “Understanding CCSIDs and Locales” on page 565.

Description

The `regerror()` function finds the description for the error code `errcode` for the regular expression `preg`. The description for `errcode` is assigned to `errbuf`. The `errbuf_size` value specifies the maximum message size that can be stored (the size of `errbuf`). The description strings for `errcode` are:

<table>
<thead>
<tr>
<th><code>errcode</code></th>
<th>Description String</th>
</tr>
</thead>
<tbody>
<tr>
<td>REG_NOMATCH</td>
<td><code>regexec()</code> failed to find a match.</td>
</tr>
<tr>
<td>REG_BADPAT</td>
<td>Invalid regular expression.</td>
</tr>
<tr>
<td>REG_ECOLLATE</td>
<td>Invalid collating element referenced.</td>
</tr>
<tr>
<td>REG_ECTYPE</td>
<td>Invalid character class type referenced.</td>
</tr>
<tr>
<td>REG_EESCAPE</td>
<td>Last character in regular expression is a .</td>
</tr>
<tr>
<td>REG_ESUBREG</td>
<td>Number in \digit invalid or error.</td>
</tr>
<tr>
<td>REG_EBRACK</td>
<td>[ ] imbalance.</td>
</tr>
<tr>
<td>REG_EPAREN</td>
<td>( ) or () imbalance.</td>
</tr>
<tr>
<td>REG_EBRACE</td>
<td>{ } imbalance.</td>
</tr>
<tr>
<td>REG_BADBDR</td>
<td>Expression between { and } is invalid.</td>
</tr>
<tr>
<td>REG_ERANGE</td>
<td>Invalid endpoint in range expression.</td>
</tr>
<tr>
<td>REG_ESPACE</td>
<td>Out of memory.</td>
</tr>
<tr>
<td>REG_BADRPT</td>
<td>?, *, or + not preceded by valid regular expression.</td>
</tr>
<tr>
<td>errcode</td>
<td>Description String</td>
</tr>
<tr>
<td>------------</td>
<td>----------------------------------------</td>
</tr>
<tr>
<td>REG_ECHAR</td>
<td>Invalid multibyte character.</td>
</tr>
<tr>
<td>REG_EBOL</td>
<td>^ anchor not at beginning of regular expression.</td>
</tr>
<tr>
<td>REG_EEOL</td>
<td>$ anchor not at end of regular expression.</td>
</tr>
<tr>
<td>REG_ECOMP</td>
<td>Unknown error occurred during regcomp() call.</td>
</tr>
<tr>
<td>REG_EEXEC</td>
<td>Unknown error occurred during regexec() call.</td>
</tr>
</tbody>
</table>

Return Value
The `regerror()` function returns the size of the buffer needed to hold the string that describes the error condition. The value of `errno` may be set to **ECONVERT** (conversion error).

Example
This example compiles an invalid regular expression, and prints an error message using the `regerror()` function.

```c
#include <regex.h>
#include <stdio.h>
#include <stdlib.h>

int main(void) {
    regex_t preg;
    char    *pattern = "a[missing.bracket";
    int     rc;
    char    buffer[100];
    if (0 != (rc = regcomp(&preg, pattern, REG_EXTENDED))) {
        regerror(rc, &preg, buffer, 100);
        printf("regcomp() failed with '%s'
", buffer);
        exit(EXIT_FAILURE);
    }
    return 0;
}
```

The output should be similar to:

```
regcomp() failed with '[ ] imbalance.'
```

Related Information
- “`regcomp()` — Compile Regular Expression” on page 294
- “`regexec()` — Execute Compiled Regular Expression” on page 297
- “`regfree()` — Free Memory for Regular Expression” on page 300
- “<regex.h>” on page 10

**`regexec()` — Execute Compiled Regular Expression**

**Format**

```c
#include <regex.h>
int regexec(const regex_t *preg, const char *string,
            size_t nmatch, regmatch_t *pmatch, int eflags);
```
Locale Sensitive

The behavior of this function might be affected by the LC_CTYPE and LC_COLLATE categories of the current locale. This function is not available when LOCALETYPE(*CLD) is specified on the compilation command. For more information, see “Understanding CCSIDs and Locales” on page 565.

Description

The `regexec()` function compares the null-ended `string` against the compiled regular expression `preg` to find a match between the two.

The `nmatch` value is the number of substrings in `string` that the `regexec()` function should try to match with subexpressions in `preg`. The array you supply for `pmatch` must have at least `nmatch` elements.

The `regexec()` function fills in the elements of the array `pmatch` with offsets of the substrings in `string` that correspond to the parenthesized subexpressions of the original pattern given to the `regcomp()` function to create `preg`. The zeroth element of the array corresponds to the entire pattern. If there are more than `nmatch` subexpressions, only the first `nmatch - 1` are stored. If `nmatch` is 0, or if the `REG_NOSUB` flag was set when `preg` was created with the `regcomp()` function, the `regexec()` function ignores the `pmatch` argument.

The `eflags` flag defines customizable behavior of the `regexec()` function:

<table>
<thead>
<tr>
<th><code>erfalg</code></th>
<th>Description String</th>
</tr>
</thead>
<tbody>
<tr>
<td>REG_NOTBOL</td>
<td>Indicates that the first character of <code>string</code> is not the beginning of line.</td>
</tr>
<tr>
<td>REG_NOTEOL</td>
<td>Indicates that the first character of <code>string</code> is not the end of line.</td>
</tr>
</tbody>
</table>

When a basic or extended regular expression is matched, any given parenthesized subexpression of the original pattern could participate in the match of several different substrings of `string`. The following rules determine which substrings are reported in `pmatch`:

1. If subexpression `i` in a regular expression is not contained within another subexpression, and it participated in the match several times, then the byte offsets in `pmatch[i]` will delimit the last such match.

2. If subexpression `i` is not contained within another subexpression, and it did not participate in an otherwise successful match, the byte offsets in `pmatch[i]` will be -1. A subexpression does not participate in the match when any of following conditions are true:
   - `*` or `{ }` appears immediately after the subexpression in a basic regular expression.
   - `*`, `?`, or `{ }` appears immediately after the subexpression in an extended regular expression, and the subexpression did not match (matched 0 times).
   - `|` is used in an extended regular expression to select this subexpression or another, and the other subexpression matched.

3. If subexpression `i` is contained within another subexpression `j`, and `i` is not contained within any other subexpression that is contained within `j`, and a match of subexpression `j` is reported in `pmatch[j]`, then the match or non-match of subexpression `i` reported in `pmatch[i]` will be as described in 1. and 2. above, but within the substring reported in `pmatch[j]` rather than the whole string.

4. If subexpression `i` is contained in subexpression `j`, and the byte offsets in `pmatch[j]` are -1, then the offsets in `pmatch[i]` also will be -1.
5. If subexpression \(i\) matched a zero-length string, then both byte offsets in \(pmatch[i]\) will be the byte offset of the character or null terminator immediately following the zero-length string.

If the REG_NOSUB flag was set when \(preg\) was created by the \(regcomp()\) function, the contents of \(pmatch\) are unspecified. If the REG_NEWLINE flag was set when \(preg\) was created, new-line characters are allowed in string.

Return Value

If a match is found, the \(regexec()\) function returns 0. If no match is found, the \(regexec()\) function returns REG_NOMATCH. Otherwise, it returns a nonzero value indicating an error. A nonzero return value can be used in a call to the \(regerror()\) function.

Example

```c
#include <regex.h>
#include <stdio.h>
#include <stdlib.h>
int main(void)
{
    regex_t    preg;
    char       *string = "a very simple simple simple string";
    char       *pattern = "\(sim[a-z]le\) \1";
    int        rc;
    size_t     nmatch = 2;
    regmatch_t pmatch[2];
    if (0 != (rc = regcomp(&preg, pattern, 0))) {
        printf("regcomp() failed, returning nonzero (%d)\n", rc);
        exit(EXIT_FAILURE);
    }
    if (0 != (rc = regexec(&preg, string, nmatch, pmatch, 0))) {
        printf("Failed to match '%s' with '%s',returning %d.\n",
                string, pattern, rc);
    }
    else {
        printf("With the whole expression, "
              "a matched substring "%.*s" is found at position %d to %d.\n",
                pmatch[0].rm_eo - pmatch[0].rm_so, &string[pmatch[0].rm_so],
                pmatch[0].rm_so, pmatch[0].rm_eo - 1);
        printf("With the sub-expression, "
              "a matched substring "%.*s" is found at position %d to %d.\n",
                pmatch[1].rm_eo - pmatch[1].rm_so, &string[pmatch[1].rm_so],
                pmatch[1].rm_so, pmatch[1].rm_eo - 1);
    }
    regfree(&preg);
    return 0;
}
```

The output should be similar to:

With the whole expression, a matched substring "simple simple" is found at position 7 to 19.
With the sub-expression, a matched substring "simple" is found at position 7 to 12.

Related Information

- “\(regcomp()\) — Compile Regular Expression” on page 294
- “\(regerror()\) — Return Error Message for Regular Expression” on page 296
- “\(regfree()\) — Free Memory for Regular Expression” on page 300
- “\(<regex.h>\)” on page 10
regfree() — Free Memory for Regular Expression

Format

```c
#include <regex.h>
void regfree(regex_t *preg);
```

Language Level

XPG4

Threading

Yes

Locale Sensitive

The behavior of this function might be affected by the LC_CTYPE and LC_COLLATE categories of the current locale. This function is not available when LOCALETYPE(*CLD) is specified on the compilation command. For more information, see “Understanding CCSIDs and Locales” on page 565.

Description

The `regfree()` function frees any memory that was allocated by the `regcomp()` function to implement the regular expression `preg`. After the call to the `regfree()` function, the expression that is defined by `preg` is no longer a compiled regular or extended expression.

Return Value

There is no return value.

Example

This example compiles an extended regular expression.

```c
#include <regex.h>
#include <stdio.h>
#include <stdlib.h>

int main(void)
{
    regex_t preg;
    char    *pattern = ".*(simple).*";
    int     rc;

    if (0 != (rc = regcomp(&preg, pattern, REG_EXTENDED))) {
        printf("regcomp() failed, returning nonzero (%d)\n", rc);
        exit(EXIT_FAILURE);
    }
    regfree(&preg);
    printf("regcomp() is successful.\n");
    return 0;
}
```

Related Information

- “regcomp() — Compile Regular Expression” on page 294
remove() — Delete File

Format

```c
#include <stdio.h>
int remove(const char *filename);
```

Language Level

ANSI

Threadsafe

Yes

Description

The `remove()` function deletes the file specified by `filename`. If the filename contains the member name, the member is removed or the file is deleted.

Note: You cannot remove a nonexistent file or a file that is open.

Return Value

The `remove()` function returns 0 if it successfully deletes the file. A nonzero return value indicates an error.

The value of `errno` may be set to `ECONVERT` (conversion error).

Example

When you call this example with a file name, the program attempts to remove that file. It issues a message if an error occurs.

```c
#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 )
            perror( "Could not remove file" );
}
```

Related Information

- “fopen() — Open Files” on page 134
- “rename() — Rename File” on page 302
- “<stdio.h>” on page 13
rename() — Rename File

Format

```c
#include <stdio.h>
int rename(const char *oldname, const char *newname);
```

Language Level

ANSI

Threadsafe

Yes

Description

The `rename()` function renames the file specified by `oldname` to the name given by `newname`. The `oldname` pointer must specify the name of an existing file. The `newname` pointer must not specify the name of an existing file. You cannot rename a file with the name of an existing file. You also cannot rename an open file.

The file formats that can be used to satisfy the new name depend on the format of the old name. The following table shows the valid file formats that can be used to specify the old file name and the corresponding valid file formats for the new name.

If the format for both new name and old name is `lib/file(member)`, then the file cannot change. If the file name changes, rename will not work. For example, the following is not valid: `lib/file1(member1) lib/file2(member1).

<table>
<thead>
<tr>
<th>Old Name</th>
<th>New Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>lib/file(member)</td>
<td>lib/file(member), lib/file, file, file(member)</td>
</tr>
<tr>
<td>lib/file</td>
<td>lib/file, file</td>
</tr>
<tr>
<td>file</td>
<td>lib/file, file</td>
</tr>
<tr>
<td>file(member)</td>
<td>lib/file(member), lib/file, file, file(member)</td>
</tr>
</tbody>
</table>

Return Value

The `rename()` function returns 0 if successful. On an error, it returns a nonzero value.

The value of `errno` may be set to `ECONVERT` (conversion error).

Example

This example takes two file names as input and uses `rename()` to change the file name from the first name to the second name.

```c
#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 )
        perror ( "Could not rename file" );
}
```
Related Information

- “fopen() — Open Files” on page 134
- “remove() — Delete File” on page 301
- “<stdio.h>” on page 13

rewind() — Adjust Current File Position

Format

```c
#include <stdio.h>
void rewind(FILE *stream);
```

Language Level

ANSI

Threadsafe

Yes

Description

The `rewind()` function repositions the file pointer associated with `stream` to the beginning of the file. A call to the `rewind()` function is the same as:

```c
(void)fseek(stream, 0L, SEEK_SET);
```

except that the `rewind()` function also clears the error indicator for the `stream`.

The `rewind()` function is not supported for files opened with type=record.

Return Value

There is no return value.

The value of `errno` may be set to:

<table>
<thead>
<tr>
<th>Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>EBADF</td>
<td>The file pointer or descriptor is not valid.</td>
</tr>
<tr>
<td>ENODEV</td>
<td>Operation attempted on a wrong device.</td>
</tr>
<tr>
<td>EIO</td>
<td>A non-recoverable I/O error occurred.</td>
</tr>
<tr>
<td>EIORECERR</td>
<td>A recoverable I/O error occurred.</td>
</tr>
</tbody>
</table>

Example

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 in the data.
```c
#include <stdio.h>

FILE *stream;
int data1, data2, data3, data4;
int main(void)
{
    data1 = 1; data2 = -37;
    /* Place data in the file */
    stream = fopen("mylib/myfile", "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 should be similar to:  **************
The values read back in are: 1 and -37
*/
```

**Related Information**

- "fgetpos() — Get File Position" on page 124
- "fseek() – fseeko() — Reposition File Position" on page 157
- "fsetpos() — Set File Position" on page 159
- "ftell() – ftello() — Get Current Position" on page 161
- "<stdio.h>" on page 13

---

**_Rfeod() — Force the End-of-Data**

**Format**

```c
#include <recio.h>
int _Rfeod(_RFILE *fp);
```

**Language Level**

ILE C Extension

**Threading**

Yes

**Description**

The `_Rfeod()` function forces an end-of-data condition for a device or member associated with the file specified by `fp`. Any outstanding updates, deletes or writes that the system is buffering will be forced to nonvolatile storage. If a database file is open for input, any outstanding locks will be released.

The `_Rfeod()` function positions the file to *END unless the file is open for multi-member processing and the current member is not the last member in the file. If multi-member processing is in effect and the current member is not the last member in the file, `_Rfeod()` will open the next member of the file and position it to *START.

The `_Rfeod()` function is valid for all types of files.
Return Value

The `_Rfeod()` function returns 1 if multi-member processing is taking place and the next member has been opened. EOF is returned if the file is positioned to *END. If the operation is unsuccessful, zero is returned. The value of errno may be set to EIOERROR (a non-recoverable error occurred) or EIORECERR (a recoverable I/O error occurred). See Table 22 on page 543 and Table 24 on page 547 for errno settings.

Example

```c
#include <stdio.h>
#include <stdlib.h>
#include <recio.h>

int main(void)
{
    _RFILE   *in;
    char     new_purchase[21] = "PEAR      1002022244";
    /* Open the file for processing in keyed sequence. */
    if ( (in = _Ropen("MYLIB/T1677RD4", "rr+, arrseq=N")) == NULL )
    {
        printf("Open failed\n");
        exit(1);
    }
    /* Update the first record in the keyed sequence. */
    _Rlocate(in, NULL, 0, __FIRST);
    _Rupdate(in, new_purchase, 20);
    /* Force the end of data. */
    _Rfeod(in);
}
```

Related Information

- “_Racquire() — Acquire a Program Device” on page 285
- “_Rfeov() — Force the End-of-File” on page 305

 `_Rfeov() — Force the End-of-File`

**Format**

```c
#include <recio.h>
int _Rfeov(_RFILE *fp);
```

**Language Level**

ILE C Extension

**Threading Capable**

Yes

**Description**

The `_Rfeov()` function forces an end-of-volume condition for a tape file that is associated with the file that is specified by `fp`. The `_Rfeov()` function positions the file to the next volume of the file. If the file is open for output, the output buffers will be flushed.

The `_Rfeov()` function is valid for tape files.
Return Value

The _Rfeov() function returns 1 if the file has moved from one volume to the next. It will return EOF if it is called while processing the last volume of the file. It will return zero if the operation is unsuccessful. The value of errno may be set to EIOERROR (a non-recoverable error occurred) or EIORECERR (a recoverable I/O error occurred). See Table 22 on page 543 and Table 24 on page 547 for errno settings.

Example

```c
#include <stdio.h>
#include <recio.h>
#include <stdlib.h>

int main(void)
{
    _RFILE *tape;
    _RFILE *fp;
    char   buf[92];
    int    i, feov2;

    /* Open source physical file containing C source. */
    if (( fp = _Ropen ( "QCSRC(T1677SRC)", "rr blkrcd=y" )) == NULL )
    {
        printf ( "could not open C source file\n" );
        exit ( 1 );
    }

    /* Open tape file to receive C source statements */
    if (( tape = _Ropen ( "T1677TPF", "wr lrecl=92 blkrcd=y" )) == NULL )
    {
        printf ( "could not open tape file\n" );
        exit ( 2 );
    }

    /* Read the C source statements, find their sizes */
    /* and add them to the tape file. */
    while ((_Rreadn ( fp, buf, sizeof(buf), __DFT )) -> num_bytes != EOF)
    {
        for ( i = sizeof(buf) - 1 ; buf[i] == ' ' && i > 12; --i );
        i = (i == 12) ? 80 : (1-12);
        memmove( buf, buf+12, i );
        _Rwrite ( tape, buf, i );
    }
    feov2 = _Rfeov (fp);

    _Rclose ( fp );
    _Rclose ( tape );
}
```

Related Information

• “_Racquire() — Acquire a Program Device” on page 285
• “_Rfeod() — Force the End-of-Data” on page 304

_Rformat() — Set the Record Format Name

Format

```c
#include <recio.h>

void _Rformat(_RFILE *fp, char *fmt);
```

Language Level

ILE C Extension
Threadsafe
Yes

Job CCSID Interface
All character data sent to this function is expected to be in the CCSID of the job. All character data returned by this function is in the CCSID of the job. See "Understanding CCSIDs and Locales" on page 565 for more information.

Description
The _Rformat() function sets the record format to \textit{fmt} for the file specified by \textit{fp}.

The \textit{fmt} parameter is a null-ended C string. The \textit{fmt} parameter must be in uppercase.

The _Rformat() function is valid for multi-format logical database, DDM files, display, ICF and printer files.

Return Value
The _Rformat() function returns \texttt{void}. See Table 22 on page 543 and Table 24 on page 547 for \texttt{errno} settings.

Example
This example shows how _Rformat() is used.
```c
#include <stdio.h>
#include <recio.h>
#include <stdlib.h>
#include <string.h>

int main(void)
{
    char       buf[40];
    int        rc = 1;
    _RFILE     *purf;
    _RFILE     *dailyf;

    /* Open purchase display file and daily transaction file */
    if ( ( purf = _Ropen ( "MYLIB/T1677RD3", "ar+,indicators=y" )) == NULL )
    {
        printf ( "Display file did not open.
" );
        exit ( 1 );
    }
    if ( ( dailyf = _Ropen ( "MYLIB/T1677RDA", "wr,commit=y" ) ) == NULL )
    {
        printf ( "Daily transaction file did not open.
" );
        exit ( 2 );
    }

    /* Select purchase record format */
    _Rformat ( purf, "PURCHASE" );

    /* Invite user to enter a purchase transaction. */
    /* The _Rwrite function writes the purchase display. */
    _Rwrite ( purf, "", 0 );
    _Rreadn ( purf, buf, sizeof(buf), __DFT );

    /* Update daily transaction file */
    rc = (( _Rwrite ( dailyf, buf, sizeof(buf) ))->num_bytes );

    /* If the databases were updated, then commit the transaction. */
    /* Otherwise, rollback the transaction and indicate to the */
    /* user that an error has occurred and end the application. */
    if ( rc )
    {
        _Rcommit ( "Transaction complete" );
    }
    else
    {
        _Rrollback ( );
        _Rformat ( purf, "ERROR" );
    }

    _Rclose ( purf );
    _Rclose ( dailyf );
}
```

Related Information

- “_Ropen() — Open a Record File for I/O Operations” on page 315

_Rindara() — Set Separate Indicator Area

**Format**

```c
#include <recio.h>

void _Rindara(_RFILE *fp, char *indic_buf);
```

**Language Level**

ILE C Extension
Threadsafe
No

Job CCSID Interface
All character data sent to this function is expected to be in the CCSID of the job. All character data returned by this function is in the CCSID of the job. See “Understanding CCSIDs and Locales” on page 565 for more information.

Description
The _Rindara() function registers indic_buf as the separate indicator area to be used by the file specified by fp. The file must be opened with the keyword indicators=Y on the _Ropen() function. The DDS for the file should specify also that a separate indicator area is to be used. It is generally best to initialize a separate indicator area explicitly with '0' (character) in each byte.

The _Rindara() function is valid for display, ICF, and printer files.

Return Value
The _Rindara() function returns void. See Table 22 on page 543 and Table 24 on page 547 for errno settings.
Example

```c
#include <stdio.h>
#include <recio.h>
#include <stdlib.h>
#include <string.h>
#define PF03     2
#define IND_OFF '0'
#define IND_ON  '1'

int main(void)
{
    char       buf[40];
    int        rc = 1;
    SYSindara ind_area;
    _RFILE    *purf;
    _RFILE    *dailyf;

    /* Open purchase display file and daily transaction file */
    if ( ( purf = _Ropen ( "MYLIB/T1677RD3", "ar+,indicators=y" )) == NULL )
    {
        printf ( "Display file did not open.
" );
        exit ( 1 );
    }
    if ( ( dailyf = _Ropen ( "MYLIB/T1677RDA", "wr,commit=y") ) == NULL )
    {
        printf ( "Daily transaction file did not open.
" );
        exit ( 2 );
    }

    /* Associate separate indicator area with purchase file */
    _Rindara ( purf, ind_area );
    /* Select purchase record format */
    _Rformat ( purf, "PURCHASE" );
    /* Invite user to enter a purchase transaction. */
    _Rwrite ( purf, "", 0 );
    /* The _Rwrite function writes the purchase display. */
    _Rreadn ( purf, buf, sizeof(buf), __DFT );
    /* While user is entering transactions, update daily and */
    /* monthly transaction files. */
    while ( rc && ind_area[PF03] == IND_OFF )
    {
        rc = (( _Rwrite ( dailyf, buf, sizeof(buf) ))->num_bytes );
        /* If the databases were updated, then commit transaction */
        /* otherwise, rollback the transaction and indicate to the */
        /* user that an error has occurred and end the application. */
        if ( rc )
        {  
            _Rcommit ( "Transaction complete" );
        }  
        else
        {   
            _Rrollback ( );
            _Rformat ( purf, "ERROR" );
        }
        _Rwrite ( purf, "", 0 );
        _Rreadn ( purf, buf, sizeof(buf), __DFT );
    }

    _Rclose ( purf );
    _Rclose ( dailyf );
}
```

Related Information

- “_Ropen() — Open a Record File for I/O Operations” on page 315

_Riofbk() — Obtain I/O Feedback Information

Format

```c
#include <recio.h>
#include <xxfdbk.h>
_XXIOFB_T *_Riofbk(_RFILE *fp);
```
**Language Level**

ILE C Extension

**Threadsafe**

Yes

**Description**

The `_Riofbk()` function returns a pointer to a copy of the I/O feedback area for the file that is specified by `fp`.

The `_Riofbk()` function is valid for all types of files.

**Return Value**

The `_Riofbk()` function returns NULL if an error occurs. See Table 22 on page 543 and Table 24 on page 547 for errno settings.

**Example**

```c
#include <stdio.h>
#include <recio.h>
#include <string.h>
#include <stdlib.h>

typedef struct {
    char name[20];
    char address[25];
} format1;
typedef struct {
    char name[8];
    char password[10];
} format2;
typedef union {
    format1 fmt1;
    format2 fmt2;
} formats;

int main(void)
{
    _RFILE *fp; /* File pointer */
    _RIOFB_T *rfb; /* Pointer to the file's feedback structure */
    _XXIOFB_T *iofb; /* Pointer to the file's feedback area */
    formats buf, in_buf, out_buf; /* Buffers to hold data */
    /* Open the device file. */
    if (( fp = _Ropen ( "MYLIB/T1677RD2", "ar+" )) == NULL )
    {
        printf ( "Could not open file\n" );
        exit ( 1 );
    }
    _Racquire ( fp,"DEVICE1" ); /* Acquire another device. Replace */
    /* with actual device name. */
    _Rformat ( fp,"FORMAT1" ); /* Set the record format for the */
    /* display file. */
    rfb = _Rwrite ( fp, ", 0"); /* Set up the display. */
    _Rpgmdev ( fp,"DEVICE2" ); /* Change the default program device. */
    /* Replace with actual device name. */
    _Rformat ( fp,"FORMAT2" ); /* Set the record format for the */
    /* display file. */
    rfb = _Rwrite ( fp, &buf, sizeof(buf) );
    rfb = _Rwriterd ( fp, &buf, sizeof(buf) );
    _Rreadindv ( fp, &buf, sizeof(buf), __DFT );
    /* Read from the first device that */
    /* enters data - device becomes */
    /* default program device. */
    /* Determine which terminal responded first. */
    iofb = _Riofbk ( fp );
    if ( !strncmp ( "FORMAT1  ", iofb -> rec_format, 10 ))
    {
        _Rrelease ( fp, "DEVICE1" );
    }
    else
    }
```
Related Information

- "_Ropnfbk() — Obtain Open Feedback Information" on page 319

_Rlocate() — Position a Record

Format

```
#include <recio.h>
_RIOFB_T *_Rlocate(_RFILE *fp, void *key, int klen_rrn, int opts);
```

Language Level

ILE C Extension

Threadsafe

Yes

However, if the file pointer is passed among threads, the I/O feedback area is shared among those threads.

Job CCSID Interface

All character data sent to this function is expected to be in the CCSID of the job. All character data returned by this function is in the CCSID of the job. See "Understanding CCSIDs and Locales" on page 565 for more information.

Description

The _Rlocate() function positions to the record in the file associated with fp and specified by the key, klen_rrn and opts parameters. The _Rlocate() function locks the record specified by the key, klen_rrn and opts parameters unless __NO_LOCK is specified.

The _Rlocate() function is valid for database and DDM files that are opened with the _Ropen() function. The following are valid parameters of the _Rlocate() function.

- **key**
  - Points to a string containing the key fields to be used for positioning.

- **klen_rrn**
  - Specifies the length of the key that is used if positioning by key or the relative record number if positioning by relative record number.

- **opts**
  - Specifies positioning options to be used for the locate operation. The possible macros are:
    - **__DFT**
      - Default to __KEY_EQ and lock the record for update if the file is open for updating.
    - **__END**
      - Positions to just after the last record in a file. There is no record that is associated with this position.
__END_FRC
Positions to just after the last record in a file. All buffered changes are made permanent. There is no record that is associated with this position.

__FIRST
Positions to the first record in the access path that is currently being used by fp. The key parameter is ignored.

__KEY_EQ
Positions to the first record with the specified key.

__KEY_GE
Positions to the first record that has a key greater than or equal to the specified key.

__KEY_GT
Positions to the first record that has a key greater than the specified key.

__KEY_LE
Positions to the first record that has a key less than or equal to the specified key.

__KEY_LT
Positions to the first record that has a key less than the specified key.

__KEY_NEXTEQ
Positions to the next record that has a key equal to the key value with a length of klen_rrn, at the current position. The key parameter is ignored.

__KEY_NEXTUNQ
Positions to the next record with a unique key from the current position in the access path. The key parameter is ignored.

__KEY_PREVEQ
Positions to the previous record with a key equal to the key value with a length of klen_rrn, at the current position. The key parameter is ignored.

__KEY_PREVUNQ
Positions to the previous record with a unique key from the current position in the access path. The key parameter is ignored.

__LAST
Positions to the last record in the access path that is currently being used by fp. The key parameter is ignored.

__NEXTH
Positions to the next record in the access path that is currently being used by fp. The key parameter is ignored.

__PREVIOUS
Positions to the previous record in the access path that is currently being used by fp. The key parameter is ignored.

__RRN_EQ
Positions to the record that has the relative record number specified on the klen_rrn parameter.

__START
Positions to just before the first record in the file. There is no record that is associated with this position.

__START_FRC
Positions to just before the first record in a file. There is no record that is associated with this position. All buffered changes are made permanent.

__DATA_ONLY
Positions to data records only. Deleted records will be ignored.

__KEY_NULL_MAP
The NULL key map is to be considered when locating to a record by key.

__NO_LOCK
The record that is positioned will not be locked.
The position of the file is not changed, but the located record will be locked if the file is open for update.

Positions to just before the requested record.

If you specify a start or end option (__START, __START_FRC, __END or __END_FRC) with any other options, the start or end option takes precedence and the other options might be ignored.

If you are positioned to __START or __END and perform a _Rreads operation, errno is set to EIOERROR.

Return Value

The _Rlocate() function returns a pointer to the _RIOFB_T structure associated with fp. If the _Rlocate() operation is successful, the num_bytes field contains 1. If __START, __START_FRC, __END or __END_FRC are specified, the num_bytes field is set to EOF. If the _Rlocate() operation is unsuccessful, the num_bytes field contains zero. The key and rrrn fields are updated, and the key field will contain the complete key even if a partial key is specified.

The value of errno may be set to:

<table>
<thead>
<tr>
<th>Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>EBADKEYLN</td>
<td>The key length that is specified is not valid.</td>
</tr>
<tr>
<td>ENOTREAD</td>
<td>The file is not open for read operations</td>
</tr>
<tr>
<td>EIOERROR</td>
<td>A non-recoverable I/O error occurred.</td>
</tr>
<tr>
<td>EIORECERR</td>
<td>A recoverable I/O error occurred.</td>
</tr>
</tbody>
</table>

See Table 22 on page 543 and Table 24 on page 547 for errno settings.

Example

```c
#include <stdio.h>
#include <stdlib.h>
#include <recio.h>

int main(void)
{
    _RFILE   *in;
    char     new_purchase[21] = "PEAR      1002022244"
    /* Open the file for processing in keyed sequence. */
    if ( (in = _Ropen("MYLIB/T1677RD4", "rr+, arrseq=N")) == NULL )
    {
        printf("Open failed\n");
        exit(1);
    }
    /* Update the first record in the keyed sequence. */
    _Rlocate(in, NULL, 0, __FIRST);
    _Rupdate(in, new_purchase, 20);
    /* Force the end of data. */
    _Rfeod(in);
    _Rclose(in);
}
```

Related Information

• “_Ropen() — Open a Record File for I/O Operations” on page 315
**_Ropen() — Open a Record File for I/O Operations**

**Format**

```c
#include <recio.h>
_RFILE *_Ropen(const char * filename, const char * mode, ...);
```

**Language Level**

ILE C Extension

**Threadsafe**

Yes

**Description**

The `_Ropen()` function opens the record file specified by `filename` according to the `mode` parameter, which may be followed by optional parameters, if the `varparm` keyword parameter is specified in the `mode` parameter. The open mode and keyword parameters may be separated by a comma and one or more spaces. The `_Ropen()` function does not dynamically create database files for you. All of the files you refer to in the `_Ropen()` function must exist, or the open operation will fail.

Files that are opened by the `_Ropen()` function are closed implicitly when the activation group they are opened in, is ended. If a pointer to a file opened in one activation group is passed to another activation group and the opening activation group is ended, the file pointer will no longer be valid.

The `_Ropen()` function applies to all types of files. The `filename` variable is any valid IBM i system file name.

The `mode` parameter specifies the type of access that is requested for the file. It contains an open mode that is followed by optional keyword parameters. The `mode` parameter may be one of the following values:

**Mode**

<table>
<thead>
<tr>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>rr</code></td>
</tr>
<tr>
<td><code>wr</code></td>
</tr>
<tr>
<td><code>ar</code></td>
</tr>
<tr>
<td><code>rr+</code></td>
</tr>
<tr>
<td><code>wr+</code></td>
</tr>
<tr>
<td><code>ar+</code></td>
</tr>
</tbody>
</table>

The `mode` may be followed by any of the following keyword parameters:

**Keyword**

<table>
<thead>
<tr>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>arrseq=value</code></td>
</tr>
</tbody>
</table>
Y  Specifies that the file is processed in arrival sequence.

N  Specifies that the file is processed using the access path that is used when the file was created.
   This is the default.

blkrcd=value
   Where value can be:

   Y  Performs record blocking. The operating system determines the most efficient block size for you.
      This parameter is valid for database, DDM, diskette and tape files. It is only valid for files opened
      for input-only or output-only (modes rr, wr, or ar).

   N  Does not perform record blocking. This is the default.

ccsid=value
   Specifies the CCSID that is used for translation of the file. The default is 0 which indicates that the job
   CCSID is used.

commit=value
   Where value can be:

   Y  Specifies that the database file is opened under commitment control. Commitment control must
      have been set up prior to this.

   N  Specifies that the database file is not opened under commitment control. This is the default.

dupkey=value
   value can be:

   Y  Duplicate key values will be flagged in the _RIOFB_T structure.

   N  Duplicate key values will not be flagged. This is the default.

indicators=value
   Indicators are valid for printer, display, and ICF files. value can be:

   Y  The indicators that are associated with the file are returned in a separate indicator area instead of
      in the I/O buffers.

   N  The indicators are returned in the I/O buffers. This is the default.

insertkeyfb=value
   Where value can be:

   Y  Specifies that insert key feedback is requested for keyed files opened for writing with arrseq=N.
      This is the default.

   N  Specifies that insert key feedback is not requested.

   Note: If the key field of the _RIOFB_T structure is never referenced by the application following
   writes, specifying insertkeyfb=N can result in better performance when writing. When the last I/O
   operation is writing, the _Rupfb() function will not update the _RIOFB_T structure with information
   about this writing operation if insertkeyfb=N is specified.

lrecl=value
   The length, in bytes, for fixed length records, and the maximum length for variable length records.
   This parameter is valid for diskette, display, printer, tape, and save files.
nullcap=value
Where value can be:

Y
The program is capable of handling null fields in records. This is valid for database and DDM files.

N
The program cannot handle null fields in records. This is the default.

cap=value
Where value can be:

Y
All fields in the _RIOFB_T structure are updated by any I/O operation that returns a pointer to the _RIOFB_T structure. However, the blk_filled_by field is not updated when using the _Rreadk function. This is the default.

N
Only the num_bytes field in the _RIOFB_T structure is updated.

rtncode=value
Where value can be:

Y
Use this option to bypass exception generation and handling. This will improve performance in the end-of-file and record-not-found cases. If the end-of-file is encountered, num_bytes will be set to EOF, but no errno values will be generated. If no record is found, num_bytes will be set to zero, and errno will be set to EIORECERR. This parameter is only valid for database and DDM files. For DDM files, num_bytes is not updated for _Rfeof.

N
The normal exception generation and handling process will occur for the cases of end-of-file and record-not-found. This is the default.

secure=value
Where value can be:

Y
Secures the file from overrides.

N
Does not secure the file from overrides. This is the default.

splfname=(value)
For spooled output only. Where value can be:

*FILE
The name of the printer file is used for the spooled output file name.

spool-file-name
Specify the name of the spooled output file. A maximum of 10 characters can be used.

usrdata=(value)
To specify, for spooled output only, user-specified data that identifies the file.

user-data
Specify up to 10 characters of user-specified text.

varparm=(list)
Where (list) is a list of optional keywords indicating which optional parameters will be passed to _Ropen(). The order of the keywords within the list indicates the order that the optional parameters will appear after the mode parameter. The following is a valid optional keyword:

lvlchk
The lvlchk keyword is used in conjunction with the lvlchk option on #pragma mapinc. When this keyword is used, a pointer to an object of type _LVLCHK_T (generated by #pragma mapinc) must be specified after the mode parameter on the _Ropen() function. For more details on this pointer, see the lvlchk option of #pragma mapinc in the ILE C/C++ Programmer's Guide.
**vlr=value**

Variable length record, where value is the minimum length of bytes of a record to be written to the file. The value can equal -1, or range from 0 to the maximum record length of the file. This parameter is valid for database and DDM files.

When VLR processing is required, _Ropen() will set min_length field. If the default value is not used, the minimum value that is provided by the user will be directly copied into min_length field. If the default value is specified, _Ropen() gets the minimum length from DB portion of the open data path.

**Return Value**

The _Ropen() function returns a pointer to a structure of type _RFILE if the file is opened successfully. It returns NULL if opening the file is unsuccessful.

The value of errno may be set to:

<table>
<thead>
<tr>
<th>Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>EBADMODE</td>
<td>The file mode that is specified is not valid.</td>
</tr>
<tr>
<td>EBADNAME</td>
<td>The file name that is specified is not valid.</td>
</tr>
<tr>
<td>ECONVERT</td>
<td>A conversion error occurred.</td>
</tr>
<tr>
<td>ENOTOPEN</td>
<td>The file is not open.</td>
</tr>
<tr>
<td>EIOERROR</td>
<td>A non-recoverable I/O error occurred.</td>
</tr>
<tr>
<td>EIORECERR</td>
<td>A recoverable I/O error occurred.</td>
</tr>
</tbody>
</table>

See Table 22 on page 543 and Table 24 on page 547 for errno settings.

**Example**

```c
#include <stdio.h>
#include <stdlib.h>
#include <recio.h>

int main(void)
{
    _RFILE      *fp;

    /* Open the file for processing in arrival sequence. */
    if (( fp = _Ropen ( "MYLIB/T1677RD1", "rr+, arrseq=Y" ) ) == NULL )
    {
        printf ( "Open failed\n" );
        exit ( 1 );
    } else
    /* Do some processing *;*/

    _Rclose ( fp );
}
```

**Related Information**

- “_Rclose() — Close a File” on page 286
- “<recio.h>” on page 7
_Ropnfbk() — Obtain Open Feedback Information

Format

```c
#include <recio.h>
#include <xxfdbk.h>

_XXOPFB_T _Ropnfbk(_RFILE *fp);
```

Language Level

ILE C Extension

Threadsafe

Yes

Description

The _Ropnfbk() function returns a pointer to a copy of the open feedback area for the file that is specified by `fp`.

The _Ropnfbk() function is valid for all types of files.

Return Value

The _Ropnfbk() function returns NULL if an error occurs. See Table 22 on page 543 and Table 24 on page 547 for errno settings.

Example

```c
#include <stdio.h>
#include <stdlib.h>
#include <recio.h>

int main(void)
{
  _RFILE      *fp;
  _XXOPFB_T   *opfb;

  /* Open the file for processing in arrival sequence. */
  if (( fp = _Ropen ( "MYLIB/T1677RD1", "rr+, arrseq=Y" )) == NULL )
  {
    printf ( "Open failed\n" );
    exit ( 1 );
  }

  /* Get the library and file names of the file opened. */
  opfb = _Ropnfbk ( fp );
  printf ( "Library: %10.10s\nFile:    %10.10s\n",
           opfb->library_name,
           opfb->file_name);

  _Rclose ( fp );
}
```

Related Information

- “_Rupfb() — Provide Information on Last I/O Operation” on page 346
Format

```c
#include <recio.h>
int _Rpgmdev(_RFILE *fp, char *dev);
```

Language Level

ILE C Extension

Threadsafe

No

Job CCSID Interface

All character data sent to this function is expected to be in the CCSID of the job. All character data returned by this function is in the CCSID of the job. See “Understanding CCSIDs and Locales” on page 565 for more information.

Description

The `_Rpgmdev()` function sets the current program device for the file that is associated with `fp` to `dev`. You must specify the device in uppercase.

The `dev` parameter is a null-ended C string.

The `_Rpgmdev()` function is valid for display, ICF, and printer files.

Return Value

The `_Rpgmdev()` function returns 1 if the operation is successful or zero if the device specified has not been acquired for the file. See Table 22 on page 543 and Table 24 on page 547 for errno settings.
Example

```c
#include <stdio.h>
#include <recio.h>
#include <string.h>
#include <stdlib.h>

typedef struct {
    char name[20];
    char address[25];
} format1;

typedef struct {
    char name[8];
    char password[10];
} format2;

typedef union {
    format1 fmt1;
    format2 fmt2;
} formats;

int main(void)
{
    _RFILE    *fp; /* File pointer */
    _RIOFB_T  *rfb; /* Pointer to the file's feedback structure */
    formats   buf, in_buf, out_buf; /* Buffers to hold data */
    
    /* Open the device file. */
    if (( fp = _Ropen ( "MYLIB/T1677RD2", "ar+" )) == NULL )
    {
        printf ( "Could not open file\n" );
        exit ( 1 );
    }

    _Rpgmdev ( fp,"DEVICE2" ); /* Change the default program device. */
    /* Replace with actual device name. */
    _Rformat ( fp,"FORMAT2" ); /* Set the record format for the display file. */

    rfb = _Rwrite ( fp, "", 0 ); /* Set up the display. */
    rfb = _Rwrread ( fp, &buf, sizeof(buf) );
    rfb = _Rwrread ( fp, &in_buf, sizeof(in_buf), &out_buf, sizeof(out_buf) );
    /* Continue processing. */

    _Rclose ( fp );
}
```

Related Information

- “_Racquire() — Acquire a Program Device” on page 285
- “_Rrelease() — Release a Program Device” on page 341

_Rreadd() — Read a Record by Relative Record Number

Format

```c
#include <recio.h>

_RIOFB_T *_Rreadd (_RFILE *fp, void *buf, size_t size,
                   int opts, long rrn);
```

Language Level

ILE C Extension
Threadsafe
Yes

However, if the file pointer is passed among threads, the I/O feedback area is shared among those threads.

Description
The _Rreadd() function reads the record that is specified by rrn in the arrival sequence access path for the file that is associated with fp. If the file is opened for updating, the _Rreadd() function locks the record specified by the rrn unless __NO_LOCK is specified. If the file is a keyed file, the keyed access path is ignored. Up to size number of bytes are copied from the record into buf (move mode only).

The following parameters are valid for the _Rreadd() function.

buf
Points to the buffer where the data that is read is to be stored. If locate mode is used, this parameter must be set to NULL.

size
Specifies the number of bytes that are to be read and stored in buf. If locate mode is used, this parameter is ignored.

rrn
The relative record number of the record to be read.

opts
Specifies the processing and access options for the file. The possible options are:

__DFT
If the file is opened for updating, then the record being read is locked for update. The previously locked record will no longer be locked.

__NO_LOCK
Does not lock the record being positioned to.

The _Rreadd() function is valid for database, DDM and display (subfiles) files.

Return Value
The _Rreadd() function returns a pointer to the _RIOFB_T structure associated with fp. If the _Rreadd() operation is successful the num_bytes field is set to the number of bytes transferred from the system buffer to the user's buffer (move mode) or the record length of the file (locate mode). If blkrcd=Y and riofb=Y are specified, the blk_count and the blk_filled_by fields of the _RIOFB_T structure are updated. The key and rrn fields are also updated. If the file associated with fp is a display file, the sysparm field is updated. If it is unsuccessful, the num_bytes field is set to a value less than size and errno will be changed.

The value of errno may be set to:

Value
Meaning
ENOTREAD
The file is not open for read operations.

ETRUNC
Truncation occurred on an I/O operation.

EIOERROR
A non-recoverable I/O error occurred.

EIORECERR
A recoverable I/O error occurred.

See Table 22 on page 543 and Table 24 on page 547 for errno settings.
Example

```c
#include <stdio.h>
#include <stdlib.h>
#include <recio.h>

int main(void)
{
    _RFILE *fp;
    _XXOPFB_T *opfb;
    /* Open the file for processing in arrival sequence. */
    if (( fp = _Ropen ( "MYLIB/T1677RD1", "rr+, arrseq=Y" )) == NULL )
    {
        printf ( "Open failed\n" );
        exit ( 1 );
    }
    /* Get the library and file names of the file opened. */
    opfb = _Ropnfbk ( fp );
    printf ( "Library: %10.10s
File:    %10.10s
", opfb->library_name,
             opfb->file_name);
    /* Get the second record. */
    _Rreadd ( fp, NULL, 20, __DFT, 2 );
    printf ( "Second record: %10.10s
", *(fp->in_buf) );
    _Rclose ( fp );
}
```

Related Information

- “_Rreadf() — Read the First Record” on page 323
- “_Rreadindv() — Read from an Invited Device” on page 325
- “_Rreadk() — Read a Record by Key” on page 328
- “_Rread() — Read the Last Record” on page 331
- “_Rreadn() — Read the Next Record” on page 332
- “_Rreadnc() — Read the Next Changed Record in a Subfile” on page 334
- “_Rreaddir() — Read the Previous Record” on page 337
- “_Rreads() — Read the Same Record” on page 339

_Rreadf() — Read the First Record

Format

```c
#include <recio.h>
_RIOFB_T *_Rreadf (_RFILE *fp, void *buf, size_t size, int opts);
```

Language Level

ILE C Extension

Threadsafely

Yes

However, if the file pointer is passed among threads, the I/O feedback area is shared among those threads.
Description
The _Rreadf() function reads the first record in the access path that is currently being used for the file specified by fp. The access path may be keyed sequence or arrival sequence. If the file is opened for updating, the _Rreadf() function locks the first record unless __NO_LOCK is specified. Up to size number of bytes are copied from the record into buf (move mode only).

The following are valid parameters for the _Rreadf() function.

_buf
This parameter points to the buffer where the data that is read is to be stored. If locate mode is used, this parameter must be set to NULL.

_size
This parameter specifies the number of bytes that are to be read and stored in buf. If locate mode is used, this parameter is ignored.

_opts
This parameter specifies the processing and access options for the file. The possible options are:

__DFT
If the file is opened for updating, then the record being read or positioned to is locked for update. The previously locked record will no longer be locked.

__NO_LOCK
Does not lock the record being positioned to.

The _Rreadf() function is valid for database and DDM files.

Return Value
The _Rreadf() function returns a pointer to the _RIOFB_T structure that is specified by fp. If the _Rreadf() operation is successful the num_bytes field is set to the number of bytes transferred from the system buffer to the user's buffer (move mode) or the record length of the file (locate mode). The key and rrn fields are updated. If record blocking is taking place, the blk_count and blk_filled_by fields are updated. The num_bytes field is set to EOF if the file is empty. If it is unsuccessful, the num_bytes field is set to a value less than size, and errno is changed.

The value of errno may be set to:

Value
Meaning
ENOTREAD
The file is not open for read operations.
ETRUNC
Truncation occurred on an I/O operation.
EIOERROR
A non-recoverable I/O error occurred.
EIORECERR
A recoverable I/O error occurred.

See Table 22 on page 543 and Table 24 on page 547 for errno settings.
Example

```c
#include <stdio.h>
#include <stdlib.h>
#include <recio.h>

int main(void)
{
    _RFILE      *fp;
    _XXOPFB_T   *opfb;

    /* Open the file for processing in arrival sequence. */
    if (( fp = _Ropen ( "MYLIB/T1677RD1", "rr+, arrseq=Y" )) == NULL )
    {
        printf ( "Open failed\n" );
        exit ( 1 );
    }

    /* Get the library and file names of the file opened. */
    opfb = _Ropnfbk ( fp );
    printf ( "Library: %10.10s
File:    %10.10s
", opfb->library_name,
             opfb->file_name);

    /* Get the first record. */
    _Rreadf ( fp, NULL, 20, __DFT );
    printf ( "First record:  %10.10s
", *(fp->in_buf) );

    /* Delete the first record. */
    _Rdelete ( fp );

    _Rclose ( fp );
}
```

Related Information

- “_Rreadd() — Read a Record by Relative Record Number” on page 321
- “_Rreadindv() — Read from an Invited Device” on page 325
- “_Rreadk() — Read a Record by Key” on page 328
- “_Rreadl() — Read the Last Record” on page 331
- “_Rreadn() — Read the Next Record” on page 332
- “_Rreadnc() — Read the Next Changed Record in a Subfile” on page 334
- “_Rreadp() — Read the Previous Record” on page 337
- “_Rreads() — Read the Same Record” on page 339

_Rreadindv() — Read from an Invited Device

Format

```c
#include <recio.h>
_RIOFB_T *_Rreadindv(_RFILE *fp, void *buf, size_t size, int opts);
```

Language Level

ILE C Extension

Threadsafe

No

Description

The _Rreadindv() function reads data from an invited device.
The following are valid parameters for the _Rreadindv() function.

**buf**
Points to the buffer where the data that is read is to be stored. If locate mode is used, this parameter must be set to NULL.

**size**
Specifies the number of bytes that are to be read and stored in *buf*. If locate mode is used, this parameter is ignored.

**opts**
Specifies the processing options for the file. Possible values are:

  __DFT
  If the file is opened for updating, then the record being read or positioned to is locked. Otherwise, the option is ignored.

The _Rreadindv() function is valid for display and ICF files.

**Return Value**
The _Rreadindv() function returns a pointer to the _RIOFB_T structure that is associated with *fp*. If the _Rreadindv() function is successful, the num_bytes field is set to the number of bytes transferred from the system buffer to the user's buffer (move mode) or the record length of the file (locate mode). The sysparm and rrn (for subfiles) fields are also updated. The num_bytes field is set to EOF if the file is empty. If the _Rreadindv() function is unsuccessful, the num_bytes field is set to a value less than the value of *size* and the errno will be changed.

The value of errno may be set to:

**Value**
**Meaning**

**ENOTREAD**
The file is not open for read operations.

**ETRUNC**
Truncation occurred on an I/O operation.

**EIOERROR**
A non-recoverable I/O error occurred.

**EIORECERR**
A recoverable I/O error occurred.

See Table 22 on page 543 and Table 24 on page 547 for errno settings.
#include <stdio.h>
#include <recio.h>
#include <string.h>
#include <stdlib.h>

typedef struct {
    char name[20];
    char address[25];
} format1;

typedef struct {
    char name[8];
    char password[10];
} format2;

typedef union {
    format1 fmt1;
    format2 fmt2;
} formats;

int main(void)
{
    _RFILE *fp;          /* File pointer */
    _RIOFB_T *rfb;       /* Pointer to the file's feedback structure */
    _XXIOFB_T *iofb;     /* Pointer to the file's feedback area */
    formats buf, in_buf, out_buf; /* Buffers to hold data */
    /* Open the device file. */
    if ((fp = _Ropen("MYLIB/T1677RD2", "ar+")) == NULL)
    {
        printf("Could not open file\n");
        exit(1);
    }
    _Racquire(fp,"DEVICE1"); /* Acquire another device. Replace */
    /* with actual device name. */
    _Rformat(fp,"FORMAT1");   /* Set the record format for the */
    /* display file. */
    rfb = _Rwrite(fp,"", 0);   /* Set up the display. */
    _Rpgmdev(fp,"DEVICE2");  /* Change the default program device. */
    _Rformat(fp,"FORMAT2");   /* Set the record format for the */
    /* display file. */
    rfb = _Rwrite(fp,"", 0);   /* Set up the display. */
    rfb = _Rwriterd(fp, &buf, sizeof(buf));
    rfb = _Rwrread(fp, &in_buf, sizeof(in_buf), &out_buf,
                   sizeof(out_buf));
    _Rreadindv(fp, &buf, sizeof(buf), __DFT);
    /* Read from the first device that */
    /* enters data - device becomes */
    /* default program device. */
    iofb = _Riofbk(fp);
    if (!strncmp("FORMAT1 ", iofb -> rec_format, 10))
    {
        _Rrelease(fp,"DEVICE1");
    }
    else
    {
        _Rrelease(fp,"DEVICE2");
    }
    /* Continue processing. */
    printf("Data displayed is \%45.45s\n", &buf);
    _Rclose(fp);
}

Related Information

• "_Rreadc() — Read a Record by Relative Record Number" on page 321
• "_Rreadf() — Read the First Record" on page 323
• "_Rreadk() — Read a Record by Key" on page 328
• "_Rreadl() — Read the Last Record" on page 331
• "_Rreadn() — Read the Next Record" on page 332
• "_Rreadnc() — Read the Next Changed Record in a Subfile" on page 334
• "_Rreadp() — Read the Previous Record" on page 337
• "_Rreads() — Read the Same Record" on page 339
**_Rreadk() — Read a Record by Key**

**Format**

```c
#include <recio.h>
_RIOFB_T *_Rreadk(_RFILE *fp, void *buf, size_t size,
    int opts, void *key, unsigned int keylen);
```

**Language Level**

ILE C Extension

**Threadsafe**

Yes

However, if the file pointer is passed among threads, the I/O feedback area is shared among those threads.

**Description**

The _Rreadk() function reads the record in the keyed access path that is currently being used for the file that is associated with `fp`. Up to `size` number of bytes are copied from the record into `buf` (move mode only). If the file is opened for updating, the _Rreadk() function locks the record positioned to unless __NO_LOCK is specified. You must be processing the file using a keyed sequence path.

The following parameters are valid for the _Rreadk() function.

- **buf**
  Points to the buffer where the data that is read is to be stored. If locate mode is used, this parameter must be set to NULL.

- **size**
  Specifies the number of bytes that are to be read and stored in `buf`. If locate mode is used, this parameter is ignored.

- **key**
  Points to the key to be used for reading.

- **keylen**
  Specifies the total length of the key to be used.

- **opts**
  Specifies the processing options for the file. Possible values are:
    - **__DFT**
      Default to __KEY_EQ.
    - **__KEY_EQ**
      Positions to and reads the first record that has the specified key.
    - **__KEY_GE**
      Positions to and reads the first record that has a key greater than or equal to the specified key.
    - **__KEY_GT**
      Positions to and reads to the first record that has a key greater than the specified key.
    - **__KEY_LE**
      Positions to and reads the first record that has a key less than or equal to the specified key.
    - **__KEY_LT**
      Positions to and reads the first record that has a key less than the specified key.
__KEY_NEXTEQ
Positions to and reads the next record that has a key equal to the key value at the current position. The key parameter is ignored.

__KEY_NEXTUNQ
Positions to and reads the next record with a unique key from the current position in the access path. The key parameter is ignored.

__KEY_PREVEQ
Positions to and reads the last record that has a key equal to the key value at the current position. The key parameter is ignored.

__KEY_PREVUNQ
Positions to and reads the previous record with a unique key from the current position in the access path. The key parameter is ignored.

__NO_LOCK
Do not lock the record for updating.

The positioning options are mutually exclusive.
The following options may be combined with the positioning options using the bit-wise OR (|) operator.

__KEY_NULL_MAP
The NULL key map is to be considered when reading a record by key.

__NO_LOCK
The record that is positioned will not be locked.

The _Rreadk() function is valid for database and DDM files.

Return Value

The _Rreadk() function returns a pointer to the _RIOFB_T structure associated with fp. If the _Rreadk() operation is successful the num_bytes field is set to the number of bytes transferred from the system buffer to the user's buffer (move mode) or the record length of the file (locate mode). The key and rrn fields will be updated. The key field will always contain the complete key if a partial key is specified. When using record blocking with _Rreadk(), only one record is read into the block. Thus there are zero records remaining in the block and the blk_count field of the _RIOFB_T structure will be updated with 0. The blk_filled_by field is not applicable to _Rreadk() and is not updated. If the record specified by key cannot be found, the num_bytes field is set to zero or EOF. If you are reading a record by a partial key, then the entire key is returned in the feedback structure. If it is unsuccessful, the num_bytes field is set to a value less than size and errno will be changed.

The value of errno may be set to:

<table>
<thead>
<tr>
<th>Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>EBADKEYLN</td>
<td>The key length specified is not valid.</td>
</tr>
<tr>
<td>ENOTREAD</td>
<td>The file is not open for read operations.</td>
</tr>
<tr>
<td>ETRUNC</td>
<td>Truncation occurred on an I/O operation.</td>
</tr>
<tr>
<td>EIOERROR</td>
<td>A non-recoverable I/O error occurred.</td>
</tr>
<tr>
<td>EIORECERR</td>
<td>A recoverable I/O error occurred.</td>
</tr>
</tbody>
</table>

See Table 22 on page 543 and Table 24 on page 547 for errno settings.
```c
#include <stdio.h>
#include <recio.h>
#include <stdlib.h>

int main(void)
{
    _RFILE *fp;
    _RIOFB_T *fb;
    char buf[4];
    /* Create a physical file */
    system("CRTPF FILE(QTEMP/MY_FILE)");
    /* Open the file for write */
    if ( (fp = _Ropen("QTEMP/MY_FILE", "wr")) == NULL )
    {
        printf("open for write fails\n");
        exit(1);
    }
    /* write some records into the file */
    _Rwrite(fp, "KEY9", 4);
    _Rwrite(fp, "KEY8", 4);
    _Rwrite(fp, "KEY7", 4);
    _Rwrite(fp, "KEY6", 4);
    _Rwrite(fp, "KEY5", 4);
    _Rwrite(fp, "KEY4", 4);
    _Rwrite(fp, "KEY3", 4);
    _Rwrite(fp, "KEY2", 4);
    _Rwrite(fp, "KEY1", 4);
    /* Close the file */
    _Rclose(fp);
    /* Open the file for read */
    if ( (fp = _Ropen("QTEMP/MY_FILE", "rr")) == NULL )
    {
        printf("open for read fails\n");
        exit(2);
    }
    /* Read the record with key KEY3 */
    fb = _Rreadk(fp, buf, 4, __KEY_EQ, "KEY3", 4);
    printf("record %d with value %4.4s\n", fb->rrn, buf);
    /* Read the next record with key less than KEY3 */
    fb = _Rreadk(fp, buf, 4, __KEY_LT, "KEY3", 4);
    printf("record %d with value %4.4s\n", fb->rrn, buf);
    /* Read the next record with key greater than KEY3 */
    fb = _Rreadk(fp, buf, 4, __KEY_GT, "KEY3", 4);
    printf("record %d with value %4.4s\n", fb->rrn, buf);
    /* Read the next record with different key */
    fb = _Rreadk(fp, buf, 4, __KEY_NEXTUNQ, "", 4);
    printf("record %d with value %4.4s\n", fb->rrn, buf);
    /* Close the file */
    _Rclose(fp);
}
```

Related Information

- "_Rreadd() — Read a Record by Relative Record Number" on page 321
- "_Rreadf() — Read the First Record" on page 323
- "_Rreadindv() — Read from an Invited Device" on page 325
- "_Rreadl() — Read the Last Record" on page 331
- "_Rreadn() — Read the Next Record" on page 332
- "_Rreadnc() — Read the Next Changed Record in a Subfile" on page 334
- "_Rreadp() — Read the Previous Record" on page 337
- "_Rreads() — Read the Same Record" on page 339
Format

```c
#include <recio.h>
_RIOFB_T *_Rreadl(_RFILE *fp, void *buf, size_t size, int opts);
```

Language Level

ILE C Extension

Threadsafe

Yes

However, if the file pointer is passed among threads, the I/O feedback area is shared among those threads.

Description

The _Rreadl() function reads the last record in the access path currently being used for the file specified by fp. The access path may be keyed sequence or arrival sequence. Up to size number of bytes are copied from the record into buf (move mode only). If the file is opened for updating, the _Rreadl() function locks the last record unless __NO_LOCK is specified.

The following parameters are valid for the _Rreadl() function.

- **buf**
  - Points to the buffer where the data that is read is to be stored. If locate mode is used, this parameter must be set to NULL.

- **size**
  - Specifies the number of bytes that are to be read and stored in buf. If locate mode is used, this parameter is ignored.

- **opts**
  - Specifies the processing options for the file. Possible values are:
    - **__DFT**
      - If the file is opened for updating, then the record being read or positioned to is locked. The previously locked record will no longer be locked.
    - **__NO_LOCK**
      - Do not lock the record being positioned to.

The _Rreadl() function is valid for database and DDM files.

Return Value

The _Rreadl() function returns a pointer to the _RIOFB_T structure that is associated with fp. If the _Rreadl() operation is successful the num_bytes field is set to the number of bytes transferred from the system buffer to the user's buffer (move mode) or the record length of the file (locate mode). The key and rrn fields will be updated. If record blocking is taking place, the blk_count and blk_filled_by fields will be updated. If the file is empty, the num_bytes field is set to EOF. If it is unsuccessful, the num_bytes field is set to a value less than size and errno will be changed.

The value of errno may be set to:

- **ENOTREAD**
  - The file is not open for read operations.
ETRUNC
Truncation occurred on an I/O operation.

EIOERROR
A non-recoverable I/O error occurred.

EIORECERR
A recoverable I/O error occurred.

See Table 22 on page 543 and Table 24 on page 547 for errno settings.

Example

```c
#include <stdio.h>
#include <stdlib.h>
#include <recio.h>

int main(void)
{
    _RFILE      *fp;
    _XXOPFB_T   *opfb;

    /* Open the file for processing in arrival sequence. */
    if (( fp = _Ropen ( "MYLIB/T1677RD1", "rr+, arrseq=Y" )) == NULL )
    {
        printf ( "Open failed\n" );
        exit ( 1 );
    }

    /* Get the library and file names of the file opened. */
    opfb = _Ropnfbk ( fp );
    printf ( "Library: %10.10s
File:    %10.10s
", opfb->library_name,
            opfb->file_name);

    /* Get the last record. */
    _Rreadl ( fp, NULL, 20, __DFT );
    printf ( "Last record: %10.10s
", *(fp->in_buf) );

    _Rclose ( fp );
}
```

Related Information

- “_Rreadd() — Read a Record by Relative Record Number” on page 321
- “_Rreadf() — Read the First Record” on page 323
- “_Rreadindv() — Read from an Invited Device” on page 325
- “_Rreadk() — Read a Record by Key” on page 328
- “_Rreadn() — Read the Next Record” on page 332
- “_Rreadnc() — Read the Next Changed Record in a Subfile” on page 334
- “_Rreadp() — Read the Previous Record” on page 337
- “_Rreads() — Read the Same Record” on page 339

_Rreadn() — Read the Next Record

Format

```c
#include <recio.h>
_RIOFB_T *_Rreadn (_RFILE *fp, void *buf, size_t size, int opts);
```

Language Level

ILE C Extension

332 IBM i: ILE C/C++ Runtime Library Functions
Threadsafe
Yes

However, if the file pointer is passed among threads, the I/O feedback area is shared among those threads.

Description
The _Rreadn() function reads the next record in the access path that is currently being used for the file that is associated with fp. The access path may be keyed sequence or arrival sequence. Up to size number of bytes are copied from the record into buf (move mode only). If the file is opened for updating, the _Rreadn() function locks the record positioned to unless __NO_LOCK is specified.

If the file associated with fp is opened for sequential member processing and the current record position is the last record of any member in the file except the last, _Rreadn() will read the first record in the next member of the file.

If an _Rlocate() operation positioned to a record specifying the __PRIOR option, _Rreadn() will read the record positioned to by the _Rlocate() operation.

If the file is open for record blocking and a call to _Rreadp() has filled the block, the _Rreadn() function is not valid if there are records remaining in the block. You can check the blk_count in _RIOFB_T to see if there are any remaining records.

The following are valid parameters for the _Rreadn() function.

buf
Points to the buffer where the data that is read is to be stored. If locate mode is used, this parameter must be set to NULL.

size
Specifies the number of bytes that are to be read and stored in buf. If locate mode is used, this parameter is ignored.

opts
Specifies the processing options for the file. Possible values are:

__DFT
If the file is opened for updating, then the record being read or positioned to is locked. The previously locked record will no longer be locked.

__NO_LOCK
Do not lock the record being positioned to.

The _Rreadn() function is valid for all types of files except printer files.

Return Value
The _Rreadn() function returns a pointer to the _RIOFB_T structure that is associated with fp. If the _Rreadn() operation is successful the num_bytes field is set to the number of bytes transferred from the system buffer to the user's buffer (move mode) or the record length of the file (locate mode). The key and rrn fields are updated. If the file that is associated with fp is a display file, the sysparm field is also updated. If record blocking is taking place, the blk_count and the blk_filled_by fields of the _RIOFB_T structure are updated. If attempts are made to read beyond the last record in the file, the num_bytes field is set to EOF. If it is unsuccessful, the num_bytes field is set to a value less than size, and errno is changed. If you are using device files and specify zero as the size, check errno to determine if the function was successful.

The value of errno may be set to:

Value
Meaning
ENOTREAD
The file is not open for read operations.
ETRUNC
   Truncation occurred on an I/O operation.

EIOERROR
   A non-recoverable I/O error occurred.

EIORECERR
   A recoverable I/O error occurred.

See Table 22 on page 543 and Table 24 on page 547 for errno settings.

Example

```c
#include <stdio.h>
#include <stdlib.h>
#include <recio.h>

int main(void)
{
   _RFILE      *fp;
   _XXOPFB_T   *opfb;
   /* Open the file for processing in arrival sequence. */
   if (( fp = _Ropen ( "MYLIB/T1677RD1", "rr+, arrseq=Y" )) == NULL )
     {
      printf ( "Open failed\n" );
      exit ( 1 );
   }
   /* Get the library and file names of the file opened. */
   opfb = _Ropnfbk ( fp );
   printf ( "Library: %10.10s\nFile:    %10.10s\n",
            opfb->library_name,
            opfb->file_name);
   /* Get the first record. */
   _Rreadf ( fp, NULL, 20, __DFT );
   printf ( "First record:  %10.10s\n", *(fp->in_buf) );
   /* Delete the second record. */
   _Rreadn ( fp, NULL, 20, __DFT );
   _Rdelete ( fp );
   _Rclose ( fp );
}
```

Related Information

- “_Rreadd() — Read a Record by Relative Record Number” on page 321
- “_Rreadf() — Read the First Record” on page 323
- “_Rreadindv() — Read from an Invited Device” on page 325
- “_Rreadk() — Read a Record by Key” on page 328
- “_Rreadl() — Read the Last Record” on page 331
- “_Rreadnc() — Read the Next Changed Record in a Subfile” on page 334
- “_Rreadp() — Read the Previous Record” on page 337
- “_Rreads() — Read the Same Record” on page 339

_Rreadnc() — Read the Next Changed Record in a Subfile

**Format**

```c
#include <recio.h>

_RIOFB_T *_Rreadnc(_RFILE *fp, void *buf, size_t size);
```
Language Level
ILE C Extension

Threadsafe
No

Description
The \_R\texttt{readnc()} function reads the next changed record from the current position in the subfile that is associated with \texttt{fp}. The minimum size of data that is read from the screen are copied from the system buffer to \texttt{buf}.

The following are valid parameters for the \_R\texttt{readnc()} function.

- **buf**
  Points to the buffer where the data that is read is to be stored. If locate mode is used, this parameter must be set to NULL.

- **size**
  Specifies the number of bytes that are to be read and stored in \texttt{buf}.

The \_R\texttt{readnc()} function is valid for subfiles.

Return Value
The \_R\texttt{readnc()} function returns a pointer to the \_R\texttt{IOFB_T} structure that is associated with \texttt{fp}. If the \_R\texttt{readnc()} operation is successful the num\_bytes field is set to the number of bytes transferred from the system buffer to the user's buffer (move mode) or the record length of the file (locate mode). The rrn and sysparm fields are updated. If there are no changed records between the current position and the end of the file, the num\_bytes field is set to EOF. If it is unsuccessful, the num\_bytes field is set to a value less than size, and errno is changed.

The value of errno may be set to:

- **Value**
  **Meaning**
- **ENOTREAD**
  The file is not open for read operations.
- **ETRUNC**
  Truncation occurred on an I/O operation.
- **EIOERROR**
  A non-recoverable I/O error occurred.
- **EIORECERR**
  A recoverable I/O error occurred.

See Table 22 on page 543 and Table 24 on page 547 for errno settings.
Example

```c
#include <stdio.h>
#include <stdlib.h>
#include <recio.h>
#define LEN          10
#define NUM_RECS     20
#define SUBFILENAME  "MYLIB/T1677RD6"
#define PFFILENAME   "MYLIB/T1677RD6"
typedef struct {
    char name[LEN];
    char phone[LEN];
} pf_t;
#define RECLEN sizeof(pf_t)
void init_subfile(_RFILE *, _RFILE *);

int main(void)
{
    _RFILE *pf, *subf;
    /***********************************************************************
    * Open the subfile and the physical file.                             *
    ***********************************************************************/
    if ((pf = _Ropen(PFFILENAME, "rr")) == NULL) {
        printf("can't open file %s\n", PFFILENAME);
        exit(1);
    }
    if ((subf = _Ropen(SUBFILENAME, "ar+")) == NULL) {
        printf("can't open file %s\n", SUBFILENAME);
        exit(2);
    }
    /**********************************************************************
    * Initialize the subfile with records                                *
    * from the physical file.                                            *
    ***********************************************************************/
    init_subfile(pf, subf);
    /***********************************************************************
    * Write the subfile to the display by writing a record to the subfile control format. *
    ***********************************************************************/
    _Rformat(subf, "SFLCTL");
    _Rwrite(subf, "", 0);
    _Rreadnc(subf, "", 0);
    /***********************************************************************
    * Close the physical file and the subfile.                          *
    ***********************************************************************
    _Rclose(pf);
    _Rclose(subf);
}

Related Information

- "_Rreadd() — Read a Record by Relative Record Number" on page 321
- "_Rreadf() — Read the First Record" on page 323
- "_Rreadindv() — Read from an Invited Device" on page 325
- "_Rreadk() — Read a Record by Key" on page 328
- "_Rreadl() — Read the Last Record" on page 331
- "_Rreadn() — Read the Next Record" on page 332
- "_Rreadp() — Read the Previous Record" on page 337
- "_Rreads() — Read the Same Record" on page 339
_Rreadp() — Read the Previous Record

Format

```c
#include <recio.h>
_RIOFB_T *_Rreadp(_RFILE *fp, void *buf, size_t size, int opts);
```

Language Level
ILE C Extension

Threadsafe
Yes
However, if the file pointer is passed among threads, the I/O feedback area is shared among those threads.

Description
The _Rreadp() function reads the previous record in the access path that is currently being used for the file that is associated with fp. The access path may be keyed sequence or arrival sequence. Up to size number of bytes are copied from the record into buf (move mode only). If the file is opened for updating, the _Rreadp() function locks the record positioned to unless __NO_LOCK is specified.

If the file associated with fp is opened for sequential member processing and the current record position is the first record of any member in the file except the first, _Rreadp() will read the last record in the previous member of the file.

If the file is open for record blocking and a call to _Rreadn() has filled the block, the _Rreadp() function is not valid if there are records remaining in the block. You can check the blk_count in _RIOFB_T to see if there are any remaining records.

The following are valid parameters for the _Rreadp() function.

- **buf**
  Points to the buffer where the data that is read is to be stored. If locate mode is used, this parameter must be set to NULL.

- **size**
  Specifies the number of bytes that are to be read and stored in buf. If locate mode is used, this parameter is ignored.

- **opts**
  Specifies the processing options for the file. Possible values are:

  - __DFT
    If the file is opened for updating, then the record being read or positioned to is locked. The previously locked record will no longer be locked.

  - __NO_LOCK
    Do not lock the record being positioned to.

The _Rreadp() function is valid for database and DDM files.

Return Value
The _Rreadp() function returns a pointer to the _RIOFB_T structure that is associated with fp. If the _Rreadp() operation is successful the num_bytes field is set to the number of bytes transferred from the system buffer to the user's buffer (move mode) or the record length of the file (locate mode). The key and rrr fields are also updated. If record blocking is taking place, the blk_count and the blk_filled_by fields of the _RIOFB_T structure are updated. If attempts are made to read prior to the first record in the
file, the num_bytes field is set to EOF. If it is unsuccessful, the num_bytes field is set to a value less than size, and errno is changed.

The value of errno may be set to:

<table>
<thead>
<tr>
<th>Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>ENOTREAD</td>
<td>The file is not open for read operations.</td>
</tr>
<tr>
<td>ETRUNC</td>
<td>Truncation occurred on an I/O operation.</td>
</tr>
<tr>
<td>EIOERROR</td>
<td>A non-recoverable I/O error occurred.</td>
</tr>
<tr>
<td>EIORECERR</td>
<td>A recoverable I/O error occurred.</td>
</tr>
</tbody>
</table>

See Table 22 on page 543 and Table 24 on page 547 for errno settings.

Example

```c
#include <stdio.h>
#include <stdlib.h>
#include <recio.h>

int main(void) {
  _RFILE  *fp;
  _XXOPFB_T *opfb;

  /* Open the file for processing in arrival sequence. */
  if ((fp = _Ropen("MYLIB/T1677RD1", "rr+, arrseq=Y")) == NULL) {
    printf("Open failed\n");
    exit(1);
  }

  /* Get the library and file names of the file opened. */
  opfb = _Ropnfbk(fp);
  printf("Library: %10.10s\nFile:    %10.10s\n",
         opfb->library_name,
         opfb->file_name);

  /* Get the last record. */
  _Rreadl(fp, NULL, 20, __DFT);
  printf("Last record: %10.10s\n", *(fp->in_buf));

  /* Get the previous record. */
  _Rreadp(fp, NULL, 20, __DFT);
  printf("Next to last record: %10.10s\n", *(fp->in_buf));

  _Rclose(fp);
}
```

Related Information

- "_Rreadd() — Read a Record by Relative Record Number" on page 321
- "_Rreadf() — Read the First Record" on page 323
- "_Rreadindv() — Read from an Invited Device" on page 325
- "_Rreadk() — Read a Record by Key" on page 328
- "_Rreadl() — Read the Last Record" on page 331
- "_Rreadn() — Read the Next Record" on page 332
- "_Rreadnc() — Read the Next Changed Record in a Subfile" on page 334
- "_Rreads() — Read the Same Record" on page 339
**_Rreads() — Read the Same Record**

**Format**

```c
#include <recio.h>
_RIOFB_T *_Rreads(_RFILE *fp, void *buf, size_t size, int opts);
```

**Language Level**

ILE C Extension

**Threadsafe**

Yes

However, if the file pointer is passed among threads, the I/O feedback area is shared among those threads.

**Description**

The _Rreads() function reads the current record in the access path that is currently being used for the file that is associated with fp. The access path may be keyed sequence or arrival sequence. Up to size number of bytes are copied from the record into buf (move mode only). If the file is opened for updating, the _Rreads() function locks the record positioned to unless __NO_LOCK is specified.

If the current position in the file that is associated with fp has no record associated with it, the _Rreads() function will fail.

The _Rreads() function is not valid when the file is open for record blocking.

The following are valid parameters for the _Rreads() function.

- **buf**
  Points to the buffer where the data that is read is to be stored. If locate mode is used, this parameter must be set to NULL.

- **size**
  Specifies the number of bytes that are to be read and stored in buf. If locate mode is used, this parameter is ignored.

- **opts**
  Specifies the processing options for the file. Possible values are:

  - **__DFT**
    If the file is opened for updating, then the record being read or positioned to is locked. The previously locked record will no longer be locked.

  - **__NO_LOCK**
    Do not lock the record being positioned to.

The _Rreads() function is valid for database and DDM files.

**Return Value**

The _Rreads() function returns a pointer to the _RIOFB_T structure that is associated with fp. If the _Rreads() operation is successful the num_bytes field is set to the number of bytes transferred from the system buffer to the user's buffer (move mode) or the record length of the file (locate mode). The key and rrn fields are also updated. If it is unsuccessful, the num_bytes field is set to a value less than size, and errno is changed.

The value of errno may be set to:
**Value**

**Meaning**

**ENOTREAD**
   The file is not open for read operations.

**ETRUNC**
   Truncation occurred on an I/O operation.

**EIOERROR**
   A non-recoverable I/O error occurred.

**EIORECERR**
   A recoverable I/O error occurred.

See Table 22 on page 543 and Table 24 on page 547 for errno settings.

**Example**

```c
#include <stdlib.h>
#include <recio.h>

int main(void)
{
   _RFILE      *fp;
   _XXOPFB_T   *opfb;

   /* Open the file for processing in arrival sequence. */
   if (( fp = _Ropen ( "MYLIB/T1677RD1", "rr+, arrseq=Y" )) == NULL )
   {
      printf ( "Open failed\n" );
      exit ( 1 );
   }

   /* Get the library and file names of the file opened. */
   opfb = _Ropnfbk ( fp );
   printf ( "Library: %10.10s\nFile: %10.10s\n",
            opfb->library_name,
            opfb->file_name);

   /* Get the last record. */
   _Rreadl ( fp, NULL, 20, __DFT );
   printf ( "Last record: %10.10s\n", *(fp->in_buf) );

   /* Get the same record without locking it. */
   _Rreads ( fp, NULL, 20, __NO_LOCK);
   printf ( "Same record: %10.10s\n", *(fp->in_buf) );

   _Rclose ( fp );
}
```

**Related Information**

- “_Rreadd() — Read a Record by Relative Record Number” on page 321
- “_Rreadf() — Read the First Record” on page 323
- “_Rreadindv() — Read from an Invited Device” on page 325
- “_Rreadk() — Read a Record by Key” on page 328
- “_Rreadl() — Read the Last Record” on page 331
- “_Rreadn() — Read the Next Record” on page 332
- “_Rreadnc() — Read the Next Changed Record in a Subfile” on page 334
- “_Rreadp() — Read the Previous Record” on page 337
**_Rrelease() — Release a Program Device**

**Format**

```c
#include <recio.h>

int _Rrelease(_RFILE *fp, char *dev);
```

**Language Level**

ILE C Extension

**Threadsafe**

No

**Job CCSID Interface**

All character data sent to this function is expected to be in the CCSID of the job. All character data returned by this function is in the CCSID of the job. See “Understanding CCSIDs and Locales” on page 565 for more information.

**Description**

The `_Rrelease()` function releases the program device that is specified by `dev` from the file that is associated with `fp`. The device name must be specified in uppercase.

The `dev` parameter is a null-ended C string.

The `_Rrelease()` function is valid for display and ICF files.

**Return Value**

The `_Rrelease()` function returns 1 if it is successful or zero if it is unsuccessful. The value of errno may be set to EIOERROR (a non-recoverable I/O error occurred) or EIORECERR (a recoverable I/O error occurred). See Table 22 on page 543 and Table 24 on page 547 for errno settings.

**Example**

```c
#include <stdio.h>
#include <recio.h>
#include <string.h>
#include <stdlib.h>

typedef struct {
    char name[20];
    char address[25];
} format1;

typedef struct {
    char name[8];
    char password[10];
} format2;

typedef union {
    format1 fmt1;
    format2 fmt2;
} formats;

int main(void)
{
    _RFILE   *fp; /* File pointer */
    _RIOFB_T *rfb; /* Pointer to the file's feedback structure */
    _XXIOFB_T *iofb; /* Pointer to the file's feedback area */
    formats  buf, in_buf, out_buf; /* Buffers to hold data */
    /* Open the device file. */
    if (( fp = _Ropen ( "MYLIB/T1677RD2", "ar+" )) == NULL )
    {
        printf ( "Could not open file\n" );
        exit (1);
    }
```


Related Information

- "_Racquire() — Acquire a Program Device" on page 285

_Rrlslck() — Release a Record Lock

Format

```c
#include <recio.h>

int _Rrlslck(_RFILE *fp);
```

Language Level

ILE C Extension

Threatsafe

Yes

Description

The _Rrlslck() function releases the lock on the currently locked record for the file specified by fp. The file must be open for update, and a record must be locked. If the _NO_POSITION option was specified on the _Rlocate() operation that locked the record, the record released may not be the record currently positioned to.

The _Rrlslck() function is valid for database and DDM files.

Return Value

The _Rrlslck() function returns 1 if the operation is successful, or zero if the operation is unsuccessful. The value of errno may be set to:
<table>
<thead>
<tr>
<th>Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>ENOTUPD</td>
<td>The file is not open for update operations.</td>
</tr>
<tr>
<td>EIOERROR</td>
<td>A non-recoverable I/O error occurred.</td>
</tr>
<tr>
<td>EIORECERR</td>
<td>A recoverable I/O error occurred.</td>
</tr>
</tbody>
</table>

See Table 22 on page 543 and Table 24 on page 547 for errno settings.

Example

```c
#include <stdio.h>
#include <stdlib.h>
#include <recio.h>

int main(void)
{
    char        buf[21];
    _FILE      *fp;
    _XXOPFB_T   *opfb;
    int         result;

    /* Open the file for processing in arrival sequence. */
    if (( fp = _Ropen ( "MYLIB/T1677RD1", "rr+, arrseq=Y" )) == NULL )
    {
        printf ( "Open failed\n" );
        exit ( 1 );
    }

    /* Get the library and file names of the file opened. */
    opfb = _Ropnfbk ( fp );
    printf ( "Library: %10.10s File: %10.10s\n",
             opfb->library_name,
             opfb->file_name);

    /* Get the last record. */
    _Rreadl ( fp, NULL, 20, __DFT );
    printf ( "Last record: %10.10s\n", *(fp->in_buf) );

    /* _Rrlslck example. */
    result = _Rrlslck ( fp );
    if ( result == 0 )
        printf("_Rrlslck failed.\n");

    _Rclose ( fp );
}
```

Related Information

- “_Rdelete() — Delete a Record” on page 289

_Rrollbck() — Roll Back Commitment Control Changes

Format

```c
#include <recio.h>
int _Rrollbck(void);
```

Language Level

ILE C Extension
Threadsafe
No

Description
The _Rrollback() function reestablishes the last commitment boundary as the current commitment boundary. All changes that are made to the files under commitment control in the job, are reversed. All locked records are released. Any file that is open under commitment control in the job will be affected. You must specify the keyword parameter commit=y when the file is opened to be under commitment control. A commitment control environment must have been set up prior to this.

The _Rrollback() function is valid for database and DDM files.

Return Value
The _Rrollback() function returns 1 if the operation is successful or zero if the operation is unsuccessful. The value of errno may be set to EIOERROR (a non-recoverable I/O error occurred) or EIORECERR (a recoverable I/O error occurred). See Table 22 on page 543 and Table 24 on page 547 for errno settings.

Example

```c
#include <stdio.h>
#include <recio.h>
#include <stdlib.h>
#include <string.h>

int main(void)
{
    char       buf[40];
    int        rc = 1;
    _RFILE     *purf;
    _RFILE     *dailyf;

    /* Open purchase display file and daily transaction file */
    if ( ( purf = _Ropen ( "MYLIB/T1677RD3", "ar+,indicators=y" )) == NULL )
    {
        printf ( "Display file did not open.\n" );
        exit ( 1 );
    }
    if ( ( dailyf = _Ropen ( "MYLIB/T1677RDA", "wr,commit=y" ) ) == NULL )
    {
        printf ( "Daily transaction file did not open.\n" );
        exit ( 2 );
    }

    /* Select purchase record format */
    _Rformat ( purf, "PURCHASE" );

    /* Invite user to enter a purchase transaction. */
    _Rwrite ( purf, "", 0 );
    _Rreadn ( purf, buf, sizeof(buf), __DFT );

    /* Update daily transaction file */
    rc = (( _Rwrite ( dailyf, buf, sizeof(buf) ))->num_bytes);

    /* If the databases were updated, then commit the transaction. */
    /* Otherwise, rollback the transaction and indicate to the */
    /* user that an error has occurred and end the application. */
    if ( rc )
    {
        _Rcommit ( "Transaction complete" );
    }
    else
    {
        _Rrollback ( );
        _Rformat ( purf, "ERROR" );
    }

    _Rclose ( purf );
}```
_Rupdate() — Update a Record

**Format**

```c
#include <recio.h>
_RIOFB_T *_Rupdate(_RFILE *fp, void *buf, size_t size);
```

**Language Level**

ILE C Extension

**Threadsafe**

Yes

However, if the file pointer is passed among threads, the I/O feedback area is shared among those threads.

**Description**

The `_Rupdate()` function updates the record that is currently locked for update in the file that is specified by `fp`. The file must be open for update. A record is locked for update by reading or locating to it unless `_NO_LOCK` is specified on the read or locate operation. If the `_NO_POSITION` option is specified on a locate operation the record updated may not be the record currently positioned to. After the update operation, the updated record is no longer locked.

The number of bytes that are copied from `buf` to the record is the minimum of `size` and the record length of the file (move mode only). If `size` is greater than the record length, the data is truncated, and `errno` is set to ETRUNC. One complete record is always written to the file. If the `size` is less than the record length of the file, the remaining data in the record will be the original data that was read into the system buffer by the read that locked the record. If a locate operation locked the record, the remaining data will be what was in the system input buffer prior to the locate.

The `_Rupdate()` function can be used to update deleted records and key fields. A deleted record that is updated will no longer be marked as a deleted record. In both of these cases any keyed access paths defined for `fp` will be changed.

**Note:** If locate mode is being used, `_Rupdate()` works on the data in the file's input buffer.

The `_Rupdate()` function is valid for database, display (subfiles) and DDM files.

**Return Value**

The `_Rupdate()` function returns a pointer to the `_RIOFB_T` structure associated with `fp`. If the `_Rupdate()` function is successful, the num_bytes field is set to the number of bytes transferred from the system buffer to the user's buffer (move mode) or the record length of the file (locate mode). If `fp` is a display file, the sysparm field is updated. If the `_Rupdate()` function is unsuccessful, the num_bytes field is set to a value less than the size specified (move mode) or zero (locate mode). The `errno` value will also be changed.

The value of `errno` may be set to:
**Value**
**Meaning**

**ENOTUPD**
The file is not open for update operations.

**EIOERROR**
A non-recoverable I/O error occurred.

**EIORECERR**
A recoverable I/O error occurred.

See Table 22 on page 543 and Table 24 on page 547 for errno settings.

**Example**

```c
#include <stdio.h>
#include <stdlib.h>
#include <recio.h>

int main(void)
{
    _RFILE   *in;
    char     new_purchase[21] = "PEAR      1002022244";
    /* Open the file for processing in keyed sequence. */
    if ( (in = _Ropen("MYLIB/T1677RD4", "rr+, arrseq=N")) == NULL )
    {
        printf("Open failed\n");
        exit(1);
    };
    /* Update the first record in the keyed sequence. */
    _Rlocate(in, NULL, 0, __FIRST);
    _Rupdate(in, new_purchase, 20);
    /* Force the end of data. */
    _Rfeod(in);
    _Rclose(in);
}
```

**Related Information**
- “_Rreadd() — Read a Record by Relative Record Number” on page 321
- “_Rreadf() — Read the First Record” on page 323
- “_Rreadindv() — Read from an Invited Device” on page 325
- “_Rreadk() — Read a Record by Key” on page 328
- “_Rreadl() — Read the Last Record” on page 331
- “_Rreadd() — Read the Next Record” on page 332
- “_Rreadnc() — Read the Next Changed Record in a Subfile” on page 334
- “_Rreadp() — Read the Previous Record” on page 337
- “_Rreadp() — Read the Same Record” on page 339

**_Rupfb() — Provide Information on Last I/O Operation**

**Format**

```c
#include <recio.h>
_RIOFB_T *_Rupfb(_RFILE *fp);
```
Language Level
ILE C Extension

Threadsafe
Yes
However, if the file pointer is passed among threads, the I/O feedback area is shared among those threads.

Description
The _Rupfb() function updates the feedback structure associated with the file specified by fp with information about the last I/O operation. The _RIOFB_T structure will be updated even if riofb=N was specified when the file was opened. The num_bytes field of the _RIOFB_T structure will not be updated. See “<recio.h>” on page 7 for a description of the _RIOFB_T structure.

The _Rupfb() function is valid for all types of files.

Return Value
The _Rupfb() function returns a pointer to the _RIOFB_T structure specified by fp. See Table 22 on page 543 and Table 24 on page 547 for errno settings.

Example
```c
#include <stdio.h>
#include <recio.h>
#include <stdlib.h>

int main(void)
{
  _RFILE *fp;
  _RIOFB_T *fb;
  /* Create a physical file */
  system("CRTPF FILE(QTEMP/MY_FILE) RCDLEN(80)");
  /* Open the file for write */
  if ( (fp = _Ropen("QTEMP/MY_FILE", "wr")) == NULL )
    
      printf("open for write fails\n");
      exit(1);
    
    /* Write some records into the file */
    _Rwrite(fp, "This is record 1", 16);
    _Rwrite(fp, "This is record 2", 16);
    _Rwrite(fp, "This is record 3", 16);
    _Rwrite(fp, "This is record 4", 16);
    _Rwrite(fp, "This is record 5", 16);
    _Rwrite(fp, "This is record 6", 16);
    _Rwrite(fp, "This is record 7", 16);
    _Rwrite(fp, "This is record 8", 16);
    _Rwrite(fp, "This is record 9", 16);
  /* Close the file */
  _Rclose(fp);
  /* Open the file for read */
  if ( (fp = _Ropen("QTEMP/MY_FILE", "rr, blkrcd = y") == NULL )
    
      printf("open for read fails\n");
      exit(2);
    
    /* Read some records */
    _Rreadn(fp, NULL, 80, __DFT);
    _Rreadn(fp, NULL, 80, __DFT);
    /* Call _Rupfb and print feedback information */
    fb = _Rupfb(fp);
    printf("record number -------------------------- %d\n", fb->rrn);
    printf("number of bytes read ------------------- %d\n", fb->num_bytes);
    printf("number of records remaining in block --- %hd\n", fb->blk_count);
    if ( fb->blk_filled_by == __READ_NEXT )
```

Library Functions 347
Related Information

- "_Ropnfbk() — Obtain Open Feedback Information" on page 319

_Rwrite() — Write the Next Record

Format

```c
#include <recio.h>
_UINT32_ T * _Rwrite(_RFILE *fp, void *buf, size_t size);
```

Language Level

ILE C Extension

Threadsafe

Yes

However, if the file pointer is passed among threads, the I/O feedback area is shared among those threads.

Description

The _Rwrite() function has two modes: move and locate. When `buf` points to a user buffer, _Rwrite() is in move mode. When `buf` is NULL, the function is in locate mode.

The _Rwrite() function appends a record to the file specified by `fp`. The number of bytes copied from `buf` to the record is the minimum of `size` and the record length of the file (move mode only). If `size` is greater than the record length, the data is truncated and `errno` is set to ETRUNC. One complete record is always written if the operation is successful.

If you are using _Ropen() and then _Rwrite() to output records to a source physical file, the sequence numbers must be manually appended.

The _Rwrite() function has no effect on the position of the file for a subsequent read operation.

Records might be lost although the _Rwrite() function indicates success when the following items are true:

- Record blocking is taking place.
- The file associated with `fp` is approaching the limit of the number of records it can contain and the file cannot be extended.
- Multiple writers are writing to the same file.

Because the output is buffered, the _Rwrite() function returns success that indicates the record is successfully copied to the buffer. However, when the buffer is flushed, the function might fail because the file has been filled to capacity by another writer. In this case, the _Rwrite() function indicates that an error occurred only on the call to the _Rwrite() function that sends the data to the file.

The _Rwrite() function is valid for all types of files.
Return Value

The _Rwrite() function returns a pointer to the _RIOFB_T structure that is associated with fp. If the _Rwrite() operation is successful the num_bytes field is set to the number of bytes written for both move mode and locate mode. The function transfers the bytes from the user's buffer to the system buffer. If record blocking is taking place, the function only updates the rrn and key fields when it sends the block to the database. If fp is a display, ICF or printer file, the function updates the sysparm field. If it is unsuccessful, the num_bytes field is set to a value less than size specified (move mode) or zero (locate mode) and errno is changed.

The value of errno may be set to:

<table>
<thead>
<tr>
<th>Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>ENOTWRITE</td>
<td>The file is not open for write operations.</td>
</tr>
<tr>
<td>ETRUNC</td>
<td>Truncation occurred on an I/O operation.</td>
</tr>
<tr>
<td>EIOERROR</td>
<td>A non-recoverable I/O error occurred.</td>
</tr>
<tr>
<td>EIORECERR</td>
<td>A recoverable I/O error occurred.</td>
</tr>
</tbody>
</table>

See Table 22 on page 543 and Table 24 on page 547 for errno settings.

Example

```c
#include <stdio.h>
#include <recio.h>
#include <string.h>
#include <stdlib.h>

typedef struct {
    char name[20];
    char address[25];
} format1;

typedef struct {
    char name[8];
    char password[10];
} format2;

typedef union {
    format1 fmt1;
    format2 fmt2;
} formats;

int main(void)
{
    _RFILE *fp; /* File pointer */
    _RIOFB_T *rfb; /* Pointer to the file's feedback structure */
    _XXIOFB_T *iofb; /* Pointer to the file's feedback area */
    formats buf, in_buf, out_buf; /* Buffers to hold data */

    /* Open the device file. */
    if (( fp = _Ropen ( "MYLIB/T1677RD2", "ar+" )) == NULL )
    {
        printf ( "Could not open file\n" );
        exit ( 1 );
    }
    _Racquire ( fp,"DEVICE1" ); /* Acquire another device. Replace*/
    _Rformat ( fp,"FORMAT1" ); /* with actual device name. */
    _Rwrite ( fp, "", 0 ); /* Set the record format for the */
    _Rpgmdev ( fp,"DEVICE2" ); /* display file. */
    _Rformat ( fp,"FORMAT2" ); /* Change the default program device. */
    _Rwrite ( fp, "", 0 ); /* Replace with actual device name. */
    _Rreadindv ( fp, &buf, sizeof(buf), __DFT ); /* Read from the first device that */
/* enters data - device becomes default program device. */
/* Determine which terminal responded first. */
iofb = _Riofbk ( fp );
if ( !strncmp ( "FORMAT1 ", iofb -> rec_format, 10 ))
{
    _Rrelease ( fp, "DEVICE1" );
}
else
{
    _Rrelease(fp, "DEVICE2" );
}
/* Continue processing. */
printf ( "Data displayed is %45.45s\n", &buf);
_Rclose ( fp );

Related Information

• "_Rwrited() — Write a Record Directly” on page 350
• "_Rwriterd() — Write and Read a Record” on page 353
• "_Rwrread() — Write and Read a Record (separate buffers)” on page 354

_Rwrited() — Write a Record Directly

Format

#include <recio.h>
_RIOFB_T *_Rwrited(_RFILE *fp, void *buf, size_t size, unsigned long rrn);

Language Level

ILE C Extension

Threatsafe

Yes

However, if the file pointer is passed among threads, the I/O feedback area is shared among those threads.

Description

The _Rwrited() function writes a record to the file associated with fp at the position specified by rrn. The _Rwrited() function will only write over deleted records. The number of bytes copied from buf to the record is the minimum of size and the record length of the file (move mode only). If size is greater than the record length, the data is truncated, and errno is set to ETRUNC. One complete record is always written if the operation is successful.

The _Rwrited() function has no effect on the position of the file for a read operation.

The _Rwrited() function is valid for database, DDM and subfiles.

Return Value

The _Rwrited() function returns a pointer to the _RIOFB_T structure associated with fp. If the _Rwrited() operation is successful the num_bytes field is set to the number of bytes transferred from the user's buffer to the system buffer (move mode) or the record length of the file (locate mode). The rrn field is updated. If fp is a display file, the sysparm field is updated. If it is unsuccessful, the num_bytes field is set to a value less than size specified (move mode) or zero (locate mode) and errno is changed.
The value of errno may be set to:

<table>
<thead>
<tr>
<th>Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>ENOTWRITE</td>
<td>The file is not open for write operations.</td>
</tr>
<tr>
<td>ETRUNC</td>
<td>Truncation occurred on an I/O operation.</td>
</tr>
<tr>
<td>EIOERROR</td>
<td>A non-recoverable I/O error occurred.</td>
</tr>
<tr>
<td>EIORECERR</td>
<td>A recoverable I/O error occurred.</td>
</tr>
</tbody>
</table>

See Table 22 on page 543 and Table 24 on page 547 for errno settings.
Example

```c
#include <stdio.h>
#include <stdlib.h>
#include <recio.h>
#define LEN    10
#define NUM_RECS 20
#define SUBFILENAME "MYLIB/T1677RD6"
#define PFILENAME "MYLIB/T1677RDB"
typedef struct {
    char name[LEN];
    char phone[LEN];
} pf_t;
#define RECLEN sizeof(pf_t)

void init_subfile(_RFILE *, _RFILE *);
int main(void)
{
    _RFILE   *pf;
    _RFILE   *subf;
    /* Open the subfile and the physical file. */
    if ((pf = _Ropen(PFILENAME, "rr")) == NULL) {
        printf("can't open file %s\n", PFILENAME);
        exit(1);
    }
    if ((subf = _Ropen(SUBFILENAME, "ar+")) == NULL) {
        printf("can't open file %s\n", SUBFILENAME);
        exit(2);
    }
    /* Initialize the subfile with records */
    init_subfile(pf, subf);
    /* Write the subfile to the display by writing */
    /* a record to the subfile control format. */
    _Rformat(subf, "SFLCTL");
    _Rwrite(subf, "", 0);
    _Rreadnc(subf, ",", 0);
    /* Close the physical file and the subfile. */
    _Rclose(pf);
    _Rclose(subf);
}

void init_subfile(_RFILE *pf, _RFILE *subf)
{
    _RIOFB_T     *fb;
    int           i;
    pf_t          record;
    /* Select the subfile record format. */
    _Rformat(subf, "SFL");
    for (i = 1; i <= NUM_RECS; i++) {
        fb = _Rreadn(pf, &record, RECLEN, __DFT);
        if (fb->num_bytes != RECLEN) {
            printf("%d\n", fb->num_bytes);
            printf("%d\n", RECLEN);
            printf("error occurred during read\n");
            exit(3);
        }
        fb = _Rwrited(subf, &record, RECLEN, i);
        if (fb->num_bytes != RECLEN) {
            printf("error occurred during write\n");
            exit(4);
        }
    }
}
```

Related Information

- "_Rwrite() — Write the Next Record" on page 348
- "_Rwriterd() — Write and Read a Record" on page 353
- "_Rwrread() — Write and Read a Record (separate buffers)" on page 354
Rwriterd() — Write and Read a Record

Format

```c
#include <recio.h>
_RIOFB_T *_Rwriterd(_FILE *fp, void *buf, size_t size);
```

Language Level

ILE C Extension

Threadsafe

No

Description

The _Rwriterd() function performs a write and then a read operation on the file that is specified by `fp`. The minimum of `size` and the length of the current record format determines the amount of data to be copied between the system buffer and `buf` for both the write and read parts of the operation. If `size` is greater than the record length of the current format, `errno` is set to ETRUNC on the write part of the operation. If `size` is less than the length of the current record format, `errno` is set to ETRUNC on the read part of the operation.

The _Rwriterd() function is valid for display and ICF files.

Return Value

The _Rwriterd() function returns a pointer to the _RIOFB_T structure that is associated with `fp`. If the _Rwriterd() operation is successful, the num_bytes field is set to the number of bytes transferred from the system buffer to `buf` on the read part of the operation (move mode) or the record length of the file (locate mode).

The value of `errno` may be set to:

<table>
<thead>
<tr>
<th>Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>ENOTUPD</td>
<td>The file is not open for update operations.</td>
</tr>
<tr>
<td>ETRUNC</td>
<td>Truncation occurred on an I/O operation.</td>
</tr>
<tr>
<td>EIOERROR</td>
<td>A non-recoverable I/O error occurred.</td>
</tr>
<tr>
<td>EIORECERR</td>
<td>A recoverable I/O error occurred.</td>
</tr>
</tbody>
</table>

See Table 22 on page 543 and Table 24 on page 547 for `errno` settings.
Example

```c
#include <stdio.h>
#include <recio.h>
#include <string.h>
#include <stdlib.h>

typedef struct {
    char name[20];
    char address[25];
} format1;

typedef struct {
    char name[8];
    char password[10];
} format2;

typedef union {
    format1 fmt1;
    format2 fmt2;
} formats;

int main(void)
{
    _RFILE *fp; /* File pointer */
    _RIOFB_T *rfb; /* Pointer to the file's feedback structure */
    formats buf, in_buf, out_buf; /* Buffers to hold data */

    /* Open the device file. */
    if (( fp = _Ropen( "MYLIB/T1677RD2", "ar+" )) == NULL )
    {
        printf( "Could not open file\n" );
        exit( 1 );
    }

    _Rpgmdev ( fp,"DEVICE2" ); /* Change the default program device. */
    _Rformat ( fp,"FORMAT2" ); /* Set the record format for the display file. */

    rfb = _Rwrite ( fp, "", 0 ); /* Set up the display. */
    rfb = _Rwrited ( fp, &buf, sizeof(buf) );
    rfb = _Rwrread ( fp, &in_buf, sizeof(in_buf), &out_buf,
                    sizeof(out_buf) );

    /* Continue processing. */

    _Rclose ( fp );
}
```

Related Information

- “_Rwrite() — Write the Next Record” on page 348
- “_Rwrited() — Write a Record Directly” on page 350
- “_Rwrread() — Write and Read a Record (separate buffers)” on page 354

_Rwrread() — Write and Read a Record (separate buffers)

**Format**

```c
#include <recio.h>

_RIOFB_T *_Rwrread(_RFILE *fp, void *in_buf, size_t in_buf_size,
                    void *out_buf, size_t out_buf_size);
```

**Language Level**

ILE C Extension
Description
The _Rwrread() function performs a write and then a read operation on the file that is specified by *fp*. Separate buffers may be specified for the input and output data. The minimum of size and the length of the current record format determines the amount of data to be copied between the system buffer and the buffers for both the write and read parts of the operation. If *out_buf_size* is greater than the record length of the current format, errno is set to ETRUNC on the write part of the operation. If *in_buf_size* is less than the length of the current record format, errno is set to ETRUNC on the read part of the operation.

The _Rwrread() function is valid for display and ICF files.

Return Value
The _Rwrread() function returns a pointer to the _RIOFB_T structure that is associated with *fp*. If the _Rwrread() operation is successful, the num_bytes field is set to the number of bytes transferred from the system buffer to *in_buf* in the read part of the operation (move mode) or the record length of the file (locate mode).

The value of errno may be set to:

<table>
<thead>
<tr>
<th>Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>ENOTUPD</td>
<td>The file is not open for update operations.</td>
</tr>
<tr>
<td>ETRUNC</td>
<td>Truncation occurred on an I/O operation.</td>
</tr>
<tr>
<td>EIOERROR</td>
<td>A non-recoverable I/O error occurred.</td>
</tr>
<tr>
<td>EIORECERR</td>
<td>A recoverable I/O error occurred.</td>
</tr>
</tbody>
</table>

See Table 22 on page 543 and Table 24 on page 547 for errno settings.
Example

```c
#include <stdio.h>
#include <recio.h>
#include <string.h>
#include <stdlib.h>

typedef struct {
     char name[20];
     char address[25];
} format1;

typedef struct {
     char name[8];
     char password[10];
} format2;

typedef union {
     format1 fmt1;
     format2 fmt2;
} formats;

int main(void)
{
    _RFILE *fp; /* File pointer */
    _RIOFB_T *rfb; /* Pointer to the file's feedback structure */
    formats buf, in_buf, out_buf; /* Buffers to hold data */

    /* Open the device file. */
    if (( fp = _Ropen ( "MYLIB/T1677RD2", "ar+" )) == NULL )
    {
        printf ( "Could not open file\n" );
        exit ( 1 );
    }

    _Rpgmdev ( fp,"DEVICE2" );/* Change the default program device. */
    /* Replace with actual device name. */

    _Rformat ( fp,"FORMAT2" ); /* Set the record format for the */
    /* display file. */

    rfb = _Rwrite ( fp,"","", 0 ); /* Set up the display. */
    rfb = _Rwriterd ( fp, &buf, sizeof(buf) );
    rfb = _Rwrread ( fp,&in_buf, sizeof(in_buf), &out_buf,
                   sizeof(out_buf) );

    /* Continue processing. */

    _Rclose ( fp );
}
```

Related Information

- “_Rwrite() — Write the Next Record” on page 348
- “_Rwrited() — Write a Record Directly” on page 350
- “_Rwriterd() — Write and Read a Record” on page 353

**samequantumd32() - samequantumd64() - samequantumd128() —
Determine if Quantum Exponents X and Y are the Same**

Format

```c
#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);
```
The `samequantumd32()`, `samequantumd64()`, and `samequantumd128()` functions determine if the quantum exponents of `x` and `y` are the same. If both `x` and `y` are NaN or both `x` and `y` are infinity, they have the same quantum exponents. If exactly one operand is infinity or exactly one operand is NaN, they do not have the same quantum exponents. The `samequantumd32()`, `samequantumd64()`, and `samequantumd128()` functions raise no floating-point exceptions.

**Return Value**

The `samequantumd32()`, `samequantumd64()`, and `samequantumd128()` functions return true when `x` and `y` have the same quantum exponents, and false otherwise.

### Example

This example illustrates the use of the `samequantumd64()` function.

```c
#define __STDC_WANT_DEC_FP__
#include <math.h>
#include <stdio.h>
#include <stdlib.h>

static void dump_value(_Decimal64 val1, _Decimal64 val2)
{
    printf( " quantexp(x)=%d  quantexp(y)=%d  samequantum=%d\n",
            quantexpd64(val1), quantexpd64(val2),
            (int)samequantumd64(val1, val2) );
}

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=%-8Da y=%-8Da\n", a1, a2 );
    dump_value(a1, a2);
    printf( "x=%-8Da y=%-8Da\n", b1, b2 );
    dump_value(b1, b2);
    printf( "x=%-8Da y=%-8Da\n", c1, c2 );
    dump_value(c1, c2);
    printf( "x=%-8Da y=%-8Da\n", d1, d2 );
    dump_value(d1, d2);
    return 0;
}
```

```
/* Output should be similar to: */
x=1.23     y=0.01
quantexp(x)=2  quantexp(y)=2  samequantum=1
```

```
x=1.234    y=0.01
quantexp(x)=3  quantexp(y)=2  samequantum=0
```

```
x=1.000    y=1.00
quantexp(x)=3  quantexp(y)=2  samequantum=0
```

```
x=0.000    y=0.00
quantexp(x)=3  quantexp(y)=2  samequantum=0
*/
Related Information
- “quantized32() - quantized64() - quantized128() — Set the Quantum Exponent of X to the Quantum Exponent of Y” on page 271
- “quantexpd32() - quantexpd64() - quantexpd128() — Compute the Quantum Exponent” on page 270

scanf() — Read Data

Format

```c
#include <stdio.h>
int scanf(const char *format-string, argument-list);
```

Language Level
ANSI

Threadsafe
Yes

Locale Sensitive
The behavior of this function might be affected by the LC_CTYPE and LC_NUMERIC categories of the current locale. The behavior might also be affected by the LC_UNI_CTYPE category of the current locale if LOCALETYPE(*LOCALEUCS2) or LOCALETYPE(*LOCALEUTF) is specified on the compilation command. For more information, see “Understanding CCSIDs and Locales” on page 565.

Description

The `scanf()` function reads data from the standard input stream stdin into the locations given by each entry in argument-list. Each argument must be a pointer to a variable with a type that corresponds to a type specifier in format-string. The format-string controls the interpretation of the input fields, and is a multibyte character string that begins and ends in its initial shift state.

The format-string can contain one or more of the following:

- White-space characters, as specified by the `isspace()` function (such as blanks and new-line characters). A white-space character causes the `scanf()` function 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-whitespace character causes the `scanf()` function to read, but not to store, a matching non-whitespace character. If the next character in stdin does not match, the `scanf()` function ends.
- Format specifications, introduced by the percent sign (%). A format specification causes the `scanf()` function to read and convert characters in the input into values of a specified type. The value is assigned to an argument in the argument list.

The `scanf()` function reads format-string from left to right. Characters outside of format specifications are expected to match the sequence of characters in stdin; the matched characters in stdin are scanned but not stored. If a character in stdin conflicts with format-string, `scanf()` ends. The conflicting character is left in stdin as if it had not been read.

When the first format specification is found, the value of the first input field is converted according to the format specification and stored in the location specified by the first entry in argument-list. The second format specification converts the second input field and stores it in the second entry in argument-list, and so on through the end of format-string.
An input field is defined as all characters up to the first white-space character (space, tab, or new line), up to the first character that cannot be converted according to the format specification, or until the field width is reached, whichever comes first. If there are too many arguments for the format specifications, the extra arguments are ignored. The results are undefined if there are not enough arguments for the format specifications.

A format specification has the following form:

```
% [ * width ] h l L lll H D DD type
```

Each field of the format specification is a single character or a number signifying a particular format option. The type character, 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 format specification contains only the percent sign and a type character (for example, %s).

Each field of the format specification is discussed in detail below. If a percent sign (%) is followed by a character that has no meaning as a format control character, the behavior is undefined. One exception to this behavior is %%. To specify a percent-sign character, use %%%.

The following restrictions apply to pointer printing and scanning:

- If a pointer is printed out and scanned back from the same activation group, the scanned back pointer will be compared equal to the pointer that is printed out.
- If a `scanf()` family function scans a pointer that was printed out by a different activation group, the `scanf()` family function will set the pointer to NULL.

See the *ILE C/C++ Programmer's Guide* for more information about using IBM i pointers.

An asterisk (*) following the percent sign suppresses assignment of the next input field, which is interpreted as a field of the specified type. The field is scanned but not stored.

The width is a positive decimal integer controlling the maximum number of characters to be read from stdin. 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 new line), or a character that cannot be converted according to the given format occurs before width is reached.

The optional size modifiers h, l, ll, H, D, and DD indicate the size of the receiving object. The conversion characters d, i, and n must be preceded by h if the corresponding argument is a pointer to a short int rather than a pointer to an int, by l if it is a pointer to a long int, or by ll if it is a pointer to a long long int. Similarly, the conversion characters o, u, x, and X must be preceded by h if the corresponding argument is a pointer to an unsigned short int rather than a pointer to an unsigned int, by l if it is a pointer to an unsigned long int, or by ll if it is a pointer to an unsigned long long int. The conversion characters a, A, e, E, f, F, g, and G must be preceded by l if the corresponding argument is a pointer to a double rather than a pointer to a float, by L if it is a pointer to a long double, by H if it is a pointer to a _Decimal32, by D if it is a pointer to a _Decimal64, or by DD if it is a pointer to a _Decimal128. Finally, the conversion characters c, s, and [ must be preceded by l if the corresponding argument is a pointer to a wchar_t rather than a pointer to a single-byte character type. If an h, l, ll, H, D, or DD appears with any other conversion character, the behavior is undefined.

The type characters and their meanings are in the following table:
<table>
<thead>
<tr>
<th>Character</th>
<th>Type of Input Expected</th>
<th>Type of Argument</th>
</tr>
</thead>
<tbody>
<tr>
<td>d</td>
<td>Signed decimal integer</td>
<td>Pointer to int.</td>
</tr>
<tr>
<td>o</td>
<td>Unsigned octal integer</td>
<td>Pointer to unsigned int.</td>
</tr>
<tr>
<td>x, X</td>
<td>Unsigned hexadecimal integer</td>
<td>Pointer to unsigned int.</td>
</tr>
<tr>
<td>i</td>
<td>Decimal, hexadecimal, or octal integer</td>
<td>Pointer to int.</td>
</tr>
<tr>
<td>u</td>
<td>Unsigned decimal integer</td>
<td>Pointer to unsigned int.</td>
</tr>
<tr>
<td>a, A, E, f, F, g, G</td>
<td>For non decimal floating-point numbers, an optionally signed floating-point number, infinity, or NaN, whose format is the same as expected for the <code>strtod()</code> function. For decimal floating-point numbers, an optionally signed floating-point number, infinity, or NaN, whose format is the same as expected for the <code>strtod64()</code> function.</td>
<td>Pointer to floating point.</td>
</tr>
<tr>
<td>D(n,p)</td>
<td>Packed decimal value consisting of an optional sign (+ or -); then a non-empty sequence of digits, optionally a series of one or more decimal digits possibly containing a decimal point, but not a decimal suffix. The subject sequence is defined as the longest initial subsequence of the input string, starting with the first non-whitespace character, in the expected form. It contains no characters if the input string is empty or consists entirely of white space, or if the first non-whitespace character is anything other than a sign, a digit, or a decimal point character.</td>
<td>Pointer to decimal(n,p). Since the internal representation of the binary coded decimal object is the same as the internal representation of the packed decimal data type, you can use the type character D(n,p).</td>
</tr>
<tr>
<td>c</td>
<td>Character; white-space characters that are ordinarily skipped are read when c is specified</td>
<td>Pointer to char large enough for input field.</td>
</tr>
<tr>
<td>s</td>
<td>String</td>
<td>Pointer to character array large enough for input field plus a ending null character (\0), which is automatically appended.</td>
</tr>
<tr>
<td>n</td>
<td>No input read from stream or buffer</td>
<td>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 <code>scanf()</code>.</td>
</tr>
<tr>
<td>p</td>
<td>Pointer to void converted to series of characters</td>
<td>Pointer to void.</td>
</tr>
<tr>
<td>lc</td>
<td>Multibyte character constant</td>
<td>Pointer to wchar_t.</td>
</tr>
<tr>
<td>ls</td>
<td>Multibyte string constant</td>
<td>Pointer to wchar_t string.</td>
</tr>
</tbody>
</table>

To read strings not delimited by space characters, substitute a set of characters in brackets ([ ]) for the `s` (string) type character. 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 caret (^), 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 %ac, where \(a\) is a decimal integer. In this instance, the c type character 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 format specifier is interpreted as a hexadecimal number.

The scanf() function scans 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 was one, is considered unread and is the first character of the next input field or the first character in subsequent read operations on stdin.

For %lc and %ls, specifies the data that is read is a multibyte string and is converted to wide characters as if by calls to mbtowc.

For the %a, %A, %e, %f, %F, %g, and %G format specifiers, a character sequence of INFINITY or NAN (ignoring case) is allowed and yields a value of INFINITY or Quiet Not-A-Number (NaN), respectively.

Alternative format specification has the following form:

As an alternative, specific entries in the argument-list may be assigned by using the format specification outlined in the diagram above. This format specification and the previous format specification may not be mixed in the same call to scanf(). Otherwise, unpredictable results may occur.

The arg-number is a positive integer constant where 1 refers to the first entry in the argument-list. Arg-number may not be greater than the number of entries in the argument-list, or else the results are undefined. Arg-number also may not be greater than NL_ARGMAX.

**Return Value**

The scanf() function returns the number of fields that were successfully converted and assigned. The return value does not include fields that were read but not assigned.

The return value is EOF for an attempt to read at end-of-file if no conversion was performed. A return value of 0 means that no fields were assigned.

**Error Conditions**

If the type of the argument that is to be assigned into is different than the format specification, unpredictable results can occur. For example, reading a floating-point value, but assigning it into a variable of type int, is incorrect and would have unpredictable results.

If there are more arguments than format specifications, the extra arguments are ignored. The results are undefined if there are not enough arguments for the format specifications.

If the format string contains an invalid format specification, and positional format specifications are being used, errno will be set to EILSEQ.

If positional format specifications are used and there are not enough arguments, errno will be set to EINVAL.
If a conversion error occurs, **errno** may be set to **ECONVERT**.

**Examples**

This example scans various types of data.

```c
#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 fields were assigned\n");
    else
    {
        printf("integer = \n", i);
        printf("real number = \n", fp);
        printf("character = \n", c);
        printf("string = \n", s);
    }
}

/************************ If input is: 12 2.5 a yes,  ******
**************     then output should be similar to:   ***********
Enter an integer, a real number, a character and a string :
integer = 12
real number = 2.500000
character = a
string = yes
*/
```

This example converts a hexadecimal integer to a decimal integer. The while loop ends if the input value is not a hexadecimal integer.

```c
#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 = \n", number);
        printf("Decimal Number     = \n", number);
    }
}

/************************ If input is: 0x231 0xf5e 0x1 q,  ******
*****************     then output should be similar to:       ***********
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
*/
```

This example reads from stdin and assigns data by using the alternative positional format string.
#include <stdio.h>
int main(int argc, char *argv[]) {
    int i;
    char s[20];
    float f;
    scanf("%2$s %3$f %1$d", &i, s, &f);
    printf("The data read was \n%i\n%s\n%f\n,i,s,f);
    return 0;
}

/*********  If the input is : test 0.2 100 ***********/
/********* then the output will be similar to: ***********/

The data read was
100
test
0.20000
*/

This example reads in a multibyte character string into a wide Unicode string. The example can be compiled with either LOCALETYPE(*LOCALEUCS2) or LOCALETYPE(*LOCALEUTF).

#include <locale.h>
#include <stdio.h>
#include <wchar.h>

void main(void) {
    wchar_t uString[20];
    setlocale(LC_UNI_ALL, "");
    scanf("Enter a string %ls", uString);
    printf("String read was %ls\n", uString);
}

/* if the input is : ABC
then the output will be similar to:
String read was ABC
*/

Related Information
- “fscanf() — Read Formatted Data” on page 155
- “printf() — Print Formatted Characters” on page 251
- “sscanf() — Read Data” on page 382
- “strtod() - strtof() - strtold() — Convert Character String to Double, Float, and Long Double” on page 422
- “strtod32() - strtod64() - strtod128() — Convert Character String to Decimal Floating-Point” on page 425
- “wscanf() — Read Data Using Wide-Character Format String” on page 539
- “fwscanf() — Read Data from Stream Using Wide Character” on page 169
- “swscanf() — Read Wide Character Data” on page 437
- “<stdio.h>” on page 13
setbuf() — Control Buffering

Format

```c
#include <stdio.h>
void setbuf(FILE *, char *buffer);
```

Language Level

ANSI

Threading Safe

Yes

Description

If the operating system supports user-defined buffers, setbuf() controls buffering for the specified stream. The setbuf() function only works in ILE C when using the integrated file system. The stream pointer must refer to an open file before any I/O or repositioning has been done.

If the buffer argument is NULL, the stream is unbuffered. If not, the buffer must point to a character array of length BUFSIZ, which is the buffer size that is defined in the <stdio.h> include file. The system uses the buffer, which you specify, for input/output buffering instead of the default system-allocated buffer for the given stream. stdout, stderr, and stdin do not support user-defined buffers.

The setvbuf() function is more flexible than the setbuf() function.

Return Value

There is no return value.

Example

This example opens the file setbuf.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.

```c
#include <stdio.h>

int main(void)
{
    char buf[BUFSIZ];
    char string[] = "hello world";
    FILE *stream;
    memset(buf, '\0', BUFSIZ); /* initialize buf to null characters */
    stream = fopen("setbuf.dat", "wb");
    setbuf(stream, buf); /* set up buffer */
    fwrite(string, sizeof(string), 1, stream);
    printf("%s
", buf); /* string is found in buf now */
    fclose(stream); /* buffer is flushed out to myfile.dat */
}
```

Related Information

- “fclose() — Close Stream” on page 116
- “fflush() — Write Buffer to File” on page 121
setjmp() — Preserve Environment

Format

```c
#include <setjmp.h>
int setjmp(jmp_buf env);
```

Language Level

ANSI

Threadsafe

Yes

Description

The `setjmp()` function saves a stack environment that can subsequently be restored by the `longjmp()` function. The `setjmp()` and `longjmp()` functions provide a way to perform a non-local goto. They are often used in signal handlers.

A call to the `setjmp()` function causes it to save the current stack environment in `env`. A subsequent call to the `longjmp()` function restores the saved environment and returns control to a point corresponding to the `setjmp()` call. The values of all variables (except register variables) available to the function receiving control contain the values they had when the `longjmp()` function was called. The values of register variables are unpredictable. Nonvolatile auto variables that are changed between calls to the `setjmp()` function and the `longjmp()` function are also unpredictable.

Return Value

The `setjmp()` function returns the value 0 after saving the stack environment. If the `setjmp()` function returns as a result of a `longjmp()` call, it returns the `value` argument of the `longjmp()` function, or 1 if the `value` argument of the `longjmp()` function is 0. There is no error return value.

Example

This example saves the stack environment at the statement:

```c
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 the `setjmp()` function returns a 0 when it saves the environment. The program prints the message:

```c
setjmp has been called
```

The subsequent call to function `p()` causes it to call the `longjmp()` function. Control is transferred to the point in the `main()` function immediately after the call to the `setjmp()` function using the environment saved in the `mark` variable. This time, the condition is TRUE because -1 is specified in the second parameter on the `longjmp()` function call as the return value to be placed on the stack. The example then performs the statements in the block, prints the message "`longjmp()` has been called", calls the `recover()` function, and leaves the program.
```c
#include <stdio.h>
#include <setjmp.h>
#include <stdlib.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");
    printf("Calling function p()\n");
    p();
    printf("This point should never be reached\n");
}

void p(void)
{
    printf("Calling longjmp() from inside function p()\n");
    longjmp(mark, -1);
    printf("This point should never be reached\n");
}

void recover(void)
{
    printf("Performing function recover()\n");
}

/****************************Output should be as follows: ****************************
  setjmp has been called
  Calling function p()
  longjmp has been called
  Calling longjmp() from inside function p()
  Recover has been called
  Performing function recover()

*********************************************************************/

Related Information

• “longjmp() — Restore Stack Environment” on page 215
• “<setjmp.h>” on page 11

setlocale() — Set Locale

Format

@include <locale.h>

char *setlocale(int category, const char *locale);

Language Level

ANSI

Threadsafe

No

Locale Sensitive

For more information, see “Understanding CCSIDs and Locales” on page 565.
Description

The `setlocale()` function changes or queries variables that are defined in the `<locale.h>` include file, that indicate location. The values for category are listed below.

<table>
<thead>
<tr>
<th>Category</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>LC_ALL</td>
<td>Names entire locale of program.</td>
</tr>
<tr>
<td>LC_COLLATE</td>
<td>Affects behavior of the <code>strcmp()</code> and <code>strxfrm()</code> functions.</td>
</tr>
<tr>
<td>LC_CTYPE</td>
<td>Affects behavior of character handling functions.</td>
</tr>
<tr>
<td>LC_MONETARY</td>
<td>Affects monetary information returned by <code>localeconv()</code> and <code>nl_langinfo()</code> functions.</td>
</tr>
<tr>
<td>LC_NUMERIC</td>
<td>Affects the decimal-point character for the formatted input/output and string conversion functions, and the non-monetary formatting information returned by <code>localeconv()</code> and <code>nl_langinfo()</code> functions.</td>
</tr>
<tr>
<td>LC_TIME</td>
<td>Affects behavior of the <code>strftime()</code> function and the time formatting information returned by the <code>nl_langinfo()</code> function.</td>
</tr>
<tr>
<td>LC_TOD</td>
<td>Affects the behavior of the time functions.</td>
</tr>
<tr>
<td></td>
<td>The category LC_TOD has several fields in it. The TNAME field is the time zone name. The TZDIFF field is the difference between local time and Greenwich Meridian time. If the TNAME field is nonblank, then the TZDIFF field is used when determining the values that are returned by some of the time functions. This value takes precedence over the system value, QUTCOFFSET.</td>
</tr>
<tr>
<td>LC_UNI_ALL*</td>
<td>This category causes <code>setlocale()</code> to load all of the the LC_UNI_ categories from the locale specified. This category accepts only a locale with a UCS-2 or UTF-32 CCSID.</td>
</tr>
<tr>
<td>LC_UNI_COLLATE*</td>
<td>Affects behavior of the <code>wcsncmp()</code> and <code>wcsxfrm()</code> functions. This category accepts only a locale with a UCS-2 or UTF-32 CCSID.</td>
</tr>
<tr>
<td></td>
<td><strong>Note:</strong> This category is not supported for UCS-2.</td>
</tr>
<tr>
<td>LC_UNI_CTYPE*</td>
<td>Affects the behavior of the wide character handling functions. This category accepts only a locale with a UCS-2 or UTF-32 CCSID.</td>
</tr>
<tr>
<td>LC_UNI_MESSAGES*</td>
<td>Affects the message formatting information returned by the _WCS_n1_langinfo() function. This category accepts only a locale with a UCS-2 or UTF-32 CCSID.</td>
</tr>
<tr>
<td>LC_UNI_MONETARY*</td>
<td>Affects the monetary information returned by the <code>wcslocaleconv()</code> and _WCS_n1_langinfo() functions. This category accepts only a locale with a UCS-2 or UTF-32 CCSID.</td>
</tr>
<tr>
<td>LC_UNI_NUMERIC*</td>
<td>Affects the decimal-point character for the wide character formatted input/output and wide character string conversion functions, and the non-monetary information returned by the <code>wcslocaleconv()</code> and _WCS_n1_langinfo() functions. This category accepts only a locale with a UCS-2 or UTF-32 CCSID.</td>
</tr>
<tr>
<td>LC_UNI_TIME*</td>
<td>Affects the behavior of the <code>wcsftime()</code> function and the time formatting information returned by the _WCS_n1_langinfo() functions. This category accepts only a locale with a UCS-2 or UTF-32 CCSID.</td>
</tr>
<tr>
<td>LC_UNI_TOD*</td>
<td>Affects the behavior of the wide character time functions. This category accepts only a locale with a UCS-2 or UTF-32 CCSID.</td>
</tr>
</tbody>
</table>
To use categories with UNI in the name, LOCALETYPE(*LOCALEUCS2) or LOCALETYPE(*LOCALEUTF) must be specified on the compilation command. If LOCALETYPE(*LOCALEUCS2) is used, the locale specified must be a UCS-2 locale. If LOCALETYPE(*LOCALEUTF) is used, the locale specified must be a UTF-32 locale.

**Note:** There are two ways of defining setlocale() and other locale-sensitive C functions. The original way to define setlocale() uses *CLD locale objects to set the locale and retrieve locale-sensitive data. The second way to define setlocale() uses *LOCALE objects to set the locale and retrieve locale-sensitive data. The original way is accessed by specifying LOCALETYPE(*CLD) on the compilation command. The second way is accessed by specifying LOCALETYPE(*LOCALE), LOCALETYPE(*LOCALEUCS2), or LOCALETYPE(*LOCALEUTF) on the compilation command. For more information about the two methods of locale definition in ILE C, see the International Locale Support section in the ILE C/C++ Programmer's Guide.

### Setlocale using *CLD locale objects

You can set the value of locale to "C", ",", LC_C, LC_C_GERMANY, LC_C_FRANCE, LC_C_SPAIN, LC_C_ITALY, LC_C_USA or LC_C_UK. A locale value of "C" indicates the default C environment. A locale value of "," tells the setlocale() function to use the default locale for the implementation.

### Setlocale with *LOCALE objects

You can set the value of locale to ",", "C", "POSIX", or the fully qualified Integrated File System path name of a *LOCALE object enclosed in double quotes. A locale value of "C" or "POSIX" indicates the default C *LOCALE object. A locale value of "," tells the setlocale() function to use the default locale for the process.

The default locale for the process is determined using the following table:

<table>
<thead>
<tr>
<th>Category</th>
<th>Default Locale</th>
</tr>
</thead>
</table>
| LC_ALL    | 1. Check the LC_ALL environment variable. If it is defined and not null, use the specified locale for all POSIX locale categories. Otherwise, go to the next step.  
2. For each POSIX locale category (LC_CTYPE, LC_COLLATE, LC_TIME, LC_NUMERIC, LC_MESSAGES, LC_MONETARY, and LC_TOD), check the environment variable with the same name. If it is defined and not null, use the locale specified. Otherwise, go to the next step.  
3. Check the LANG environment variable. For every locale category that was not set in the previous step, if the LANG environment variable is defined and not null, set the locale category to the specified locale. Otherwise, set it to the default C *LOCALE object. |

---

**IBM i: ILE C/C++ Runtime Library Functions**

368
<table>
<thead>
<tr>
<th>Category</th>
<th>Default Locale</th>
</tr>
</thead>
<tbody>
<tr>
<td>LC_CTYPE</td>
<td>1. Check the LC_ALL environment variable&lt;sup&gt;1&lt;/sup&gt;. If it is defined and not null, use the specified locale&lt;sup&gt;2&lt;/sup&gt;. Otherwise, go to the next step.</td>
</tr>
<tr>
<td>LC_COLLATE</td>
<td>2. Check the environment variable with the same name&lt;sup&gt;1&lt;/sup&gt; as the specified locale category. If it is defined and not null, use the locale specified&lt;sup&gt;2&lt;/sup&gt;. Otherwise, go to the next step.</td>
</tr>
<tr>
<td>LC_TIME</td>
<td>3. Check the LANG environment variable&lt;sup&gt;1&lt;/sup&gt;. If it is defined and not null, set the locale category to the specified locale&lt;sup&gt;2&lt;/sup&gt;. Otherwise, go to the next step.</td>
</tr>
<tr>
<td>LC_NUMERIC</td>
<td>4. Set the locale category to the default C *LOCALE object.</td>
</tr>
<tr>
<td>LC_MESSAGES</td>
<td></td>
</tr>
<tr>
<td>LC_MONETARY</td>
<td></td>
</tr>
<tr>
<td>LC_TOD</td>
<td></td>
</tr>
<tr>
<td>LC_UNI_ALL</td>
<td>If your module is compiled with the LOCALETYPE(*LOCALEUCS2) option:</td>
</tr>
<tr>
<td></td>
<td>1. Check the LC_UCS2_ALL environment variable&lt;sup&gt;1&lt;/sup&gt;. If it is defined and not null, use the specified locale for all Unicode locale categories. Otherwise, go to the next step.</td>
</tr>
<tr>
<td></td>
<td>2. For each Unicode locale category check the corresponding environment variable&lt;sup&gt;1&lt;/sup&gt; (LC_UCS2_CTYPE, LC_UCS2_COLLATE, LC_UCS2_TIME, LC_UCS2_NUMERIC, LC_UCS2_MESSAGES, LC_UCS2_MONETARY, or LC_UCS2_TOD)&lt;sup&gt;3&lt;/sup&gt;. If it is defined and not null, use the locale specified. Otherwise, go to the next step.</td>
</tr>
<tr>
<td></td>
<td>3. Set the locale category to the default UCS-2 *LOCALE object.</td>
</tr>
<tr>
<td></td>
<td>If your module is compiled with the LOCALETYPE(*LOCALEUTF) option:</td>
</tr>
<tr>
<td></td>
<td>1. Check the LC_UTF_ALL environment variable&lt;sup&gt;1&lt;/sup&gt;. If it is defined and not null, use the specified locale for all Unicode locale categories. Otherwise, go to the next step.</td>
</tr>
<tr>
<td></td>
<td>2. For each Unicode locale category check the corresponding environment variable&lt;sup&gt;1&lt;/sup&gt; (LC_UTF_CTYPE, LC_UTF_COLLATE, LC_UTF_TIME, LC_UTF_NUMERIC, LC_UTF_MESSAGES, LC_UTF_MONETARY, or LC_UTF_TOD)&lt;sup&gt;3&lt;/sup&gt;. If it is defined and not null, use the locale specified. Otherwise, go to the next step.</td>
</tr>
<tr>
<td></td>
<td>3. Check the LANG environment variable&lt;sup&gt;1&lt;/sup&gt;. For every locale category that was not set in the previous step, if the LANG environment variable is defined and not null, set the locale category to the specified locale. Otherwise, set it to the default UTF *LOCALE object.</td>
</tr>
<tr>
<td>Category</td>
<td>Default Locale</td>
</tr>
<tr>
<td>------------------------</td>
<td>-------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>LC_UNI_CTYPE</td>
<td>If your module is compiled with the LOCALETYPE(*LOCALEUCS2) option:</td>
</tr>
<tr>
<td>LC_UNI_COLLATE</td>
<td>1. Check the environment variable corresponding to the specified locale category(^1) (LC_UCS2_CTYPE, LC_UCS2_COLLATE, LC_UCS2_TIME, LC_UCS2_NUMERIC, LC_UCS2_MESSAGES, LC_UCS2_MONETARY, or LC_UCS2_TOD)(^3). If it is defined and not null, use the locale specified. Otherwise, go to the next step.</td>
</tr>
<tr>
<td>LC_UNI_TIME</td>
<td>2. Check the LC_UCS2_ALL environment variable(^3). If it is defined and not null, use the specified locale. Otherwise, go to the next step.</td>
</tr>
<tr>
<td>LC_UNI_NUMERIC</td>
<td>3. Set the locale category to the default UCS-2 *LOCALE object.</td>
</tr>
<tr>
<td>LC_UNI_MESSAGES</td>
<td>If your module is compiled with the LOCALETYPE(*LOCALEUTF) option:</td>
</tr>
<tr>
<td>LC_UNI_MONETARY</td>
<td>1. Check the environment variable corresponding to the specified locale category(^1) (LC_UTF_CTYPE, LC_UTF_COLLATE, LC_UTF_TIME, LC_UTF_NUMERIC, LC_UTF_MESSAGES, LC_UTF_MONETARY, or LC_UTF_TOD)(^3). If it is defined and not null, use the locale specified. Otherwise, go to the next step.</td>
</tr>
<tr>
<td>LC_UNI_TOD</td>
<td>2. Check the LC_UTF_ALL environment variable(^3). If it is defined and not null, use the specified locale. Otherwise, go to the next step.</td>
</tr>
<tr>
<td></td>
<td>3. Check the LANG environment variable(^1). If the LANG environment variable is defined and not null, set the locale category to the specified locale. Otherwise, set it to the default UTF *LOCALE object.</td>
</tr>
</tbody>
</table>

**Note:** 1 The environment variables with names corresponding to locale categories are created by the user. The LANG environment variable is automatically created during job initiation when you specify a locale path name for either of the following:

- the LOCALE parameter in your user profile (see the CHGUSRPRF (Change User Profile) command information in the Information Center).
- the QLOCALE system value (see the QLOCALE system value information in the Information Center).

The locale environment variables are expected to contain a locale path name of the form /QSYS.LIB/<locname>.LOCALE or /QSYS.LIB/<libname>.LIB/<locname>.LOCALE. If your module is compiled with the LOCALETYPE(*LOCALEUTF) option, the environment variable will be ignored if the <locname> portion of the path exceeds 8 characters. This restriction exists because a 2 character suffix must be appended to the locale name to get the name of the corresponding UTF locale.

**Note:** 2 When LOCALETYPE(*LOCALEUTF) is specified on the compilation command, the setlocale() function appends a trailing _8 to the LC_ALL, LC_CTYPE, LC_COLLATE, LC_TIME, LC_NUMERIC, LC_MESSAGES, LC_MONETARY, LC_TOD, and LANG environment variables. If this locale is not found, the UTF default locale object is used. For example, setlocale(LC_ALL, "") when LANG is set to /QSYS.LIB/EN_US.LOCALE causes setlocale() to attempt to load the locale /QSYS.LIB/EN_US_8.LOCALE. If the LANG environment variable is used to set one of the Unicode locale categories (LC_UNI_ALL, LC_UNI_CTYPE, LC_UNI_COLLATE, LC_UNI_TIME, LC_UNI_NUMERIC, LC_UNI_MESSAGES, LC_UNI_MONETARY, or LC_UNI_TOD), setlocale() appends a trailing _4 to the locale name stored in the environment variable. This is an attempt to locate the corresponding UTF-32 locale. If this locale is not found, the default UTF-32 locale object is used. For example, setlocale(LC_UNI_TIME, "") when LANG is set to /QSYS.LIB/EN_US.LOCALE causes setlocale() to attempt to load the locale /QSYS.LIB/EN_US_4.LOCALE. Locale names ending in _4
and _8 follow a naming convention introduced by the CRTLOCALE CL command (see the CRTLOCALE
(Create Locale) command information in the Information Center) for locales created with CCSID(*UTF).

Note: 3 The LC_UNI_ALL, LC_UNI_COLLATE, LC_UNI_CTYPE, LC_UNI_TIME, LC_UNI_NUMERIC,
LC_UNI_MESSAGES, LC_UNI_MONETARY, and LC_UNI_TOD locale category names are shared between
UCS-2 and UTF. The environment variables corresponding to these categories cannot be shared, so the
names of the environment variables do not exactly match the locale category names. For UCS-2
environment variable names, UNI is replaced with UCS2 (for example, LC_UNI_ALL locale category
becomes LC_UCS2_ALL environment variable). For UTF environment variable names, UNI is replaced with
UTF (for example, LC_UNI_ALL locale category becomes LC_UTF_ALL environment variable).

If compiled with LOCALETYPE(*LOCALEUCS2) or LOCALETYPE(*LOCALEUTF), the locale must be a
pointer to a valid Unicode locale for the categories starting with LC_UNI_, and must not be a Unicode
locale for the other categories.

Return Value
The setlocale() function returns a pointer to a string that represents the current locale setting. If the
returned string is stored, the stored string value can be used as input to the setlocale() function to
restore the locale setting at any time. However, you need to copy the string to a user-defined buffer;
otherwise, the string is overwritten on subsequent calls to setlocale().

Note: Because the string to which a successful call to setlocale() points may be overwritten by
subsequent calls to the setlocale() function, you should copy the string if you plan to use it later. The
exact format of the locale string is different between locale types of *CLD, *LOCALE, *LOCALEUCS2, and
*LOCALEUTF.

To query the locale, give a NULL as the second parameter. For example, to query all the categories of your
locale, enter the following statement:

```
char *string = setlocale(LC_ALL, NULL);
```

Error Conditions
On error, the setlocale() function returns NULL, and the program's locale is not changed.

Example that uses *CLD locale objects

```
/************************************************************************
This example sets the locale of the program to
LC_C_FRANCE *CLD and prints the string
that is associated with the locale. This example must be compiled with
the LOCALETYPE(*CLD) parameter on the compilation command.
************************************************************************/
#include <stdio.h>
#include <locale.h>

char *string;

int main(void)
{
    string = setlocale(LC_ALL, LC_C_FRANCE);
    if (string != NULL)
        printf(" %s \
",string);
}
```

Example that uses *LOCALE objects

```c
#include <stdio.h>
#include <locale.h>

char *string;

int main(void)
{
    string = setlocale(LC_ALL, "POSIX");
    if (string != NULL)
        printf(" %s 
",string);
}
```

Related Information

- “getenv() — Search for Environment Variables” on page 176
- “localeconv() — Retrieve Information from the Environment” on page 202
- “nl_langinfo() — Retrieve Locale Information” on page 246
- “<locale.h>" on page 5

setvbuf() — Control Buffering

Format

```c
#include <stdio.h>
int setvbuf(FILE *stream, char *buf, int type, size_t size);
```

Language Level

ANSI

Threadsafe

Yes

Description

The `setvbuf()` function allows control over the buffering strategy and buffer size for a specified stream. The `setvbuf()` function only works in ILE C when using the integrated file system. The stream must refer to a file that has been opened, but not read or written to.

The array pointed to by `buf` designates an area that you provide that the C 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 C 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.

The `type` must be one of the following:

<table>
<thead>
<tr>
<th>Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>_IONBF</td>
<td>No buffer is used.</td>
</tr>
</tbody>
</table>
_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. The buffer is deleted when a new-line character is written, when the buffer is full, or when input is requested.

If type is _IOFBF or _IOLBF, size is the size of the supplied buffer. If buf is NULL, the C library takes size as the suggested size for its own buffer. If type is _IONBF, both buf and size are ignored.

The value for size must be greater than 0.

Return Value
The setvbuf() function returns 0 if successful. It returns nonzero if a value that is not valid was specified in the parameter list, or if the request cannot be performed.

The setvbuf() function has no effect on stdout, stdin, or stderr.

Warning: The array that is used as the buffer must still exist when the specified stream is closed. For example, if the buffer is declared within the scope of a function block, the stream must be closed before the function is ended and frees the storage allocated to the buffer.

Example
This example sets up a buffer of buf for stream1 and specifies that input to stream2 is to be unbuffered.

```c
#include <stdio.h>

#define BUF_SIZE 1024
char buf[BUF_SIZE];
FILE *stream1, *stream2;

int main(void)
{
   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\n");

   /* stream2 is unbuffered */
   if (setvbuf(stream2, NULL, _IONBF, 0) != 0)
      printf("Incorrect type or size of buffer\n");

   /* This is a program fragment and not a complete function example */
}
```

Related Information
• “fclose() — Close Stream” on page 116
• “fflush() — Write Buffer to File” on page 121
• “fopen() — Open Files” on page 134
• “setbuf() — Control Buffering” on page 364
• “<stdio.h>” on page 13
signal() — Handle Interrupt Signals

Format

```c
#include <signal.h>
void (*signal (int sig, void(*)(int)))(int);
```

Language Level

ANSI

Threadsafe

Yes

Description

The `signal()` function allows a program to choose one of several ways to handle an interrupt signal from the operating system or from the `raise()` function. If compiled with the SYSIFCOPT(*ASYNCSIGNAL) option, this function uses asynchronous signals. The asynchronous version of this function behaves like `sigaction()` with SA_NODEFER and SA_RESETHAND options. Asynchronous signal handlers may not call `abort()` or `exit()`. The remainder of this function description will describe synchronous signals.

The `sig` argument must be one of the macros SIGABRT, SIGALL, SIGILL, SIGINT, SIGFPE, SIGIO, SIGOTHER, SIGSEGV, SIGTERM, SIGUSR1, or SIGUSR2, defined in the `signal.h` include file. SIGALL, SIGIO, and SIGOTHER are only supported by the ILE C/C++ runtime library. The `func` argument must be one of the macros SIG_DFL or SIG_IGN, defined in the `<signal.h>` include file, or a function address.

The meaning of the values of `sig` is as follows:

<table>
<thead>
<tr>
<th>Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>SIGABRT</td>
<td>Abnormal termination</td>
</tr>
<tr>
<td>SIGALL</td>
<td>Catch-all for signals whose current handling action is SIG_DFL. When SYSIFCOPT(*ASYNCSIGNAL) is specified, SIGALL is not a catch-all signal. A signal handler for SIGALL is only invoked for a user-raised SIGALL signal.</td>
</tr>
<tr>
<td>SIGINT</td>
<td>Detection of a function image that was not valid</td>
</tr>
<tr>
<td>SIGFPE</td>
<td>Arithmetic exceptions that are not masked, such as overflow, division by zero, and operations that are not valid</td>
</tr>
<tr>
<td>SIGIO</td>
<td>Record file I/O error</td>
</tr>
<tr>
<td>SIGOTHER</td>
<td>ILE C signal</td>
</tr>
<tr>
<td>SIGSEGV</td>
<td>Access to memory that was not valid</td>
</tr>
<tr>
<td>SIGTERM</td>
<td>End request sent to the program</td>
</tr>
</tbody>
</table>
SIGUSR1
Intended for use by user applications. (extension to ANSI)

SIGUSR2
Intended for use by user applications. (extension to ANSI)

The action that is 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.

**Return Value**

A return value of SIG_ERR indicates an error in the call to `signal()`. If successful, the call to `signal()` returns the most recent value of `func`. The value of errno may be set to EINVAL (the signal is not valid).

**Example**

This example shows you how to establish a signal handler.

```c
#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(SIG, StrCln)       {                    
    if (signal((SIG), (StrCln)) == SIG_ERR) {            
        perror("Could not signal user signal");            
        abort();                                           
    }                                                    
}                                                    

void StrCln(int);
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;
    *buffer = malloc(size*ONE_K);    /* get the size in number of K */
    if (*buffer == NULL) {
        if (raise(OUT_OF_STORAGE)) {
            perror("Could not raise user signal");
            abort();
        }
    }
    return;
}
/* This is a program fragment and not a complete function example */
```
**sin() — Calculate Sine**

**Format**

```c
#include <math.h>
double sin(double x);
```

**Language Level**

ANSI

**Threading**

Yes

**Description**

The `sin()` function calculates the sine of `x`, with `x` expressed in radians. If `x` is too large, a partial loss of significance in the result may occur.

**Return Value**

The `sin()` function returns the value of the sine of `x`. The value of `errno` may be set to either EDOM or ERANGE.

**Example**

This example computes `y` as the sine of π/2.

```c
#include <math.h>
#include <stdio.h>

int main(void)
{
    double pi, x, y;
    pi = 3.1415926535;
    x = pi/2;
    y = sin(x);
    printf("sin( %.1f ) = %.1f\n", x, y);
}
/*
 **sin( 1.570796 ) = 1.000000
 */
```

**Related Information**

- “acos() — Calculate Arccosine” on page 62
- “asin() — Calculate Arcsine” on page 66
sinh() — Calculate Hyperbolic Sine

Format

```c
#include <math.h>
double sinh(double x);
```

Language Level

ANSI

Threatsafe

Yes

Description

The `sinh()` function calculates the hyperbolic sine of `x`, with `x` expressed in radians.

Return Value

The `sinh()` function returns the value of the hyperbolic sine of `x`. If the result is too large, the `sinh()` function sets `errno` to ERANGE and returns the value `HUGE_VAL` (positive or negative, depending on the value of `x`).

Example

This example computes `y` as the hyperbolic sine of \( \frac{\pi}{2} \).

```c
#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);
}
```

Related Information

- “acos() — Calculate Arccosine” on page 62
- “asin() — Calculate Arcsine” on page 66
snprintf() — Print Formatted Data to Buffer

Format

```c
#include <stdio.h>
int snprintf(char *buffer, size_t n, const char *format-string, argument-list);
```

Language Level

ANSI

Threadsafe

Yes

Locale Sensitive

The behavior of this function might be affected by the LC_CTYPE and LC_NUMERIC categories of the current locale. The behavior might also be affected by the LC_CTYPE category of the current locale if LOCALETYPE(*LC_ALL) or LOCALETYPE(*LOCALEUTF) is specified on the compilation command. For more information, see “Understanding CCSIDs and Locales” on page 565.

Description

The snprintf() function formats and stores a series of characters and values in the array buffer. Any argument-list is converted and put out according to the corresponding format specification in the format-string. The snprintf() function is identical to the sprintf() function with the addition of the n argument, which indicates the maximum number of characters (including the ending null character) to be written to buffer.

The format-string consists of ordinary characters and has the same form and function as the format string for the printf() function.

Return Value

The snprintf() function returns the number of bytes that are written in the array, not counting the ending null character.

Example

This example uses snprintf() to format and print various data.

```c
#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 = snprintf(buffer, 6, "%s\n", s);
    j += snprintf(buffer+j, 6, "%c\n", c);
    j += snprintf(buffer+j, 6, "%d\n", i);
    j += snprintf(buffer+j, 6, "%f\n", fp);
    printf("string:\n%s\ncharacter count = %d\n", buffer, j);
}

/**************************** Output should be similar to:  ****************************
string:
baltil
35
1.732
character count = 15
 */

Related Information

• “fprintf() — Write Formatted Data to a Stream” on page 141
• “printf() — Print Formatted Characters” on page 251
• “sprintf() — Print Formatted Data to Buffer” on page 379
• “vsnprintf() — Print Argument Data to Buffer” on page 467
• “<stdio.h>” on page 13

sprintf() — Print Formatted Data to Buffer

Format

#include <stdio.h>
int sprintf(char *buffer, const char *format-string, argument-list);

Language Level

ANSI

Threading Safe

Yes

Locale Sensitive

The behavior of this function might be affected by the LC_CTYPE and LC_NUMERIC categories of the current locale. The behavior might also be affected by the LC_UNI_CTYPE category of the current locale if LOCALETYPE(*LOCALEUCS2) or LOCALETYPE(*LOCALEUTF) is specified on the compilation command. For more information, see “Understanding CCSIDs and Locales” on page 565.

Description

The sprintf() function formats and stores a series of characters and values in the array buffer. Any argument-list is converted and put out according to the corresponding format specification in the format-string.

The format-string consists of ordinary characters and has the same form and function as the format-string argument for the printf() function.
**Return Value**
The `sprintf()` function returns the number of bytes that are written in the array, not counting the ending null character.

**Example**
This example uses `sprintf()` to format and print various data.

```c
#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 should be similar to: */
string:
baltimore
l
35
1.732051
character count = 24
```

**Related Information**
- “fprintf() — Write Formatted Data to a Stream” on page 141
- “printf() — Print Formatted Characters” on page 251
- “sscanf() — Read Data” on page 382
- “swprintf() — Format and Write Wide Characters to Buffer” on page 435
- “vfprintf() — Print Argument Data to Stream” on page 457
- “vprintf() — Print Argument Data” on page 464
- “vsprintf() — Print Argument Data to Buffer” on page 468
- “<stdio.h>” on page 13

**sqrt() — Calculate Square Root**

**Format**
```c
#include <math.h>
double sqrt(double x);
```

**Language Level**
ANSI
Threadsafe
Yes

Description
The sqrt() function calculates the nonnegative value of the square root of x.

Return Value
The sqrt() function returns the square root result. If x is negative, the function sets errno to EDOM, and returns 0.

Example
This example computes the square root of the quantity that is passed as the first argument to main. It prints an error message if you pass a negative value.

```c
#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( %lf ) = %lf\n", value, sqrt( value ));
    }
}

/****************************  If the input is 45,  ****************************
********  then the output should be similar to:  **********
sqrt( 45.000000 ) = 6.708204
*/

Related Information
• “exp() — Calculate Exponential Function” on page 114
• “hypot() — Calculate Hypotenuse” on page 191
• “log() — Calculate Natural Logarithm” on page 212
• “log10() — Calculate Base 10 Logarithm” on page 213
• “pow() — Compute Power” on page 250
• “<math.h>” on page 6

srand() — Set Seed for rand() Function

Format
```c
#include <stdlib.h>
void srand(unsigned int seed);
```
The `srand()` function sets the starting point for producing a series of pseudo-random integers. If `srand()` is not called, the `rand()` seed is set as if `srand(1)` were called at program start. Any other value for `seed` sets the generator to a different starting point.

The `rand()` function generates the pseudo-random numbers.

**Return Value**

There is no return value.

**Example**

This example first calls `srand()` with a value other than 1 to initiate the random value sequence. Then the program computes five random values for the array of integers that are called `ranvals`.

```c
#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 should be similar to:  *****************
    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**

- “`rand()` – `rand_r()` — Generate Random Number” on page 284
- “`<stdlib.h>`” on page 15

---

**sscanf() — Read Data**

**Format**

```c
#include <stdio.h>
int sscanf(const char *buffer, const char *format, argument-list);
```

**Language Level**

ANSI
Threadsafe
Yes

Locale Sensitive
The behavior of this function might be affected by the LC_CTYPE and LC_NUMERIC categories of the current locale. The behavior might also be affected by the LC_CTYPE category of the current locale if LC_CHARS=(*LOCALEUCS2) or LC_PCASE=(*LOCALEUTF) is specified on the compilation command. For more information, see “Understanding CCSIDs and Locales” on page 565.

Description
The sscanf() function reads data from buffer into the locations that are given by argument-list. Each argument must be a pointer to a variable with a type that corresponds to a type specifier in the format-string.

Return Value
The sscanf() function returns the number of fields that were successfully converted and assigned. The return value does not include fields that were read but not assigned.

The return value is EOF when the end of the string is encountered before anything is converted.

Example
This example uses sscanf() to read various data from the string tokenstring, and then displays that data.

```c
#include <stdio.h>
#include <stddef.h>

int main(void)
{
    char *tokenstring = "15 12 14";
    char *string = "ABC Z";
    wchar_t ws[81];
    wchar_t wc;
    int i;
    float fp;
    char s[81];
    char c;
    /* Input various data                                         */
    /* In the first invocation of sscanf, the format string is     */
    /* "%s %c%d%f". If there were no space between %s and %c,    */
    /* sscanf would read the first character following the        */
    /* string, which is a blank space.                           */
    sscanf(tokenstring, "%s %c%d%f", s, &c, &i, &fp);
    sscanf(string, "%ls %lc", ws,&wc);
    /* 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);
    printf("wide-character string = %S\n",ws);
    printf("wide-character = %C\n",wc);
}

/******************** Output should be similar to: ********************/
string = 15
character = 1
integer = 2
floating-point number = 14.000000
wide-character string = ABC
wide-character = Z
*****************************************************************************/
Related Information

- “fscanf() — Read Formatted Data” on page 155
- “scanf() — Read Data” on page 358
- “swscanf() — Read Wide Character Data” on page 437
- “fwscanf() — Read Data from Stream Using Wide Character” on page 169
- “wscanf() — Read Data Using Wide-Character Format String” on page 539
- “sprintf() — Print Formatted Data to Buffer” on page 379
- “<stdio.h>” on page 13

strcasecmp() — Compare Strings without Case Sensitivity

Format

```c
#include <strings.h>
int strcasecmp(const char *string1, const char *string2);
```

Language Level

XPG4

Threadsafe

Yes

Locale Sensitive

The behavior of this function might be affected by the LC_CTYPE category of the current locale. This function is not available when LOCALETYPE("CLD) is specified on the compilation command. For more information, see “Understanding CCSIDs and Locales” on page 565.

Description

The `strcasecmp()` function compares `string1` and `string2` without sensitivity to case. All alphabetic characters in `string1` and `string2` are converted to lowercase before comparison.

The `strcasecmp()` function operates on null terminated strings. The string arguments to the function are expected to contain a null character (`\0`) marking the end of the string.

Return Value

The `strcasecmp()` function returns a value indicating the relationship between the two strings, as follows:

<table>
<thead>
<tr>
<th>Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>Less than 0</td>
<td><code>string1</code> less than <code>string2</code></td>
</tr>
<tr>
<td>0</td>
<td><code>string1</code> equivalent to <code>string2</code></td>
</tr>
<tr>
<td>Greater than 0</td>
<td><code>string1</code> greater than <code>string2</code></td>
</tr>
</tbody>
</table>

Table 9. Return values of `strcasecmp()`
Example
This example uses strcasecmp() to compare two strings.

```c
#include <stdio.h>
#include <strings.h>

int main(void)
{
    char_t *str1 = "STRING";
    char_t *str2 = "string";
    int result;
    result = strcasecmp(str1, str2);
    if (result == 0)
        printf("Strings compared equal.\n");
    else if (result < 0)
        printf("\"%s\" is less than \"%s\".\n", str1, str2);
    else
        printf("\"%s\" is greater than \"%s\".\n", str1, str2);
    return 0;
}

/****** The output should be similar to: **************/
Strings compared equal.

/*****************************/
```

Related Information
- “strcasecmp() — Compare Strings without Case Sensitivity” on page 405
- “strncmp() — Compare Strings” on page 408
- “stricmp() — Compare Strings without Case Sensitivity” on page 403
- “wcscmp() — Compare Wide-Character Strings” on page 485
- “wcsncmp() — Compare Wide-Character Strings” on page 497
- “__wcsicmp() — Compare Wide Character Strings without Case Sensitivity” on page 492
- “__wcsnicmp() — Compare Wide Character Strings without Case Sensitivity” on page 500
- “<strings.h>” on page 16

strcat() — Concatenate Strings

Format
```
#include <string.h>
char *strcat(char *string1, const char *string2);
```

Language Level
ANSI

Threading Safe
Yes

Description
The strcat() function concatenates string2 to string1 and ends the resulting string with the null character.
The `strcat()` function operates on null-ended strings. The string arguments to the function should contain a null character (\0) that marks the end of the string. No length checking is performed. You should 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.

**Return Value**

The `strcat()` function returns a pointer to the concatenated string (`string1`).

**Example**

This example creates the string "computer program" using `strcat()`.

```c
#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 should be similar to: *******************/
buffer1 = computer program
```

**Related Information**

- “`strchr()` — Search for Character” on page 386
- “`strcmp()` — Compare Strings” on page 388
- “`strcpy()` — Copy Strings” on page 392
- “`strcspn()` — Find Offset of First Character Match” on page 393
- “`strncat()` — Concatenate Strings” on page 406
- “`wcsat()` — Concatenate Wide-Character Strings” on page 483
- “`wcscat()` — Concatenate Wide-Character Strings” on page 496
- “`<string.h>`” on page 15

---

**strchr() — Search for Character**

**Format**

```c
#include <string.h>
char *strchr(const char *string, int c);
```

**Language Level**

ANSI

**Threadsafe**

Yes
Locale Sensitive
The behavior of this function might be affected by the LC_CTYPE category of the current locale. For more information, see “Understanding CCSIDs and Locales” on page 565.

Description
The strchr() function finds the first occurrence of a character in a 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-ended strings. The string arguments to the function should contain a null character (\0) that marks the end of the string.

Return Value
The strchr() function returns a pointer to the first occurrence of c that is converted to a character in string. The function returns NULL if the specified character is not found.

Example
This example finds the first occurrence of the character "p" in "computer program".

```c
#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 should be similar to:  ********************/

The first occurrence of p in 'computer program' is 'puter program'

Related Information
- “strcat() — Concatenate Strings” on page 385
- “strcmp() — Compare Strings” on page 388
- “strcpy() — Copy Strings” on page 392
- “strcsnp() — Find Offset of First Character Match” on page 393
- “strncmp() — Compare Strings” on page 408
- “strpos() — Find Characters in String” on page 413
- “strchr() — Locate Last Occurrence of Character in String” on page 418
- “strspn() — Find Offset of First Non-matching Character” on page 419
- “wcschr() — Search for Wide Character” on page 484
- “wcsspnp() — Find Offset of First Non-matching Wide Character” on page 507
- “<string.h>” on page 15
strcmp() — Compare Strings

Format

```c
#include <string.h>
int strcmp(const char *string1, const char *string2);
```

Language Level

ANSI

Threadsafe

Yes

Description

The `strcmp()` function compares `string1` and `string2`. The function operates on null-ended strings. The string arguments to the function should contain a null character (\0) that marks the end of the string.

Return Value

The `strcmp()` function returns a value indicating the relationship between the two strings, as follows:

<table>
<thead>
<tr>
<th>Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>Less than 0</td>
<td><code>string1</code> less than <code>string2</code></td>
</tr>
<tr>
<td>0</td>
<td><code>string1</code> identical to <code>string2</code></td>
</tr>
<tr>
<td>Greater than 0</td>
<td><code>string1</code> greater than <code>string2</code></td>
</tr>
</tbody>
</table>

Example

This example compares the two strings that are passed to `main()` using `strcmp()`.
```c
#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] );
}
}

//************** 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**

- “strcat() — Concatenate Strings” on page 385
- “strchr() — Search for Character” on page 386
- “strcpy() — Copy Strings” on page 392
- “strcspn() — Find Offset of First Character Match” on page 393
- “strncmp() — Compare Strings” on page 408
- “strpbrk() — Find Characters in String” on page 413
- “strrchr() — Locate Last Occurrence of Character in String” on page 418
- “strspn() — Find Offset of First Non-matching Character” on page 419
- “wcschr() — Search for Wide Character” on page 484
- “wcsstr() — Find Offset of First Non-matching Wide Character” on page 507
- “<string.h>” on page 15

**strcmpl() — Compare Strings Without Case Sensitivity**

**Format**

```c
#include <string.h>
int strcmpl(const char *string1, const char *string2);
```

**Note:** The `strcmpl` function is available for C++ programs. It is available for C only when the program defines the `__cplusplus__` macro.

**Language Level**

Extension
Threadsafe
Yes

Locale Sensitive
The behavior of this function might be affected by the LC_CTYPE category of the current locale. For more information, see “Understanding CCSIDs and Locales” on page 565.

Description
strcmpi compares string1 and string2 without sensitivity to case. All alphabetic characters in the two arguments string1 and string2 are converted to lowercase before the comparison.

The function operates on null-ended strings. The string arguments to the function are expected to contain a null character (\0) marking the end of the string.

Return Value
strcmpi returns a value indicating the relationship between the two strings, as follows:

<table>
<thead>
<tr>
<th>Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>Less than 0</td>
<td>string1 less than string2</td>
</tr>
<tr>
<td>0</td>
<td>string1 equivalent to string2</td>
</tr>
<tr>
<td>Greater than 0</td>
<td>string1 greater than string2</td>
</tr>
</tbody>
</table>

Example
This example uses strcmpi to compare two strings.

```c
#include <stdio.h>
#include <string.h>
int main(void)
{
    /* Compare two strings without regard to case.                  */
    if (0 == strcmpi("hello", "HELLO"))
        printf("The strings are equivalent.\n");
    else
        printf("The strings are not equivalent.\n");
    return 0;
}
```

The output should be:

The strings are equivalent.

Related Information
• “strcoll() — Compare Strings” on page 391
• “strcspn() — Find Offset of First Character Match” on page 393
• “strdup() — Duplicate String” on page 395
• “strcmp() — Compare Strings without Case Sensitivity” on page 403
• “strncmp() — Compare Strings” on page 408
• “strnicmp() — Compare Substrings Without Case Sensitivity” on page 411
• “wcscmp() — Compare Wide-Character Strings” on page 485
• “wcsncmp() — Compare Wide-Character Strings” on page 497
• “strcasicmp() — Compare Strings without Case Sensitivity” on page 384
• “strnケースcmp() — Compare Strings without Case Sensitivity” on page 405
strcoll() — Compare Strings

Format

```c
#include <string.h>
int strcoll(const char *string1, const char *string2);
```

Language Level

ANSI

Threadsafe

Yes

Locale Sensitive

The behavior of this function might be affected by the LC_COLLATE category of the current locale. For more information, see “Understanding CCSIDs and Locales” on page 565.

Description

The `strcoll()` function compares two strings using the collating sequence that is specified by the program's locale.

Return Value

The `strcoll()` function returns a value indicating the relationship between the strings, as listed below:

<table>
<thead>
<tr>
<th>Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>Less than 0</td>
<td><code>string1</code> less than <code>string2</code></td>
</tr>
<tr>
<td>0</td>
<td><code>string1</code> equivalent to <code>string2</code></td>
</tr>
<tr>
<td>Greater than 0</td>
<td><code>string1</code> greater than <code>string2</code></td>
</tr>
</tbody>
</table>

If `strcoll()` is unsuccessful, errno is changed. The value of errno may be set to EINVAL (the `string1` or `string2` arguments contain characters that are not available in the current locale).

Example

This example compares the two strings that are passed to `main()` using `strcoll()`:
```c
#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] );
    }
}
/**************************** If the input is the strings  ****************************
/**************************** "firststring" and "secondstring",  ****************************
/**************************** then the expected output is:  ****************************
/"firststring" is less than "secondstring"
*/

Related Information
• “setlocale() — Set Locale” on page 366
• “strcmp() — Compare Strings” on page 388
• “strncpy() — Compare Strings” on page 408
• “wcscoll() — Language Collation String Comparison” on page 487
• “<string.h>” on page 15

strcpy() — Copy Strings

Format

```c
#include <string.h>
char *strcpy(char *string1, const char *string2);
```
**Return Value**

The `strcpy()` function returns a pointer to the copied string (`string1`).

**Example**

This example copies the contents of source to destination.

```c
#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 should be similar to:  ****************************
destination is originally = "And this is the destination string"
After strcpy, destination becomes "This is the source string"
*/
```

**Related Information**

- “`strcat()` — Concatenate Strings” on page 385
- “`strchr()` — Search for Character” on page 386
- “`strcmp()` — Compare Strings” on page 388
- “`strcspn()` — Find Offset of First Character Match” on page 393
- “`strncpy()` — Copy Strings” on page 409
- “`strpbrk()` — Find Characters in String” on page 413
- “`strrchr()` — Locate Last Occurrence of Character in String” on page 418
- “`strspn()` — Find Offset of First Non-matching Character” on page 419
- “`wcscpy()` — Copy Wide-Character Strings” on page 488
- “`wcsncpy()` — Copy Wide-Character Strings” on page 499
- “`<string.h>`” on page 15

**strcspn() — Find Offset of First Character Match**

**Format**

```c
#include <string.h>
size_t strcspn(const char *string1, const char *string2);
```

**Language Level**

ANSI

**Threadsafe**

Yes
Locale Sensitive

The behavior of this function might be affected by the LC_CTYPE category of the current locale. For more information, see “Understanding CCSIDs and Locales” on page 565.

Description

The `strcspn()` function finds the first occurrence of a character in `string1` that belongs to the set of characters that is specified by `string2`. Null characters are not considered in the search.

The `strcspn()` function operates on null-ended strings. The string arguments to the function should contain a null character (\0) marking the end of the string.

Return Value

The `strcspn()` function returns the index of the first character found. This value is equivalent to the length of the initial substring of `string1` that consists entirely of characters not in `string2`.

Example

This example uses `strcspn()` to find the first occurrence of any of the characters "a", "x", "l", or "e" in `string`.

```c
#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 should be similar to: **************
The first 10 characters in the string "This is the source string" are not in the string "axle"
*/
```

Related Information

- “`strcat()` — Concatenate Strings” on page 385
- “`strchr()` — Search for Character” on page 386
- “`strcmp()` — Compare Strings” on page 388
- “`strcpy()` — Copy Strings” on page 392
- “`strncpy()` — Compare Strings” on page 408
- “`strpbrk()` — Find Characters in String” on page 413
- “`strstr()` — Locate Last Occurrence of Character in String” on page 418
- “`strspn()` — Find Offset of First Non-matching Character” on page 419
- “`<string.h>`” on page 15
strdup() — Duplicate String

**Format**

```c
#include <string.h>
char *strdup(const char *string);
```

**Note:** The `strdup` function is available for C++ programs. It is available for C only when the program defines the `__cplusplus__strings__` macro.

**Language Level**

XPG4, Extension

**Threadsafe**

Yes

**Description**

`strdup` reserves storage space for a copy of `string` by calling `malloc`. The string argument to this function is expected to contain a null character (`\0`) marking the end of the string. Remember to free the storage reserved with the call to `strdup`.

**Return Value**

`strdup` returns a pointer to the storage space containing the copied string. If it cannot reserve storage, `strdup` returns NULL.

**Example**

This example uses `strdup` to duplicate a string and print the copy.

```c
#include <stdio.h>
#include <string.h>
int main(void)
{
    char *string = "this is a copy";
    char *newstr;
    /* Make newstr point to a duplicate of string */
    if ((newstr = strdup(string)) != NULL)
        printf("The new string is: %s\n", newstr);
    return 0;
}
```

The output should be:

```
The new string is: this is a copy
```

**Related Information**

- “strncpy() — Copy Strings” on page 392
- “strncpy() — Copy Strings” on page 409
- “wcsncpy() — Copy Wide-Character Strings” on page 488
- “wcscspn() — Find Offset of First Wide-Character Match” on page 489
- “<string.h>” on page 15
strerror() — Set Pointer to Runtime Error Message

Format

```c
#include <string.h>
char *strerror(int.errnum);
```

Language Level
ANSI

Threadsafe
Yes

Description
The `strerror()` function maps the error number in `errnum` to an error message string.

Return Value
The `strerror()` function returns a pointer to the string. It does not return a NULL value. The value of `errno` may be set to `ECONVERT` (conversion error).

Example
This example opens a file and prints a runtime error message if an error occurs.

```c
#include <stdlib.h>
#include <string.h>
#include <errno.h>

int main(void)
{
    FILE *stream;
    if ((stream = fopen("mylib/myfile", "r")) == NULL)
        printf(" %s \
", strerror(errno));
}
/*  This is a program fragment and not a complete function example  */
```

Related Information
- “clearerr() — Reset Error Indicators” on page 87
- “ferror() — Test for Read/Write Errors” on page 120
- “perror() — Print Error Message” on page 249
- “<string.h>” on page 15

strfmon() — Convert Monetary Value to String

Format

```c
#include <monetary.h>
int strfmon(char *s, size_t maxsize, const char *format, argument_list);
```
Locale Sensitive

The behavior of this function might be affected by the LC_CTYPE and LC_MONETARY categories of the current locale. This function is not available when LOCALETYPE(*CLD) is specified on the compilation command. For more information, see “Understanding CCSIDs and Locales” on page 565.

Description

The `strfmon()` function 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 stream, and directives, each of which results in the fetching of zero or more arguments, which 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. Only 15 significant digits are guaranteed on conversions involving double values.

A directive consists of a % character, optional conversion specifications, and a ending character that determines the directive's behavior.

A directive consists of the following sequence:

- A % character.
- Optional flags.
- Optional field width.
- Optional left precision.
- Optional right precision.
- A required conversion character indicating the type of conversion to be performed.

Flags

<table>
<thead>
<tr>
<th>Flag</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>=f</td>
<td>An = followed by a single character f which is used as the numeric fill character. By default the numeric fill character is a space character. This flag does not affect field width filling, which always uses a space character. This flag is ignored unless left precision is specified.</td>
</tr>
<tr>
<td>^</td>
<td>Do not use grouping characters when formatting the currency value. Default is to insert grouping characters as defined in the current locale.</td>
</tr>
<tr>
<td>+ or (</td>
<td>Specify the style representing positive and negative currency amounts. If + is specified, the locale's equivalent of + and – for monetary quantities will be used. If ( is specified, negative amounts are enclosed within parenthesis. Default is +.</td>
</tr>
<tr>
<td>!</td>
<td>Do not output the currency symbol. Default is to output the currency symbol.</td>
</tr>
</tbody>
</table>
Table 11. Flags (continued)

<table>
<thead>
<tr>
<th>Flag</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>-</td>
<td>Use left justification for double arguments. Default is right justification.</td>
</tr>
</tbody>
</table>

**Field Width**

w

A decimal digit string \( w \) specifying a minimum field width in bytes in which the result of the conversion is right-justified (or left-justified if the flag - is specified). The default is 0.

**Left Precision**

A # followed by a decimal digit string \( n \) specifying a 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 `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 than \( n \) digit positions are required, this conversion specification is ignored. Digit positions in excess of those actually required are filled with the numeric fill character (see the =f flag above).

If grouping has not been suppressed with the ^ flag, and it is defined for the current locale, grouping separators are inserted before the fill characters (if any) are added. Grouping separators are not applied to fill characters even if the fill character is a digit. 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.

**Right Precision**

A period followed by a decimal digit string \( p \) specifies the number of digits after the radix character. If the value of the right precision \( p \) is 0, no radix character appears. If a right precision is not specified, a default specified by the current locale is used. The amount being formatted is rounded to the specified number of digits prior to formatting.

**Conversion Characters**

<table>
<thead>
<tr>
<th>Specifier</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>%i</td>
<td>The double argument is formatted according to the locale's international currency format.</td>
</tr>
<tr>
<td>%n</td>
<td>The double argument is formatted according to the locale's national currency format.</td>
</tr>
<tr>
<td>%%</td>
<td>Is replaced by %. No argument is converted.</td>
</tr>
</tbody>
</table>

**Return Value**

If the total number of resulting bytes including the ending null character is not more than \( \text{maxsize} \), the `strfmon()` function returns the number of bytes placed into the array pointed to by \( s \), but excludes the ending null character. Otherwise, zero is returned, and the contents of the array are undefined.

The value of `errno` may be set to:
E2BIG
Conversion stopped due to lack of space in the buffer.

Example

```c
#include <stdio.h>
#include <monetary.h>
#include <locale.h>

int main(void)
{
    char string[100];
    double money = 1234.56;
    if (setlocale(LC_ALL, "/qsys.lib/en_us.locale") == NULL) {
        printf("Unable to setlocale().\n");
        exit(1);
    }

    strftime(string, 100, "%i", money); /* USD 1,234.56 */
    printf("%s\n", string);
    strftime(string, 100, "%n", money); /* $1,234.56 */
    printf("%s\n", string);
}
```

The output should be similar to:
USD 1,234.56
$1,234.56

Related Information
- “localeconv() — Retrieve Information from the Environment” on page 202
- “<monetary.h>” on page 6

strftime() — Convert Date/Time to String

**Format**

```c
#include <time.h>
size_t strftime(char *s, size_t maxsize, const char *format,
                const struct tm *timeptr);
```

**Language Level**
ANSI

**Threadsafe**
Yes

**Locale Sensitive**
The behavior of this function might be affected by the LC_CTYPE, LC_TIME, and LC_TOD categories of the current locale. For more information, see “Understanding CCSIDs and Locales” on page 565.

**Description**
The strftime() function places bytes into the array pointed to by s as controlled by the string pointed to by format. The format string consists of zero or more conversion specifications and ordinary characters. A conversion specification consists of a % character and a terminating conversion character that determines the behavior of the conversion. All ordinary characters (including the terminating null byte, and multi-byte chars) are copied unchanged into the array. If copying takes place between objects that overlap, then the behavior is undefined. No more than maxsize bytes are placed in the array.
appropriate characters are determined by the values contained in the structure pointed to by `timeptr`, and by the values stored in the current locale.

Each standard conversion specification is replaced by appropriate characters as described in the following table:

<table>
<thead>
<tr>
<th>Specifier</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>%a</td>
<td>Abbreviated weekday name.</td>
</tr>
<tr>
<td>%A</td>
<td>Full weekday name.</td>
</tr>
<tr>
<td>%b</td>
<td>Abbreviated month name.</td>
</tr>
<tr>
<td>%B</td>
<td>Full month name.</td>
</tr>
<tr>
<td>%c</td>
<td>Date/Time in the format of the locale.</td>
</tr>
<tr>
<td>%C</td>
<td>Century number [00-99], the year divided by 100 and truncated to an integer.</td>
</tr>
<tr>
<td>%d</td>
<td>Day of the month [01-31].</td>
</tr>
<tr>
<td>%D</td>
<td>Date Format, same as %m/%d/%y.</td>
</tr>
<tr>
<td>%e</td>
<td>Same as %d, except single digit is preceded by a space [1-31].</td>
</tr>
<tr>
<td>%g</td>
<td>2 digit year portion of ISO week date [00,99].</td>
</tr>
<tr>
<td>%F</td>
<td>ISO Date Format, same as %Y-%m-%d.</td>
</tr>
<tr>
<td>%G</td>
<td>4 digit year portion of ISO week date. Can be negative.</td>
</tr>
<tr>
<td>%h</td>
<td>Same as %b.</td>
</tr>
<tr>
<td>%H</td>
<td>Hour in 24-hour format [00-23].</td>
</tr>
<tr>
<td>%I</td>
<td>Hour in 12-hour format [01-12].</td>
</tr>
<tr>
<td>%j</td>
<td>Day of the year [001-366].</td>
</tr>
<tr>
<td>%m</td>
<td>Month [01-12].</td>
</tr>
<tr>
<td>%M</td>
<td>Minute [00-59].</td>
</tr>
<tr>
<td>%n</td>
<td>Newline character.</td>
</tr>
<tr>
<td>%p</td>
<td>AM or PM string.</td>
</tr>
<tr>
<td>%r</td>
<td>Time in AM/PM format of the locale. If not available in the locale time format, defaults to the POSIX time AM/PM format: %I:%M:%S %p.</td>
</tr>
<tr>
<td>%R</td>
<td>24-hour time format without seconds, same as %H:%M.</td>
</tr>
<tr>
<td>%S</td>
<td>Second [00-61]. The range for seconds allows for a leap second and a double leap second.</td>
</tr>
<tr>
<td>%t</td>
<td>Tab character.</td>
</tr>
<tr>
<td>%T</td>
<td>24-hour time format with seconds, same as %H:%M:%S.</td>
</tr>
<tr>
<td>%u</td>
<td>Weekday [1,7]. Monday is 1 and Sunday is 7.</td>
</tr>
<tr>
<td>%U</td>
<td>Week number of the year [00-53]. Sunday is the first day of the week.</td>
</tr>
<tr>
<td>%V</td>
<td>ISO week number of the year [01-53]. Monday is the first day of the week. If the week containing January 1st 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 year is week 1 of the new year.</td>
</tr>
<tr>
<td>%w</td>
<td>Weekday [0,6], Sunday is 0.</td>
</tr>
<tr>
<td>Specifier</td>
<td>Meaning</td>
</tr>
<tr>
<td>-----------</td>
<td>---------</td>
</tr>
<tr>
<td>%W</td>
<td>Week number of the year [00-53]. Monday is the first day of the week.</td>
</tr>
<tr>
<td>%x</td>
<td>Date in the format of the locale.</td>
</tr>
<tr>
<td>%X</td>
<td>Time in the format of the locale.</td>
</tr>
<tr>
<td>%y</td>
<td>2-digit year [00,99].</td>
</tr>
<tr>
<td>%Y</td>
<td>4-digit year. Can be negative.</td>
</tr>
<tr>
<td>%z</td>
<td>UTC offset. Output is a string with format +HHMM or -HHMM, where + indicates east of GMT, - indicates west of GMT, HH indicates the number of hours from GMT, and MM indicates the number of minutes from GMT.</td>
</tr>
<tr>
<td>%Z</td>
<td>Time zone name.</td>
</tr>
<tr>
<td>%%</td>
<td>% character.</td>
</tr>
</tbody>
</table>

**Modified Conversion Specifiers**

Some conversion specifiers can be modified by the E or O modifier characters to indicate that an alternate format or specification should be used rather than the one normally used by the unmodified conversion specifier. If a modified conversion specifier uses a field in the current locale that is unavailable, then the behavior will be as if the unmodified conversion specification were used. For example, if the era string is the empty string "", which means that the string is unavailable, then %EY would act like %Y.

<table>
<thead>
<tr>
<th>Specifier</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>%Ec</td>
<td>Date/time for current era.</td>
</tr>
<tr>
<td>%EC</td>
<td>Era name.</td>
</tr>
<tr>
<td>%Ex</td>
<td>Date for current era.</td>
</tr>
<tr>
<td>%EX</td>
<td>Time for current era.</td>
</tr>
<tr>
<td>%Ey</td>
<td>Era year. This is the offset from the base year.</td>
</tr>
<tr>
<td>%EY</td>
<td>Year for current era.</td>
</tr>
<tr>
<td>%Od</td>
<td>Day of the month using alternate digits.</td>
</tr>
<tr>
<td>%Oe</td>
<td>Same as %Od.</td>
</tr>
<tr>
<td>%OH</td>
<td>Hour in 24 hour format using alternate digits.</td>
</tr>
<tr>
<td>%OI</td>
<td>Hour in 12 hour format using alternate digits.</td>
</tr>
<tr>
<td>%OM</td>
<td>Month using alternate digits.</td>
</tr>
<tr>
<td>%OS</td>
<td>Seconds using alternate digits.</td>
</tr>
<tr>
<td>%Ou</td>
<td>Weekday using alternate digits. Monday is 1 and Sunday is 7.</td>
</tr>
<tr>
<td>%OU</td>
<td>Week number of the year using alternate digits. Sunday is the first day of the week.</td>
</tr>
<tr>
<td>%OV</td>
<td>ISO week number of the year using alternate digits. See %V for explanation of ISO week number.</td>
</tr>
<tr>
<td>%Ow</td>
<td>Weekday using alternate digits. Sunday is 0.</td>
</tr>
<tr>
<td>%OW</td>
<td>Week number of the year using alternate digits. Monday is the first day of the week.</td>
</tr>
<tr>
<td>%Oy</td>
<td>2-digit year using alternate digits.</td>
</tr>
<tr>
<td>Specifier</td>
<td>Meaning</td>
</tr>
<tr>
<td>-----------</td>
<td>---------</td>
</tr>
<tr>
<td>%OZ</td>
<td>If the time zone name exists in the current locale, this is the same as %Z; otherwise, the abbreviated time zone name of the current job is returned.</td>
</tr>
</tbody>
</table>

**Note:** %C, %D, %e, %h, %n, %r, %R, %t, %T, %u, %V, and the modified conversion specifiers are not available when LOCALETYPE(*CLD) is specified on the compilation command.

**Return Value**

If the total number of resulting bytes including the terminating null byte is not more than maxsize, strftime() returns the number of bytes placed into the array pointed to by s, not including the terminating null byte. Otherwise, 0 is returned and the contents of the array are indeterminate.

If a conversion error occurs, errno may be set to ECONV.

**Example**

```c
#include <stdio.h>
#include <time.h>

int main(void)
{
    char s[100];
    int rc;
    time_t temp;
    struct tm *timeptr;
    temp = time(NULL);
    timeptr = localtime(&temp);
    rc = strftime(s,sizeof(s),"Today is %A, %b %d.
Time:  %r", timeptr);
    printf("%d characters written.
%s\n",rc,s);
    return 0;
}
```

The output should be similar to:

46 characters written
Today is Wednesday, Oct 24.
Time: 01:01:15 PM

**Related Information**

- “asctime() — Convert Time to Character String” on page 63
- “asctime_r() — Convert Time to Character String (Restartable)” on page 65
- “ctime() — Convert Time to Character String” on page 96
- “ctime64() — Convert Time to Character String” on page 98
- “ctime64_r() — Convert Time to Character String (Restartable)” on page 101
- “ctime_r() — Convert Time to Character String (Restartable)” on page 100
- “gmtime() — Convert Time” on page 183
- “gmtime64() — Convert Time” on page 185
- “gmtime64_r() — Convert Time (Restartable)” on page 189
- “gmtime_r() — Convert Time (Restartable)” on page 187
- “localtime() — Convert Time” on page 207
- “localtime64() — Convert Time” on page 208
- “localtime64_r() — Convert Time (Restartable)” on page 211
- “localtime_r() — Convert Time (Restartable)” on page 210
stricmp() — Compare Strings without Case Sensitivity

Format

```c
#include <string.h>
int stricmp(const char *string1, const char *string2);
```

Note: The `stricmp()` function is available for C++ programs. It is available for C only when the program defines the `__cplusplus__strings__` macro.

Language Level

Extension

Threadsafe

Yes

Locale Sensitive

The behavior of this function might be affected by the LCCTYPE category of the current locale. For more information, see “Understanding CCSIDs and Locales” on page 565.

Description

The `stricmp()` function compares `string1` and `string2` without sensitivity to case. All alphabetic characters in the two arguments `string1` and `string2` are converted to lowercase before the comparison.

The function operates on null-ended strings. The string arguments to the function are expected to contain a null character (`\0`) marking the end of the string.

Return Value

The `stricmp()` function returns a value indicating the relationship between the two strings, as follows:

<table>
<thead>
<tr>
<th>Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>Less than 0</td>
<td><code>string1</code> less than <code>string2</code></td>
</tr>
<tr>
<td>0</td>
<td><code>string1</code> equivalent to <code>string2</code></td>
</tr>
<tr>
<td>Greater than 0</td>
<td><code>string1</code> greater than <code>string2</code></td>
</tr>
</tbody>
</table>

Example

This example uses `stricmp()` to compare two strings.

```c
#include <stdio.h>
#include <string.h>
int main(void)
{
   /* Compare two strings as lowercase */
   if (0 == stricmp("hello", "HELLO"))  
      
      */
```
The strings are equivalent.

Related Information

- “strcmpi() — Compare Strings Without Case Sensitivity” on page 389
- “strcoll() — Compare Strings” on page 391
- “strcsprintf() — Find Offset of First Character Match” on page 393
- “strdup() — Duplicate String” on page 395
- “strncmp() — Compare Strings” on page 408
- “strxcasecmp() — Compare Strings without Case Sensitivity” on page 384
- “strxncasecmp() — Compare Strings without Case Sensitivity” on page 405
- “strxnicmp() — Compare Substrings Without Case Sensitivity” on page 411
- “wcsxcmp() — Compare Wide-Character Strings” on page 485
- “wcsxncmp() — Compare Wide-Character Strings” on page 497
- “<string.h>” on page 15

**strlen() — Determine String Length**

**Format**

```c
#include <string.h>
size_t strlen(const char *string);
```

**Language Level**

ANSI

**Threadsafe**

Yes

**Description**

The `strlen()` function determines the length of `string` excluding the ending null character.

**Return Value**

The `strlen()` function returns the length of `string`.

**Example**

This example determines the length of the string that is passed to `main()`.
```c
#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] ) );
}

/**
 * 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

• “mblen() — Determine Length of a Multibyte Character” on page 219
• “strncat() — Concatenate Strings” on page 406
• “strncmp() — Compare Strings” on page 408
• “strncpy() — Copy Strings” on page 409
• “wcslen() — Calculate Length of Wide-Character String” on page 494
• “<string.h>” on page 15

strncasecmp() — Compare Strings without Case Sensitivity

Format

#include <strings.h>
int strncasecmp(const char *string1, const char *string2, size_t count);

Language Level

XPG4

Threatasafe

Yes

Locale Sensitive

The behavior of this function might be affected by the LC_CTYPE category of the current locale. This
function is not available when LOCALETYPE("CLD") is specified on the compilation command. For more
information, see “Understanding CCSIDs and Locales” on page 565.

Description

The strncasecmp() function compares up to count characters of string1 and string2 without sensitivity
to case. All alphabetic characters in string1 and string2 are converted to lowercase before comparison.

The strncasecmp() function operates on null terminated strings. The string arguments to the function
are expected to contain a null character (\0) marking the end of the string.

Return Value

The strncasecmp() function returns a value indicating the relationship between the two strings, as
follows:
### Table 14. Return values of `strnecmp()`

<table>
<thead>
<tr>
<th>Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>Less than 0</td>
<td><code>string1</code> less than <code>string2</code></td>
</tr>
<tr>
<td>0</td>
<td><code>string1</code> equivalent to <code>string2</code></td>
</tr>
<tr>
<td>Greater than 0</td>
<td><code>string1</code> greater than <code>string2</code></td>
</tr>
</tbody>
</table>

### Example

This example uses `strnecmp()` to compare two strings.

```c
#include <stdio.h>
#include <strings.h>

int main(void)
{
    char_t *str1 = "STRING ONE";
    char_t *str2 = "string TWO";
    int result;
    result = strnecmp(str1, str2, 6);
    if (result == 0)
        printf("Strings compared equal.\n");
    else if (result < 0)
        printf(""%s" is less than "%s".\n", str1, str2);
    else
        printf(""%s" is greater than "%s".\n", str1, str2);
    return 0;
}

/******** The output should be similar to: ************
Strings compared equal.

*******************************************************************/
```

### Related Information
- “`strncasecmp()` — Compare Strings without Case Sensitivity” on page 384
- “`strncmp()` — Compare Strings” on page 408
- “`stricmp()` — Compare Strings without Case Sensitivity” on page 403
- “`wscmp()` — Compare Wide-Character Strings” on page 485
- “`wcsncmp()` — Compare Wide-Character Strings” on page 497
- “`__wcsicmp()` — Compare Wide Character Strings without Case Sensitivity” on page 492
- “`__wcsnicmp()` — Compare Wide Character Strings without Case Sensitivity” on page 500
- “`<strings.h>`” on page 16

### strncat() — Concatenate Strings

**Format**

```c
#include <string.h>
char *strncat(char *string1, const char *string2, size_t count);
```

**Language Level**

ANSI
Threadsafe
Yes

Description
The `strncat()` 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`, the length of `string2` is used in place of `count`.

The `strncat()` function operates on null-ended strings. The string argument to the function should contain a null character (\0) marking the end of the string.

Return Value
The `strncat()` function returns a pointer to the joined string (`string1`).

Example
This example demonstrates the difference between `strcat()` and `strncat()`. The `strcat()` function appends the entire second string to the first, whereas `strncat()` appends only the specified number of characters in the second string to the first.

```c
#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"
", 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"
", buffer1 );
}

*************** Output should be similar to: ***************
strcat : buffer1 = "computer program"
strncat: buffer1 = "computer pr"
```

Related Information
- “`strcat()` — Concatenate Strings” on page 385
- “`strncpy()` — Copy Strings” on page 409
- “`strpbrk()` — Find Characters in String” on page 413
- “`strstr()` — Locate Last Occurrence of Character in String” on page 418
- “`strspn()` — Find Offset of First Non-matching Character” on page 419
- “`wcscat()` — Concatenate Wide-Character Strings” on page 483
- “`wcsncat()` — Concatenate Wide-Character Strings” on page 496
- “<`string.h>`” on page 15
strncmp() — Compare Strings

Format

```c
#include <string.h>
int strncmp(const char *string1, const char *string2, size_t count);
```

Language Level

ANSI

Threadsafe

Yes

Description

The `strncmp()` function compares `string1` and `string2` to the maximum of `count`.

Return Value

The `strncmp()` function returns a value indicating the relationship between the strings, as follows:

<table>
<thead>
<tr>
<th>Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>Less than 0</td>
<td><code>string1</code> less than <code>string2</code></td>
</tr>
<tr>
<td>0</td>
<td><code>string1</code> equivalent to <code>string2</code></td>
</tr>
<tr>
<td>Greater than 0</td>
<td><code>string1</code> greater than <code>string2</code></td>
</tr>
</tbody>
</table>

Example

This example demonstrates the difference between the `strcmp()` function and the `strncmp()` function.
```c
#include <stdio.h>
#include <string.h>
#define SIZE 10

int main(void)
{
    int result;
    int index = 3;
    char buffer1[SIZE] = "abcdefg";
    char buffer2[SIZE] = "abcfg";
    void print_result( int, char *, char *);
    result = strcmp( buffer1, buffer2 );
    printf( "Comparison of each character\n" );
    printf( "  strcmp: " );
    print_result( result, buffer1, buffer2 );
    result = strncmp( buffer1, buffer2, index);
    printf( "\nComparison of only the first %i characters\n", index );
    printf( "  strncmp: " );
    print_result( result, buffer1, buffer2 );
}

void print_result( int res, char * p_buffer1, char * p_buffer2 )
{
    if ( res == 0 )
        printf( "\"%s\" is identical to \"%s\"\n", 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 should be similar to: ***************
Comparison of each character
strcmp: "abcdefg" is less than "abcfg"

Comparison of only the first 3 characters
strncmp: "abcdefg" is identical to "abcfg"
*/

Related Information

• “strcmp() — Compare Strings” on page 388
• “strcspn() — Find Offset of First Character Match” on page 393
• “strncat() — Concatenate Strings” on page 406
• “strncpy() — Copy Strings” on page 409
• “strpbrk() — Find Characters in String” on page 413
• “strrchr() — Locate Last Occurrence of Character in String” on page 418
• “strstr() — Find Offset of First Non-matching Character” on page 419
• “wcscmp() — Compare Wide-Character Strings” on page 485
• “wcsncmp() — Compare Wide-Character Strings” on page 497
• “<string.h>” on page 15
• “__wcsicmp() — Compare Wide Character Strings without Case Sensitivity ” on page 492
• “__wcsnicmp() — Compare Wide Character Strings without Case Sensitivity” on page 500

strncpy() — Copy Strings

Format

#include <string.h>
char *strncpy(char *string1, const char *string2, size_t count);
```
Language Level
ANSI

Threadsafe
Yes

Description
The `strncpy()` function copies `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`.

Return Value
The `strncpy()` function returns a pointer to `string1`.

Example
This example demonstrates the difference between `strcpy()` and `strncpy()`.

```c
#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 );
}
```

Related Information
• “strcpy() — Copy Strings” on page 392
• “strcspn() — Find Offset of First Character Match” on page 393
• “strncat() — Concatenate Strings” on page 406
• “strncmp() — Compare Strings” on page 408
• “strpbrk() — Find Characters in String” on page 413
• “strrchr() — Locate Last Occurrence of Character in String” on page 418
• “strspn() — Find Offset of First Non-matching Character” on page 419
strnicmp() — Compare Substrings Without Case Sensitivity

Format

```
#include <string.h>
int strnicmp(const char *string1, const char *string2, int n);
```

Note: The strnset and strset functions are available for C++ programs. They are available for C only when the program defines the __cplusplus__strings__ macro.

Language Level

Extension

Threaddsafe

Yes

Locale Sensitive

The behavior of this function might be affected by the LC_CTYPE category of the current locale. For more information, see “Understanding CCSIDs and Locales” on page 565.

Description

The strnicmp() function compares, at most, the first n characters of string1 and string2 without sensitivity to case.

The function operates on null terminated strings. The string arguments to the function are expected to contain a null character (\0) marking the end of the string.

Return Value

The strnicmp() function returns a value indicating the relationship between the substrings, as follows:

<table>
<thead>
<tr>
<th>Table 15. Return values of strnicmp()</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Value</strong></td>
</tr>
<tr>
<td>Less than 0</td>
</tr>
<tr>
<td>0</td>
</tr>
<tr>
<td>Greater than 0</td>
</tr>
</tbody>
</table>

Example

This example uses strnicmp() to compare two strings.

```
#include <stdio.h>
#include <string.h>
int main(void)
{
    char *str1 = "THIS IS THE FIRST STRING";
    char *str2 = "This is the second string";
    int numresult;
    /* Compare the first 11 characters of str1 and str2
       without regard to case */
    numresult = strnicmp(str1, str2, 11);
    if (numresult < 0)
        printf("String 1 is less than string2.\n");
    else
        printf("String 1 is equivalent to string2.\n");
    return 0;
}
```
if (numresult > 0)
   printf("String 1 is greater than string2.\n");
else
   printf("The two strings are equivalent.\n");
return 0;
}

The output should be:

The two strings are equivalent.

Related Information

- “strcmp() — Compare Strings” on page 388
- “strcmpi() — Compare Strings Without Case Sensitivity” on page 389
- “stricmp() — Compare Strings without Case Sensitivity” on page 403
- “strncmp() — Compare Strings” on page 408
- “wcsncmp() — Compare Wide-Character Strings” on page 497
- “<string.h>” on page 15

strnset() – strset() — Set Characters in String

Format

```
#include <string.h>
char *strnset(char *string, int c, size_t n);
char *strset(char *string, int c);
```

Note: The strnset and strset functions are available for C++ programs. They are available for C only when the program defines the __cplusplus__strings__ macro.

Language Level

Extension

Threading

Yes

Description

strnset sets, at most, the first n characters of string to c (converted to a char). If n is greater than the length of string, the length of string is used in place of n. strset sets all characters of string, except the ending null character (\0), to c (converted to a char). For both functions, the string is a null-terminated string.

Return Value

Both strset and strnset return a pointer to the altered string. There is no error return value.

Example

In this example, strnset sets not more than four characters of a string to the character 'x'. Then the strset function changes any non-null characters of the string to the character 'k'.

```c
#include <stdio.h>
#include <string.h>
int main(void)
```
```
char str[] = "abcdefghi";
printf("This is the string: %s\n", str);
printf("This is the string after strnset: %s\n", strnset((char*)str, 'x', 4));
printf("This is the string after strset: %s\n", strset((char*)str, 'k'));
return 0;
}
The output should be:
This is the string: abcdefghi
This is the string after strnset: xxxxefghi
This is the string after strset: kkkkkkkkk

Related Information
• “strchr() — Search for Character” on page 386
• “strrchr() — Find Characters in String” on page 413
• “wcschr() — Search for Wide Character” on page 484
• “wcspbrk() — Locate Wide Characters in String” on page 501
• “<string.h>” on page 15

strpbrk() — Find Characters in String

Format
#include <string.h>
char *strpbrk(const char *string1, const char *string2);

Language Level
ANSI

Threadsafe
Yes

Locale Sensitive
The behavior of this function might be affected by the LC_CTYPE category of the current locale. For more information, see “Understanding CCSIDs and Locales” on page 565.

Description
The strpbrk() function locates the first occurrence in the string pointed to by string1 of any character from the string pointed to by string2.

Return Value
The strpbrk() function returns a pointer to the character. If string1 and string2 have no characters in common, a NULL pointer is returned.

Example
This example returns a pointer to the first occurrence in the array string of either a or b.
```c
#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 should be similar to:  ***************
The first occurrence of any of the characters "ab" in "The Blue Danube"
 is "anube"
*/
```

Related Information

- “strchr() — Search for Character” on page 386
- “strcmp() — Compare Strings” on page 388
- “strcspn() — Find Offset of First Character Match” on page 393
- “strncmp() — Compare Strings” on page 408
- “strchr() — Locate Last Occurrence of Character in String” on page 418
- “strspn() — Find Offset of First Non-matching Character” on page 419
- “wcschr() — Search for Wide Character” on page 484
- “wcscspn() — Find Offset of First Wide-Character Match” on page 489
- “wcschr() — Locate Last Occurrence of Wide Character in String” on page 501
- “wcscwcs() — Locate Wide-Character Substring” on page 522
- “<string.h>” on page 15

**strptime() — Convert String to Date/Time**

**Format**

```c
#include <time.h>
char *strptime(const char *buf, const char *format, struct tm *tm);
```

**Language Level**

XPG4

**Threadsafe**

Yes

**Locale Sensitive**

The behavior of this function might be affected by the LC_CTYPE, LC_TIME, and LC_TOD categories of the current locale. This function is not available when LOCALETYPE(*CLD) is specified on the compilation command. For more information, see “Understanding CCSIDs and Locales” on page 565.
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 `format`.

The `format` contains zero or more directives. A directive contains either an ordinary character (not % or a white space), or a conversion specification. Each conversion specification is composed of a % character followed by one or more conversion characters, which specify the replacement required. There must be a white space or other delimiter in both `buf` and `format` to be guaranteed that the function will behave as expected. There must be a delimiter between two string-to-number conversions, or the first number conversion may convert characters that belong to the second conversion specifier.

Any whitespace (as specified by `isspace()`) encountered before a directive is scanned in either the format string or the input string will be ignored. A directive that is an ordinary character must exactly match the next scanned character in the input string. Case is relevant when matching ordinary character directives. If the ordinary character directive in the format string does not match the character in the input string, `strptime()` is not successful. No more characters will be scanned.

Any other conversion specification is matched by scanning characters in the input string until a character that is not a possible character for that specification is found or until no more characters can be scanned. If the specification was string-to-number, the possible character range is +,- or a character specified by `isdigit()`. Number specifiers do not require leading zeros. If the specification needs to match a field in the current locale, scanning is repeated until a match is found. Case is ignored when matching fields in the locale. If a match is found, the structure pointed to by `tm` will be updated with the corresponding locale information. If no match is found, `strptime()` is not successful. No more characters will be scanned.

Missing fields in the `tm` structure may be filled in by `strptime()` if given enough information. For example, if a date is given, `tm_yday` can be calculated.

Each standard conversion specification is replaced by appropriate characters as described in the following table:

<table>
<thead>
<tr>
<th>Specifier</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>%a</td>
<td>Name of day of the week, can be either the full name or an abbreviation.</td>
</tr>
<tr>
<td>%A</td>
<td>Same as %a.</td>
</tr>
<tr>
<td>%b</td>
<td>Month name, can be either the full name or an abbreviation.</td>
</tr>
<tr>
<td>%B</td>
<td>Same as %b.</td>
</tr>
<tr>
<td>%c</td>
<td>Date/time, in the format of the locale.</td>
</tr>
<tr>
<td>%C</td>
<td>Century number [00–99]. Calculates the year if a two-digit year is used.</td>
</tr>
<tr>
<td>%d</td>
<td>Day of the month [1–31].</td>
</tr>
<tr>
<td>%D</td>
<td>Date format, same as %m/%d/%y.</td>
</tr>
<tr>
<td>%e</td>
<td>Same as %d.</td>
</tr>
<tr>
<td>%g</td>
<td>2 digit year portion of ISO week date [00–99].</td>
</tr>
<tr>
<td>%G</td>
<td>4 digit year portion of ISO week date. Can be negative.</td>
</tr>
<tr>
<td>%h</td>
<td>Same as %b.</td>
</tr>
<tr>
<td>%H</td>
<td>Hour in 24-hour format [0–23].</td>
</tr>
<tr>
<td>%I</td>
<td>Hour in 12-hour format [1–12].</td>
</tr>
<tr>
<td>%j</td>
<td>Day of the year [1-366].</td>
</tr>
<tr>
<td>%m</td>
<td>Month [1-12].</td>
</tr>
<tr>
<td>%M</td>
<td>Minute [0-59].</td>
</tr>
<tr>
<td>Specifier</td>
<td>Meaning</td>
</tr>
<tr>
<td>-----------</td>
<td>---------</td>
</tr>
<tr>
<td>%n</td>
<td>Skip all whitespaces until a newline character is found.</td>
</tr>
<tr>
<td>%p</td>
<td>AM or PM string, used for calculating the hour if 12-hour format is used.</td>
</tr>
<tr>
<td>%r</td>
<td>Time in AM/PM format of the locale. If not available in the locale time format, defaults to the POSIX time AM/PM format: %I:%M:%S %p.</td>
</tr>
<tr>
<td>%R</td>
<td>24-hour time format without seconds, same as %H:%M.</td>
</tr>
<tr>
<td>%S</td>
<td>Second [00-61]. The range for seconds allows for a leap second and a double leap second.</td>
</tr>
<tr>
<td>%t</td>
<td>Skip all whitespaces until a tab character is found.</td>
</tr>
<tr>
<td>%T</td>
<td>24 hour time format with seconds, same as %H:%M:%S.</td>
</tr>
<tr>
<td>%u</td>
<td>Weekday [1–7]. Monday is 1 and Sunday is 7.</td>
</tr>
<tr>
<td>%U</td>
<td>Week number of the year [0-53], Sunday is the first day of the week. Used in calculating the day of the year.</td>
</tr>
<tr>
<td>%V</td>
<td>ISO week number of the year [1-53]. Monday is the first day of the week. If the week containing January 1st has four or more days in the new year, it is considered week 1. Otherwise, it is the last week of the previous year, and the next week is week 1 of the new year. Used in calculating the day of the year.</td>
</tr>
<tr>
<td>%w</td>
<td>Weekday [0-6]. Sunday is 0.</td>
</tr>
<tr>
<td>%W</td>
<td>Week number of the year [0-53], Sunday is the first day of the week. Used in calculating the day of the year.</td>
</tr>
<tr>
<td>%x</td>
<td>Date in the format of the locale.</td>
</tr>
<tr>
<td>%X</td>
<td>Time in the format of the locale.</td>
</tr>
<tr>
<td>%y</td>
<td>2-digit year [0-99].</td>
</tr>
<tr>
<td>%Y</td>
<td>4-digit year. Can be negative.</td>
</tr>
<tr>
<td>%z</td>
<td>UTC offset. Output is a string with format +HHMM or -HHMM, where + indicates east of GMT, - indicates west of GMT, HH indicates the number of hours from GMT, and MM indicates the number of minutes from GMT.</td>
</tr>
<tr>
<td>%Z</td>
<td>Time zone name.</td>
</tr>
<tr>
<td>%%</td>
<td>% character.</td>
</tr>
</tbody>
</table>

**Modified Conversion Specifiers**

Some conversion specifiers can be modified by the E or O modifier characters to indicate that an alternate format or specification should be used. If a modified conversion specifier uses a field in the current locale that is unavailable, then the behavior will be as if the unmodified conversion specification were used. For example, if the era string is the empty string "", which means that era is unavailable, then %EY would act like %Y.

<table>
<thead>
<tr>
<th>Specifier</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>%Ec</td>
<td>Date/time for current era.</td>
</tr>
<tr>
<td>%EC</td>
<td>Era name.</td>
</tr>
<tr>
<td>%Ex</td>
<td>Date for current era.</td>
</tr>
<tr>
<td>%EX</td>
<td>Time for current era.</td>
</tr>
<tr>
<td>Specifier</td>
<td>Meaning</td>
</tr>
<tr>
<td>-----------</td>
<td>---------</td>
</tr>
<tr>
<td>%Ey</td>
<td>Era year. This is the offset from the base year.</td>
</tr>
<tr>
<td>%EY</td>
<td>Year for the current era.</td>
</tr>
<tr>
<td>%Od</td>
<td>Day of the month using alternate digits.</td>
</tr>
<tr>
<td>%Oe</td>
<td>Same as %Od.</td>
</tr>
<tr>
<td>%OH</td>
<td>Hour in 24-hour format using alternate digits.</td>
</tr>
<tr>
<td>%OI</td>
<td>Hour in 12-hour format using alternate digits.</td>
</tr>
<tr>
<td>%Om</td>
<td>Month using alternate digits.</td>
</tr>
<tr>
<td>%OM</td>
<td>Minutes using alternate digits.</td>
</tr>
<tr>
<td>%OS</td>
<td>Seconds using alternate digits.</td>
</tr>
<tr>
<td>%Ou</td>
<td>Day of the week using alternate digits. Monday is 1 and Sunday is 7.</td>
</tr>
<tr>
<td>%OU</td>
<td>Week number of the year using alternate digits. Sunday is the first day of the week.</td>
</tr>
<tr>
<td>%OV</td>
<td>ISO week number of the year using alternate digits. See %V for explanation of ISO week number.</td>
</tr>
<tr>
<td>%Ow</td>
<td>Weekday using alternate digit. Sunday is 0 and Saturday is 6.</td>
</tr>
<tr>
<td>%OW</td>
<td>Week number of the year using alternate digits. Monday is the first day of the week.</td>
</tr>
<tr>
<td>%Oy</td>
<td>2-digit year using alternate digits.</td>
</tr>
<tr>
<td>%OZ</td>
<td>Abbreviated time zone name.</td>
</tr>
</tbody>
</table>

**Return Value**

On successful completion, the `strptime()` function returns a pointer to the character following the last character parsed. Otherwise, a null pointer is returned. The value of `errno` may be set to `ECONVERT` (conversion error).

**Example**

```c
#include <stdio.h>
#include <locale.h>
#include <time.h>

int main(void)
{
    char buf[100];
    time_t t;
    struct tm *timeptr,result;

    setlocale(LC_ALL,"/QSYS.LIB/EN_US.LOCALE");
    t = time(NULL);
    timeptr = localtime(&t);
    strftime(buf,sizeof(buf), "%a %m/%d/%Y %r", timeptr);
    if (strptime(buf, "%a %m/%d/%Y %r", &result) == NULL)
        printf("\nstrptime failed\n");
    else
    {
        printf("tm_hour:  %d\n",result.tm_hour);
        printf("tm_min:  %d\n",result.tm_min);
        printf("tm_sec:  %d\n",result.tm_sec);
        printf("tm_mon:  %d\n",result.tm_mon);
        printf("tm_mday:  %d\n",result.tm_mday);
        printf("tm_year:  %d\n",result.tm_year);
        printf("tm_yday:  %d\n",result.tm_yday);
        printf("tm_wday:  %d\n",result.tm_wday);
    }
}
```

Library Functions  417
```c
return 0;
}

/**************************
The output should be similar to:
Tue 10/30/2001 10:59:10 AM
tm_hour: 10
tm_min: 59
tm_sec: 10
tm_mon: 9
tm_mday: 30
tm_year: 101
tm_yday: 302
tm_wday: 2
*****************************/
```

**Related Information**

- “asctime() — Convert Time to Character String” on page 63
- “asctime_r() — Convert Time to Character String (Restartable)” on page 65
- “ctime() — Convert Time to Character String” on page 96
- “ctime64() — Convert Time to Character String” on page 98
- “ctime64_r() — Convert Time to Character String (Restartable)” on page 101
- “ctime_r() — Convert Time to Character String ( Restartable)” on page 100
- “gmtime() — Convert Time” on page 183
- “gmtime64() — Convert Time” on page 185
- “gmtime64_r() — Convert Time (Restartable)” on page 189
- “gmtime_r() — Convert Time ( Restartable)” on page 187
- “localtime() — Convert Time” on page 207
- “localtime64() — Convert Time” on page 208
- “localtime64_r() — Convert Time (Restartable)” on page 211
- “localtime_r() — Convert Time (Restartable)” on page 210
- “setlocale() — Set Locale” on page 366
- “strftime() — Convert Date/Time to String” on page 399
- “time() — Determine Current Time” on page 441
- “time64() — Determine Current Time” on page 443
- “<time.h>” on page 16
- “wcsptime() — Convert Wide Character String to Date/Time” on page 502

---

**strrchr() — Locate Last Occurrence of Character in String**

**Format**

```c
#include <string.h>
char *strrchr(const char *string, int c);
```

**Language Level**

ANSI

**Threadsafe**

Yes
Locale Sensitive

The behavior of this function might be affected by the LC_CTYPE category of the current locale. For more information, see “Understanding CCSIDs and Locales” on page 565.

Description

The `strrchr()` function finds the last occurrence of `c` (converted to a character) in `string`. The ending null character is considered part of the `string`.

Return Value

The `strrchr()` function returns a pointer to the last occurrence of `c` in `string`. If the given character is not found, a NULL pointer is returned.

Example

This example compares the use of `strchr()` and `strrchr()`. It searches the string for the first and last occurrence of `p` in the string.

```c
#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'
", ch, buf, ptr );
    /* This illustrates strrchr */
    ptr = strrchr( buf, ch );
    printf( "The last occurrence of %c in '%s' is '%s'
", ch, buf, ptr );
}
```

/* Output should be similar to: */
The first occurrence of p in 'computer program' is 'puter program'
The last occurrence of p in 'computer program' is 'program'

Related Information

- “`strchr()` — Search for Character” on page 386
- “`strcmp()` — Compare Strings” on page 388
- “`strcspn()` — Find Offset of First Character Match” on page 393
- “`strncmp()` — Compare Strings” on page 408
- “`strpbrk()` — Find Characters in String” on page 413
- “`strspn()` — Find Offset of First Non-matching Character” on page 419
- “<string.h>” on page 15

`strspn()` — Find Offset of First Non-matching Character

Format

```c
#include <string.h>
size_t strspn(const char *string1, const char *string2);
```
Language Level
ANSI

Threadsafe
Yes

Locale Sensitive
The behavior of this function might be affected by the LC_CTYPE category of the current locale. For more information, see “Understanding CCSIDs and Locales” on page 565.

Description
The `strspn()` function finds the first occurrence of a character in `string1` that is not contained in the set of characters that is specified by `string2`. The null character (\0) that ends `string2` is not considered in the matching process.

Return Value
The `strspn()` function returns the index of the first character found. This value is equal to the length of the initial substring of `string1` that consists entirely of characters from `string2`. If `string1` begins with a character not in `string2`, the `strspn()` function returns 0. If all the characters in `string1` are found in `string2`, the length of `string1` is returned.

Example
This example finds the first occurrence in the array `string` of a character that is not an a, b, or c. Because the string in this example is cabbage, the `strspn()` function returns 5, the length of the segment of cabbage before a character that is not an a, b, or c.

```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"
            index, string, source );
}
```

Related Information
- “strcat() — Concatenate Strings” on page 385
- “strchr() — Search for Character” on page 386
- “strcmp() — Compare Strings” on page 388
- “strcpy() — Copy Strings” on page 392
- “strcspn() — Find Offset of First Character Match” on page 393
- “strchr() — Find Characters in String” on page 413
- “strchr() — Locate Last Occurrence of Character in String” on page 418
- “wcschr() — Search for Wide Character” on page 484
- “wcscspn() — Find Offset of First Wide-Character Match” on page 489
### strstr() — Locate Substring

**Format**

```c
#include <string.h>
char *strstr(const char *string1, const char *string2);
```

**Language Level**

ANSI

**Threading Safe**

Yes

**Description**

The `strstr()` function finds the first occurrence of `string2` in `string1`. The function ignores the null character (\0) that ends `string2` in the matching process.

**Return Value**

The `strstr()` function returns a pointer to the beginning of the first occurrence of `string2` in `string1`. If `string2` does not appear in `string1`, the `strstr()` function returns NULL. If `string2` points to a string with zero length, the `strstr()` function returns `string1`.

**Example**

This example locates the string "haystack" in the string "needle in a haystack".

```c
#include <string.h>
#include <stdio.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 should be similar to: *******************
haystack
*/
```

**Related Information**

- “strchr() — Search for Character” on page 386
- “strcmp() — Compare Strings” on page 388
- “strcspn() — Find Offset of First Character Match” on page 393
strtod() - strtof() - strtold() — Convert Character String to Double, Float, and Long Double

Format

```c
#include <stdlib.h>
double strtod(const char *nptr, char **endptr);
float strtof(const char *nptr, char **endptr);
long double strtold(const char *nptr, char **endptr);
```

Language Level

ANSI

Threadsafe

Yes

Locale Sensitive

The behavior of these functions might be affected by the LC_CTYPE and LC_NUMERIC categories of the current locale. For more information, see “Understanding CCSIDs and Locales” on page 565.

Description

The `strtod()`, `strtof()`, and `strtold()` functions convert a character string to a double, float, or long double value. The parameter `nptr` points to a sequence of characters that can be interpreted as a numeric binary floating-point value. These functions stop reading the string at the first character that is not recognized as part of a number. This character can be the null character at the end of the string.

The `strtod()`, `strtof()`, and `strtold()` functions expect `nptr` to point to a string with the following form:
The first character that does not fit this form stops the scan. In addition, a sequence of INFINITY or NAN (ignoring case) is allowed.

If an exponent is specified with the hexadecimal digit form, the exponent is interpreted as a binary (base 2) exponent. If an exponent is specified with the decimal digit form, the exponent is interpreted as a decimal (base 10) exponent.

**Return Value**

The `strtod()`, `strtof()`, and `strtold()` functions return the value of the floating-point number, except when the representation causes an underflow or overflow. For an overflow, `strtof()` returns `HUGE_VALF` or `-HUGE_VALF`; `strtod()` and `strtold()` return `HUGE_VAL` or `-HUGE_VAL`. For an underflow, all functions return 0.

In both cases, `errno` is set to `ERANGE`. If the string pointed to by `nptr` does not have the expected form, no conversion is performed and the value of `nptr` is stored in the object pointed to by `endptr`, provided that `endptr` is not a NULL pointer.

The `strtod()`, `strtof()`, and `strtold()` functions do not fail if a character other than a digit follows an E or e that is read as an exponent. For example, 100elf is converted to the floating-point value 100.0.

A character sequence of INFINITY (ignoring case) yields a value of INFINITY. A character value of NAN yields a Quiet Not-A-Number (NAN) value.

**Example**

This example converts the strings to double, float, and long double values. It prints the converted values and the substring that stopped the conversion.
```c
#include <stdlib.h>
#include <stdio.h>

int main(void)
{
    char *string, *stopstring;
    double x;
    float f;
    long double ld;

    string = "3.1415926This stopped it";
    f = strtof(string, &stopstring);
    printf("string = \"%s\"\n", string);
    printf("   strtof = %f\n", f);
    printf("   Stopped scan at \"%s\"\n\", stopstring);
    string = "100ergs";
    f = strtof(string, &stopstring);
    printf("string = \"%s\"\n", string);
    printf("   strtof = %f\n", f);
    printf("   Stopped scan at \"%s\"\n\", stopstring);

    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\", stopstring);
    string = "100ergs";
    x = strtod(string, &stopstring);
    printf("string = \"%s\"\n", string);
    printf("   strtod = %f\n", x);
    printf("   Stopped scan at \"%s\"\n\", stopstring);

    string = "3.1415926This stopped it";
    ld = strtold(string, &stopstring);
    printf("string = \"%s\"\n", string);
    printf("   strtold = %lf\n", ld);
    printf("   Stopped scan at \"%s\"\n\", stopstring);
    string = "100ergs";
    ld = strtold(string, &stopstring);
    printf("string = \"%s\"\n", string);
    printf("   strtold = %lf\n", ld);
    printf("   Stopped scan at \"%s\"\n\", stopstring);
}

/******************** Output should be similar to: **********************/
string = "3.1415926This stopped it"
strtof = 3.141593
Stopped scan at "This stopped it"

string = "100ergs"
    strtof = 100.000000
    Stopped scan at "ergs"

string = "3.1415926This stopped it"
    strtod = 3.141593
    Stopped scan at "This stopped it"

string = "100ergs"
    strtod = 100.000000
    Stopped scan at "ergs"

string = "3.1415926This stopped it"
    strtold = 3.141593
    Stopped scan at "This stopped it"

string = "100ergs"
    strtold = 100.000000
    Stopped scan at "ergs"

/*

Related Information
• "atof() — Convert Character String to Float" on page 71
• "atoi() — Convert Character String to Integer" on page 72
• "atol() — atoll() — Convert Character String to Long or Long Long Integer" on page 74
*/
```
strtod32() - strtod64() - strtod128() — Convert Character String to Decimal Floating-Point

Format

```c
#define __STDC_WANT_DEC_FP__
#include <stdlib.h>
_Decimal32 strtod32(const char *nptr, char **endptr);
_Decimal64 strtod64(const char *nptr, char **endptr);
_Decimal128 strtod132(const char *nptr, char **endptr);
```

Language Level

ANSI

Threadsafe

Yes

Locale Sensitive

The behavior of these functions might be affected by the LC_CTYPE and LC_NUMERIC categories of the current locale. For more information, see “Understanding CCSIDs and Locales” on page 565.

Description

The `strtod32()`, `strtod64()`, and `strtod128()` functions convert a character string to a single-precision, double-precision, or quad-precision decimal floating-point value. The parameter `nptr` points to a sequence of characters that can be interpreted as a numeric decimal floating-point value. These functions stop reading the string at the first character that is not recognized as part of a number. This character can be the null character at the end of the string. The `endptr` parameter is updated to point to this character, provided that `endptr` is not a NULL pointer.

The `strtod32()`, `strtod64()`, and `strtod128()` functions expect `nptr` to point to a string with the following form:

```
whitespace + digits . digits digits e + digits
```

Library Functions 425
The first character that does not fit this form stops the scan. In addition, a sequence of INFINITY or NAN (ignoring case) is allowed.

**Return Value**

The `strtod32()`, `strtod64()`, and `strtod128()` functions return the value of the floating-point number, except when the representation causes an underflow or overflow. For an overflow, `strtod32()` returns `HUGE_VAL_D32` or `-HUGE_VAL_D32`; `strtod64()` returns `HUGE_VAL_D64` or `-HUGE_VAL_D64`; `strtod128()` returns `HUGE_VAL_D128` or `-HUGE_VAL_D128`. For an underflow, all functions return +0.E0.

In both the overflow and underflow cases, `errno` is set to `ERANGE`. If the string pointed to by `nptr` does not have the expected form, a value of +0.E0 is returned and the value of `nptr` is stored in the object pointed to by `endptr`, provided that `endptr` is not a NULL pointer.

The `strtod32()`, `strtod64()`, and `strtod128()` functions do not fail if a character other than a digit follows an E or e that is read as an exponent. For example, 100elf is converted to the floating-point value 100.0.

A character sequence of INFINITY (ignoring case) yields a value of INFINITY. A character value of NAN yields a Quiet Not-A-Number (NaN) value.

If necessary, the return value is rounded using the rounding mode Round to Nearest, Ties to Even.

**Example**

This example converts the strings to single-precision, double-precision, and quad-precision decimal floating-point values. It prints the converted values and the substring that stopped the conversion.
#define __STDC_WANT_DEC_FP__
#include <stdlib.h>
#include <stdio.h>

int main(void)
{
    char *string, *stopstring;
    _Decimal32 d32;
    _Decimal64 d64;
    _Decimal128 d128;

    string = "3.1415926This stopped it";
    d32 = strtod32(string, &stopstring);
    printf("string = %s\n", string);
    printf(" strtod32 = %Hf\n", d32);
    printf(" Stopped scan at %s\n", stopstring);
    string = "100ergs";
    d32 = strtod32(string, &stopstring);
    printf("string = \"%s\"\n", string);
    printf(" strtof = %Hf\n", d32);
    printf(" Stopped scan at \"%s\"\n", stopstring);

    string = "3.1415926This stopped it";
    d64 = strtod64(string, &stopstring);
    printf("string = %s\n", string);
    printf(" strtod64 = %Df\n", d64);
    printf(" Stopped scan at %s\n", stopstring);
    string = "100ergs";
    d64 = strtod64(string, &stopstring);
    printf("string = \"%s\"\n", string);
    printf(" strtof = %Df\n", d64);
    printf(" Stopped scan at \"%s\"\n", stopstring);

    string = "3.1415926This stopped it";
    d128 = strtod128(string, &stopstring);
    printf("string = %s\n", string);
    printf(" strtod128 = %DDf\n", d128);
    printf(" Stopped scan at %s\n", stopstring);
    string = "100ergs";
    d128 = strtod128(string, &stopstring);
    printf("string = \"%s\"\n", string);
    printf(" strtof = %DDf\n", d128);
    printf(" Stopped scan at \"%s\"\n", stopstring);
}

/**************************** Output should be similar to: ***************

string = 3.1415926This stopped it
strtof = 3.141593
Stopped scan at This stopped it

string = "100ergs"
strtof = 100.000000
Stopped scan at "ergs"

string = 3.1415926This stopped it
strtod= 3.141593
Stopped scan at This stopped it

string = "100ergs"
strtod = 100.000000
Stopped scan at "ergs"

string = 3.1415926This stopped it
strtodl = 3.141593
Stopped scan at This stopped it

string = "100ergs"
strtodl = 100.000000
Stopped scan at "ergs"

/*****

Related Information
• “atof() — Convert Character String to Float” on page 71
strtok() — Tokenize String

Format

```c
#include <string.h>
char *strtok(char *string1, const char *string2);
```

Language Level

ANSI

Threadsafe

No

Use `strtok_r()` instead.

Locale Sensitive

The behavior of this function might be affected by the LC_CTYPE category of the current locale. For more information, see “Understanding CCSIDs and Locales” on page 565.

Description

The `strtok()` function reads `string1` as a series of zero or more tokens, and `string2` as the set of characters serving as delimiters of the tokens in `string1`. The tokens in `string1` can be separated by one or more of the delimiters from `string2`. The tokens in `string1` can be located by a series of calls to the `strtok()` function.

In the first call to the `strtok()` function for a given `string1`, the `strtok()` function searches for the first token in `string1`, skipping over leading delimiters. A pointer to the first token is returned.

When the `strtok()` function is called with a NULL `string1` argument, the next token is read from a stored copy of the last non-null `string1` parameter. Each delimiter is replaced by a null character. The set of delimiters can vary from call to call, so `string2` can take any value. Note that the initial value of `string1` is not preserved after the call to the `strtok()` function.

Note that the `strtok()` function writes data into the buffer. The function should be passed to a non-critical buffer containing the string to be tokenized because the buffer will be damaged by the `strtok()` function.
Return Value

The first time the `strtok()` function is called, it returns a pointer to the first token in `string1`. In later calls with the same token string, the `strtok()` function 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-ended.

Note: The `strtok()` function uses an internal static pointer to point to the next token in the string being tokenized. A reentrant version of the `strtok()` function, `strtok_r()`, which does not use any internal static storage, can be used in place of the `strtok()` function.

Example

Using a loop, this example gathers tokens, separated by commas, from a string until no tokens are left. The example prints the tokens, a string, of, and tokens.

```
#include <stdio.h>
#include <string.h>

int main(void)
{
    char *token, *string = "a string, of, ,tokens\0,after null terminator";

    /* the string pointed to by string is broken up into the tokens
       "a string", " of", " ", and "tokens" ; the null terminator (\0)
       is encountered and execution stops after the token "tokens"     */
    token = strtok(string, ",");
    do
    {
        printf("token: %s\n", token);
    }
    while (token = strtok(NULL, ",");

    /*************** Output should be similar to: ***************
    token: a string
    token: of
    token: tokens
    */
```

Related Information

- “strcat() — Concatenate Strings” on page 385
- “strchr() — Search for Character” on page 386
- “strcmp() — Compare Strings” on page 388
- “strcpy() — Copy Strings” on page 392
- “strcspn() — Find Offset of First Character Match” on page 393
- “strspn() — Find Offset of First Non-matching Character” on page 419
- “strtok_r() — Tokenize String (Restartable)” on page 429
- “<string.h>” on page 15

`strtok_r()` — Tokenize String (Restartable)

Format

```
#include <string.h>
char *strtok_r(char *string, const char *seps,
               char **lasts);
```

Language Level

XPG4
Threadsafe
Yes

Locale Sensitive
The behavior of this function might be affected by the LC_CTYPE category of the current locale. For more information, see “Understanding CCSIDs and Locales” on page 565.

Description
This function is the restartable version of `strtok()`. The `strtok_r()` function reads `string` as a series of zero or more tokens, and `seps` as the set of characters serving as delimiters of the tokens in `string`. The tokens in `string` can be separated by one or more of the delimiters from `seps`. The arguments `lasts` points to a user-provided pointer, which points to stored information necessary for the `strtok_r()` function to continue scanning the same string.

In the first call to the `strtok_r()` function for a given null-ended `string`, it searches for the first token in `string`, skipping over leading delimiters. It returns a pointer to the first character of the first token, writes a null character into `string` immediately following the returned token, and updates the pointer to which `lasts` points.

To read the next token from `string`, call the `strtok_r()` function with a NULL `string` argument. This causes the `strtok_r()` function to search for the next token in the previous token string. Each delimiter in the original `string` is replaced by a null character, and the pointer to which `lasts` points is updated. The set of delimiters in `seps` can vary from call to call, but `lasts` must remain unchanged from the previous call. When no tokens remain in `string`, a NULL pointer is returned.

Return Value
The first time the `strtok_r()` function is called, it returns a pointer to the first token in `string`. In later calls with the same token string, the `strtok_r()` function 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-ended.

Related Information
• “`strcat()` — Concatenate Strings” on page 385
• “`strchr()` — Search for Character” on page 386
• “`strcmp()` — Compare Strings” on page 388
• “`strcpy()` — Copy Strings” on page 392
• “`strcspn()` — Find Offset of First Character Match” on page 393
• “`strspn()` — Find Offset of First Non-matching Character” on page 419
• “`strtok()` — Tokenize String” on page 428
• “<string.h>” on page 15

---

**strtol() – strtoll() — Convert Character String to Long and Long Long Integer**

Format (strtol())
```
#include <stdlib.h>
long int strtol(const char *nptr, char **endptr, int base);
```

Format (strtoll())
```
#include <stdlib.h>
long long int strtoll(char *string, char **endptr, int base);
```
Language Level
ANSI

Threadsafe
Yes

Locale Sensitive
The behavior of these functions might be affected by the LC_CTYPE category of the current locale. For more information, see “Understanding CCSIDs and Locales” on page 565.

Description
The `strtol()` function converts a character string to a long integer value. The parameter `nptr` points to a sequence of characters that can be interpreted as a numeric value of type long int.

The `strtoll()` function converts a character string to a long long integer value. The parameter `nptr` points to a sequence of characters that can be interpreted as a numeric value of type long long int.

When you use these functions, the `nptr` parameter should point to a string with the following form:

```
whitespace + - 0 0x 0X digits
```

If the `base` parameter is a value between 2 and 36, the subject sequence’s expected form is a sequence of letters and digits representing an integer whose radix is specified by the `base` parameter. This sequence is optionally preceded by a positive (+) or negative (-) sign. Letters from a to z inclusive (either upper or lower case) are ascribed the values 10 to 35; only letters whose ascribed values are less than that of the base parameter are permitted. If the base parameter has a value of 16, the characters 0x or 0X optionally precede the sequence of letters and digits, following the positive (+) or negative (-) sign, if present.

If the value of the base parameter is 0, the string determines the base. After an optional leading sign a leading 0 indicates octal conversion, a leading 0x or 0X indicates hexadecimal conversion, and all other leading characters result in decimal conversion.

These functions scan the string up to the first character that is inconsistent with the `base` parameter. This character may be the null character (‘\0’) at the end of the string. Leading white-space characters are ignored, and an optional sign may precede the digits.

If the value of the `endptr` parameter is not null a pointer, a pointer to the character that ended the scan is stored in the value pointed to by `endptr`. If a value cannot be formed, the value pointed to by `endptr` is set to the `nptr` parameter.

Return Value
If base has an invalid value (less than 0, 1, or greater than 36), `errno` is set to EINVAL and 0 is returned. The value pointed to by the `endptr` parameter is set to the value of the `nptr` parameter.

If the value is outside the range of representable values, `errno` is set to ERANGE. If the value is positive, the `strtol()` function will return LONG_MAX, and the `strtoll()` function will return LONGLONG_MAX. If the value is negative, the `strtol()` function will return LONG_MIN, and the `strtoll()` function will return LONGLONG_MIN.

If no characters are converted, the `strtoll()` and `strtol()` functions will set `errno` to EINVAL and 0 is returned. For both functions, the value pointed to by `endptr` is set to the value of the `nptr` parameter. Upon successful completion, both functions return the converted value.
This example converts the strings to a long value. It prints out the converted value and the substring that stopped the conversion.

```c
#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 should be similar to:

```
string = 10110134932
strtol = 45 (base 2)
Stopped scan at 34932

strtol = 4423 (base 4)
Stopped scan at 4932
```

### Related Information

- “atof() — Convert Character String to Float” on page 71
- “atoi() — Convert Character String to Integer” on page 72
- “atol() — Convert Character String to Long or Long Long Integer” on page 74
- “strtol() - strtol() - strtold() — Convert Character String to Double, Float, and Long Double” on page 422
- “strtod32() - strtod64() - strtod128() — Convert Character String to Decimal Floating-Point” on page 425
- “strtoul() — strtoull() — Convert Character String to Unsigned Long and Unsigned Long Long Integer” on page 432
- “wcstol() – wcstoll() — Convert Wide Character String to Long and Long Long Integer” on page 515
- “<stdlib.h>” on page 15

### strtoul() — strtoull() — Convert Character String to Unsigned Long and Unsigned Long Long Integer

#### Format (strtoul())

```c
#include <stdlib.h>
unsigned long int strtoul(const char *nptr, char **endptr, int base);
```

#### Format (strtoull())

```c
#include <stdlib.h>
unsigned long long int strtoull(char *string, char **endptr, int base);
```
Language Level
ANSI

Threadsafe
Yes

Locale Sensitive
The behavior of these functions might be affected by the LC_CTYPE category of the current locale. For more information, see “Understanding CCSIDs and Locales” on page 565.

Description
The `strtoul()` function converts a character string to an unsigned long integer value. The parameter `nptr` points to a sequence of characters that can be interpreted as a numeric value of type unsigned long int.

The `strtoull()` function converts a character string to an unsigned long long integer value. The parameter `nptr` points to a sequence of characters that can be interpreted as a numeric value of type unsigned long long int.

When you use these functions, the `nptr` parameter should point to a string with the following form:

```
whitespace + 0 0x 0X digits
```

If the `base` parameter is a value between 2 and 36, the subject sequence’s expected form is a sequence of letters and digits representing an integer whose radix is specified by the base parameter. This sequence is optionally preceded by a positive (+) or negative (-) sign. Letters from a to z inclusive (either upper or lower case) are ascribed the values 10 to 35. Only letters whose ascribed values are less than that of the base parameter are permitted. If the `base` parameter has a value of 16 the characters `0x` or `0X` optionally precede the sequence of letters and digits, following the positive (+) or negative (-) sign, if present.

If the value of the `base` parameter is 0, the string determines the `base`. After an optional leading sign a leading 0 indicates octal conversion, a leading 0x or 0X indicates hexadecimal conversion, and all other leading characters result in decimal conversion.

These functions scan the string up to the first character that is inconsistent with the base parameter. This character may be the null character (`\0`) at the end of the string. Leading white-space characters are ignored, and an optional sign may precede the digits.

If the value of the `endptr` parameter is not null a pointer, a pointer to the character that ended the scan is stored in the value pointed to by `endptr`. If a value cannot be formed, the value pointed to by `endptr` is set to the `nptr` parameter.

Return Value
If `base` has an invalid value (less than 0, 1, or greater than 36), `errno` is set to EINVAL and 0 is returned. The value pointed to by the `endptr` parameter is set to the value of the `nptr` parameter.

If the value is outside the range of representable values, `errno` is set to ERANGE. The `strtoul()` function will return ULONG_MAX and the `strtoull()` function will return ULONGLONG_MAX.

If no characters are converted, the `strtoull()` function will set `errno` to EINVAL and 0 is returned. The `strtoul()` function will return 0 but will not set `errno` to EINVAL. In both cases the value pointed to by `endptr` is set to the value of the `nptr` parameter. Upon successful completion, both functions return the converted value.
Example
This example converts the string to an unsigned long value. It prints out the converted value and the substring that stopped the conversion.

```c
#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 should be similar to:  *****************
string = 1000e13 e
   strtoul = 8 (base 2)
   Stopped scan at e13 e
*/
```

Related Information
- “atof() — Convert Character String to Float” on page 71
- “atoi() — Convert Character String to Integer” on page 72
- “atol() — Convert Character String to Long or Long Long Integer” on page 74
- “strtol() - strtol() - strtok() — Convert Character String to Long and Long Long Integer” on page 430
- “wcstoul() – wcstoull() — Convert Wide Character String to Unsigned Long and Unsigned Long Long Integer” on page 520
- “<stdlib.h>” on page 15

strxfrm() — Transform String

Format

```c
#include <string.h>
size_t strxfrm(char *string1, const char *string2, size_t count);
```

Language Level
ANSI

Thatsafe
Yes

Locale Sensitive
The behavior of this function might be affected by the LC_CTYPE and LC_COLLATE categories of the current locale. For more information, see “Understanding CCSIDs and Locales” on page 565.
Description

The `strxfrm()` function transforms the string pointed to by `string2` and places the result into the string pointed to by `string1`. The transformation is determined by the program’s current locale. The transformed string is not necessarily readable, but can be used with the `strcmp()` or the `strncmp()` functions.

Return Value

The `strxfrm()` function returns the length of the transformed string, excluding the ending null character. If the returned value is greater than or equal to `count`, the contents of the transformed string are indeterminate.

If `strxfrm()` is unsuccessful, `errno` is changed. The value of `errno` may be set to EINVAL (the `string1` or `string2` arguments contain characters which are not available in the current locale).

Example

This example prompts the user to enter a string of characters, then uses `strxfrm()` to transform the string and return its length.

```c
#include <stdio.h>
#include <string.h>

int main(void)
{
    char *string1, buffer[80];
    int length;

    printf("Type in a string of characters.\n ");
    string1 = gets(buffer);
    length = strxfrm(NULL, string1, 0);
    printf("You would need a %d element array to hold the string\n", length);
    printf("\n\n%Wn\n\ntransformed according\n", string1);
    printf(" to this program’s locale. \n");
}
```

Related Information

- “localeconv() — Retrieve Information from the Environment” on page 202
- “setlocale() — Set Locale” on page 366
- “strcmp() — Compare Strings” on page 388
- “strcoll() — Compare Strings” on page 391
- “strncmp() — Compare Strings” on page 408
- “<string.h>” on page 15

`swprintf()` — Format and Write Wide Characters to Buffer

Format

```c
#include <wchar.h>
int swprintf(wchar_t *wcsbuffer, size_t n, const wchar_t *format, argument-list);
```

Language Level

ANSI

Threading

Yes
**Locale Sensitive**

The behavior of this function might be affected by the LC_CTYPE and LC_NUMERIC categories of the current locale. The behavior might also be affected by the LC_UNI_CTYPE and LC_UNI_NUMERIC categories of the current locale if LOCALETYPE(*LOCALEUCS2) or LOCALETYPE(*LOCALEUTF) is specified on the compilation command. This function is not available when LOCALETYPE(*CLD) is specified on the compilation command. For more information, see “Understanding CCSIDs and Locales” on page 565.

**Wide Character Function**

See “Wide Characters” on page 568 for more information.

**Description**

The `swprintf()` function formats and stores a series of wide characters and values into the wide-character buffer `wcsbuffer`. The `swprintf()` function is equivalent to the `sprintf()` function, except that it operates on wide characters.

The value `n` specifies the maximum number of wide characters to be written, including the ending null character. The `swprintf()` function converts each entry in the argument-list according to the corresponding wide-character format specifier in `format`. The format has the same form and function as the format string for the `printf()` function, with the following exceptions:

- `%c` (without an l prefix) converts a character argument to wchar_t, as if by calling the `mbtowc()` function.
- `%lc` and `%C` copy a wchar_t to wchar_t. %#lc and %#C are equivalent to %lc and %C, respectively.
- `%s` (without an l prefix) converts an array of multibyte characters to an array of wchar_t, as if by calling the `mbstowcs()` function. The array is written up to, but not including, the ending null character, unless the precision specifies a shorter output.
- `%ls` and `%S` copy an array of wchar_t (no conversion). The array is written up to, but not including, the ending NULL character, unless the precision specifies a shorter output. %#ls and %#S are equivalent to %ls and %S, respectively.

Width and precision always are wide characters.

A null wide character is added to the end of the wide characters written; the null wide character is not counted as part of the returned value. If copying takes place between objects that overlap, the behavior is undefined.

**Return Value**

The `swprintf()` function returns the number of wide characters that are written to the output buffer, not counting the ending null wide character or a negative value if an error is encountered. If `n` or more wide characters are requested to be written, a negative value is returned.

The value of `errno` may be set to `EINVAL`, invalid argument.

**Example**

This example uses the `swprintf()` function to format and print several values to buffer.
#include <wchar.h>
#include <stdio.h>

#define BUF_SIZE 100

int main(void)
{
    wchar_t wcsbuf[BUF_SIZE];
    wchar_t wstring[] = L"ABCDE'';
    int     num;

    num = swprintf(wcsbuf, BUF_SIZE, L"%s", "xyz");
    num += swprintf(wcsbuf + num, BUF_SIZE - num, L"%ls", wstring);
    num += swprintf(wcsbuf + num, BUF_SIZE - num, L"%i", 100);
    printf("The array wcsbuf contains: \"%ls\"\n", wcsbuf);
    return 0;
}

/****************************
 The output should be similar to :
 The array wcsbuf contains: "xyzABCDE100"
 ****************************/

Related Information
• “printf() — Print Formatted Characters” on page 251
• “sprintf() — Print Formatted Data to Buffer” on page 379
• “vswprintf() — Format and Write Wide Characters to Buffer” on page 471
• “<wchar.h>” on page 16

swscanf() — Read Wide Character Data

Format

#include <wchar.h>
int swscanf(const wchar_t *buffer, const wchar_t *format, argument-list);

Language Level
ANSI

Threatsafe
Yes

Locale Sensitive
The behavior of this function might be affected by the LC_CTYPE and LC_NUMERIC categories of the current locale. The behavior might also be affected by the LC_UNI_CTYPE and LC_UNI_NUMERIC categories of the current locale if LOCALETYPE(*LOCALEUCS2) or LOCALETYPE(*LOCALEUTF) is specified on the compilation command. This function is not available when LOCALETYPE(*CLD) is specified on the compilation command. For more information, see “Understanding CCSIDs and Locales” on page 565.

Wide Character Function
See “Wide Characters” on page 568 for more information.
Description

The `swscanf()` function is equivalent of the `fwscanf()` function, except that the argument buffer specifies a wide string from which the input is to be obtained, rather than from a stream. Reaching the end of the wide string is equivalent to encountering end-of-file for the `fwscanf()` function.

Return Value

The `swscanf()` function returns the number of fields that were successfully converted and assigned. The return value does not include fields that were read but not assigned. The return value is EOF when the end of the string is encountered before anything is converted.

The value of `errno` may be set `EINVAL`, invalid argument.

Example

This example uses the `swscanf()` function to read various data from the string `ltokenstring`, and then displays that data.

```c
#include <wchar.h>
#include <stdio.h>

wchar_t *ltokenstring = L"15 12 14";
int i;
float fp;
char s[10];
char c;

int main(void)
{
    /* Input various data                                           */
    swscanf(ltokenstring, L"%s %c%d%f", s, &c, &i, &fp);
    /* If there were no space between %s and %c,        */
    /* swscanf would read the first character following */
    /* the string, which is a blank space.                      */
    printf("string = %s
",s);
    printf("character = %c
",c);
    printf("integer = %d
",i);
    printf("floating-point number = %f
",fp);
}
```

Related Information

- “fscanf() — Read Formatted Data” on page 155
- “scanf() — Read Data” on page 358
- “fwscanf() — Read Data from Stream Using Wide Character” on page 169
- “wscanf() — Read Data Using Wide-Character Format String” on page 539
- “sscanf() — Read Data” on page 382
- “sprintf() — Print Formatted Data to Buffer” on page 379
- “<wchar.h>” on page 16

system() — Execute a Command

Format

```c
#include <stdlib.h>
int system(const char *string);
```
Language Level
ANSI

Threadsafe
Yes

However, the CL command processor and all CL commands are NOT threadsafe. Use this function with caution.

Description
The `system()` function passes the given string to the CL command processor for processing.

Return Value
If passed a non-NULL pointer to a string, the `system()` function passes the argument to the CL command processor. The `system()` function returns zero if the command is successful. If passed a NULL pointer to a string, `system()` returns -1, and the command processor is not called. If the command fails, `system()` returns 1. If the `system()` function fails, the global variable `_EXCP_MSGID` in `<stddef.h>` is set with the exception message ID. The exception message ID set within the `_EXCP_MSGID` variable is in job CCSID.

Example
Example that uses `system()`.

```c
#include <stdlib.h>
int main(void)
{
    int result;
    /* A data area is created, displayed and deleted: */
    result = system("CRTDTAARA QTEMP/TEST TYPE(*CHAR) VALUE('Test')");
    result = system("DSPDTAARA TEST");
    result = system("DLTDTAARA TEST");
}
```

Related Information
- “exit() — End Program” on page 113
- “<stdlib.h>” on page 15

**tan() — Calculate Tangent**

Format
```
#include <math.h>
double tan(double x);
```

Language Level
ANSI

Threadsafe
Yes
Description
The tan() function calculates the tangent of x, where x is expressed in radians. If x is too large, a partial loss of significance in the result can occur and sets errno to ERANGE. The value of errno may also be set to EDOM.

Return Value
The tan() function returns the value of the tangent of x.

Example
This example computes x as the tangent of π/4.

```c
#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 should be similar to: ***************
tan( 0.785398 ) is 1.000000
*/
```

Related Information
- “acos() — Calculate Arccosine” on page 62
- “asin() — Calculate Arcsine” on page 66
- “atan() – atan2() — Calculate Arctangent” on page 69
- “cos() — Calculate Cosine” on page 89
- “cosh() — Calculate Hyperbolic Cosine” on page 90
- “sin() — Calculate Sine” on page 376
- “sinh() — Calculate Hyperbolic Sine” on page 377
- “tanh() — Calculate Hyperbolic Tangent” on page 440
- “<math.h>” on page 6

**tanh() — Calculate Hyperbolic Tangent**

Format

```c
#include <math.h>
double tanh(double x);
```

Language Level
ANSI

Threadsafe
Yes
Description
The `tanh()` function calculates the hyperbolic tangent of \( x \), where \( x \) is expressed in radians.

**Return Value**
The `tanh()` function returns the value of the hyperbolic tangent of \( x \). The result of `tanh()` cannot have a range error.

**Example**
This example computes \( x \) as the hyperbolic tangent of \( \pi/4 \).

```c
#include <math.h>
#include <stdio.h>

int main(void)
{
    double pi, x;
    pi = 3.1415926;
    x = tanh(pi/4);
    printf("tanh( %lf ) = %lf
", pi/4, x);
}
```

Related Information
- “`acos()` — Calculate Arcosine” on page 62
- “`asin()` — Calculate Arcsine” on page 66
- “`atan()` — Calculate Arctangent” on page 69
- “`cos()` — Calculate Cosine” on page 89
- “`cosh()` — Calculate Hyperbolic Cosine” on page 90
- “`sin()` — Calculate Sine” on page 376
- “`sinh()` — Calculate Hyperbolic Sine” on page 377
- “`tan()` — Calculate Tangent” on page 439
- “`<math.h>`” on page 6

**time() — Determine Current Time**

**Format**

```c
#include <time.h>
time_t time(time_t *timeptr);
```

**Language Level**
ANSI

**Threadsafe**
Yes
Description

The `time()` function determines the current calendar time, in seconds.

**Note:** Calendar time is the number of seconds that have elapsed since EPOCH, which is 00:00:00, January 1, 1970 Universal Coordinate Time (UTC).

Return Value

The `time()` function returns the current calendar time. The return value is also stored in the location that is given by `timeptr`. If `timeptr` is NULL, the return value is not stored. If the calendar time is not available, the value `(time_t)(-1)` is returned.

Example

This example gets the time and assigns it to `ltime`. The `ctime()` function then converts the number of seconds to the current date and time. This example then prints a message giving the current time.

```c
#include <time.h>
#include <stdio.h>

int main(void)
{
    time_t ltime;
    if (time(&ltime) == -1)
    {
        printf("Calendar time not available.\n");
        exit(1);
    }
    printf("The time is %.2s\n", ctime(&ltime));
}

/******************  Output should be similar to:  ****************
The time is Mon Mar 22 19:01:41 2004
*/
```

Related Information

- “asctime() — Convert Time to Character String” on page 63
- “<time.h>” on page 16
- “asctime_r() — Convert Time to Character String (Restartable)” on page 65
- “ctime() — Convert Time to Character String” on page 96
- “ctime64() — Convert Time to Character String” on page 98
- “ctime64_r() — Convert Time to Character String (Restartable)” on page 101
- “ctime_r() — Convert Time to Character String (Restartable)” on page 100
- “gmtime() — Convert Time” on page 183
- “gmtime64() — Convert Time” on page 185
- “gmtime64_r() — Convert Time (Restartable)” on page 189
- “gmtime_r() — Convert Time (Restartable)” on page 187
- “localtime() — Convert Time” on page 207
- “localtime64() — Convert Time” on page 208
- “localtime64_r() — Convert Time (Restartable)” on page 211
- “localtime_r() — Convert Time (Restartable)” on page 210
- “mktime() — Convert Local Time” on page 241
- “mktime64() — Convert Local Time” on page 242
- “time64() — Determine Current Time” on page 443
- “<time.h>” on page 16
time64() — Determine Current Time

Format

```c
#include <time.h>

time64_t time64(time64_t *timeptr);
```

Language Level

ANSI

Threadsafe

Yes

Description

The `time64()` function determines the current calendar time, in seconds.

**Note:** Calendar time is the number of seconds that have elapsed since EPOCH, which is 00:00:00, January 1, 1970 Universal Coordinate Time (UTC).

Return Value

The `time64()` function returns the current calendar time. The return value is also stored in the location that is given by `timeptr`. If `timeptr` is NULL, the return value is not stored. If the calendar time is not available, the value (time64_t)(-1) is returned.

Example

This example gets the time and assigns it to `ltime`. The `ctime64()` function then converts the number of seconds to the current date and time. This example then prints a message giving the current time.

```c
#include <time.h>
#include <stdio.h>

int main(void)
{
    time64_t ltime;
    if (time64(&ltime) == -1)
    {
        printf("Calendar time not available.\n");
        exit(1);
    }
    printf("The time is %s", ctime64(&ltime));
}
```

/******************  Output should be similar to:  ****************
The time is Mon Mar 22 19:01:41 2004
*/

Related Information

- “asctime() — Convert Time to Character String” on page 63
- “asctime_r() — Convert Time to Character String (Restartable)” on page 65
- “ctime() — Convert Time to Character String” on page 96
- “ctime64() — Convert Time to Character String” on page 98
- “ctime_r() — Convert Time to Character String (Restartable)” on page 100
- “ctime() — Convert Time to Character String” on page 96
tmpfile() — Create Temporary File

Format

```
#include <stdio.h>
FILE *tmpfile(void);
```

Language Level

ANSI

Threadsafe

Yes

Description

The tmpfile() function creates a temporary binary file. The file is automatically removed when it is closed or when the program is ended.

The tmpfile() function opens the temporary file in wb+ mode.

Return Value

The tmpfile() function returns a stream pointer, if successful. If it cannot open the file, it returns a NULL pointer. On normal end (exit()), these temporary files are removed.

On the Data Management system, the tmpfile() function creates a new file that is named QTEMP/QACXxxxxxx. If you specify the SYSIFCOPT(*IFSIO) option on the compilation command, the tmpfile() function creates a new file that is named /tmp/QACXaaaaaa. At the end of the job, the file that is created with the filename from the tmpfile() function is discarded. You can use the remove() function to remove files.

Example

This example creates a temporary file, and if successful, writes tmpstring to it. At program end, the file is removed.
```c
#include <stdio.h>

FILE *stream;
char tmpstring[ ] = "This is the string to be temporarily written";

int main(void)
{
    if ((stream = tmpfile( )) == NULL)
        perror("Cannot make a temporary file");
    else
        fprintf(stream, "%s", tmpstring);
}
```

---

**Related Information**

- “fopen() — Open Files” on page 134
- “<stdio.h>” on page 13

---

### tmpnam() — Produce Temporary File Name

**Format**

```c
#include <stdio.h>
char *tmpnam(char *string);
```

**Language Level**

ANSI

**Threadsafe**

Yes

However, using tmpnam(NULL) is NOT threadsafe.

**Description**

The `tmpnam()` function 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 NULL, the `tmpnam()` function leaves the result in an internal static buffer. Any subsequent calls destroy this value. If `string` is not NULL, it must point to an array of at least `L_tmpnam` bytes. The value of `L_tmpnam` is defined in `<stdio.h>`.

The `tmpnam()` function produces a different name each time it is called within an activation group up to at least `TMP_MAX` names. For ILE C, `TMP_MAX` is 32 767. This is a theoretical limit; the actual number of files that can be opened at the same time depends on the available space in the system.

**Return Value**

The `tmpnam()` function returns a pointer to the name. If it cannot create a unique name then it returns NULL.

**Example**

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.
", name1);
    else
        printf("Cannot create a unique file name\n");
}

Related Information
• “fopen() — Open Files” on page 134
• “remove() — Delete File” on page 301
• “<stdio.h>” on page 13

toascii() — Convert Character to Character Representable by ASCII

Format
#include <ctype.h>
int toascii(int c);

Language Level
XPG4

Threadsafe
Yes

Locale Sensitive
The behavior of this function might be affected by the LC_CTYPE category of the current locale. This function is not available when LOCALETYPE("CLD) is specified on the compilation command. For more information, see “Understanding CCSIDs and Locales” on page 565.

Description
The toascii() function determines to what character c would be mapped to in a 7–bit US-ASCII locale and returns the corresponding character encoding in the current locale.

Return Value
The toascii() function maps the character c according to a 7–bit US-ASCII locale and returns the corresponding character encoding in the current locale.

Example
This example prints encodings of the 7–bit US-ASCII characters 0x7c to 0x82 are mapped to by toascii().
#include <stdio.h>
#include <ctype.h>

void main(void)
{
    int ch;
    for (ch=0x7c; ch<=0x82; ch++) {
        printf("toascii(%#04x) = %c\n", ch, toascii(ch));
    }
}

/*****************And the output should be:******************************/
toascii(0x7c) = @
toascii(0x7d) = '
toascii(0x7e) = =
toascii(0x7f) = "
toascii(0x80) = X
toascii(0x81) = a
toascii(0x82) = b
**************************************************************************/

Related Information
- “isascii() — Test for Character Representable as ASCII Value” on page 193
- “<ctype.h>” on page 1

**tolower() – toupper() — Convert Character Case**

**Format**

```c
#include <ctype.h>
int tolower(int C);
int toupper(int c);
```

**Language Level**

ANSI

**Threadsafe**

Yes

**Locale Sensitive**

The behavior of these functions might be affected by the LC_CTYPE category of the current locale. For more information, see “Understanding CCSIDs and Locales” on page 565.

**Description**

The `tolower()` function converts the uppercase letter `C` to the corresponding lowercase letter.
The `toupper()` function converts the lowercase letter `c` to the corresponding uppercase letter.

**Return Value**

Both functions return the converted character. If the character `c` does not have a corresponding lowercase or uppercase character, the functions return `c` unchanged.

**Example**

This example uses the `toupper()` and `tolower()` functions to change characters between code 0 and code 7f.
```c
#include <stdio.h>
#include <ctype.h>

int main(void)
{
    int ch;
    for (ch = 0; ch <= 0x7f; ch++)
    {
        printf("toupper=%#04x\n", toupper(ch));
        printf("tolower=%#04x\n", tolower(ch));
        putchar('n');
    }
}
```

**Related Information**

- “isalnum() – isxdigit() — Test Integer Value” on page 192
- “tolower() – towupper() — Convert Wide Character Case” on page 449
- “<ctype.h>” on page 1

## towctrans() — Translate Wide Character

### Format

```c
#include <wctype.h>
wint_t towctrans(wint_t wc, wctrans_t desc);
```

### Language Level

ANSI

### Threading Safe

Yes

### Locale Sensitive

The behavior of this function might be affected by the LC_CTYPE category of the current locale if LOCALETYPE(*LOCALE) is specified on the compilation command. It might also be affected by the LC_UNI_CTYPE category of the current locale if either the LOCALETYPE(*LOCALEUCS2) or LOCALETYPE(*LOCALEUTF) option is specified on the compilation command. This function is not available when LOCALETYPE(*CLD) is specified on the compilation command. For more information, see “Understanding CCSIDs and Locales” on page 565.

### Wide Character Function

See “Wide Characters” on page 568 for more information.

### Description

The `towctrans()` function maps the wide character `wc` using the mapping that is described by `desc`. A `towctrans(wc, wctrans("tolower"))` behaves in the same way as the call to the wide-character, case-mapping function `tolower()`. A `towctrans(wc, wctrans("toupper"))` behaves in the same way as the call to the wide-character, case-mapping function `towupper()`.
Return Value

The towctrans() function returns the mapped value of wc using the mapping that is described by desc.

Example

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <wchar.h>
#include <wctype.h>

int main()
{
    char *alpha = "abcdefghijklmnopqrstuvwxyz";
    char *tocase[2] = {"toupper", "tolower"};
    wchar_t *wcalpha;
    int i, j;
    size_t alphalen;

    alphalen = strlen(alpha)+1;
    wcalpha = (wchar_t *)malloc(sizeof(wchar_t)*alphalen);
    mbstowcs(wcalpha, alpha, 2*alphalen);

    for (i=0; i<2; ++i) {
        printf("Input string: %ls\n", wcalpha);
        for (j=0; j<strlen(alpha); ++j) {
            wcalpha[j] = (wchar_t)towctrans((wint_t)wcalpha[j], wctrans(tocase[i]));
        }
        printf("Output string: %ls\n", wcalpha);
        printf("\n");
    }
    return 0;
}

/**************************** Output should be similar to: ****************************

Input string: abcdefghijklmnopqrstuvwxyz
Output string: ABCDEFGHIJKLMNOPQRSTUVWXYZ
Input string: ABCDEFGHIJKLMNOPQRSTUVWXYZ
Output string: abcdefghijklmnopqrstuvwxyz

*******************************************************************/
```

Related Information

- “wctrans() — Get Handle for Character Mapping” on page 528
- “<wchar.h>” on page 16

tolower() – toupper() — Convert Wide Character Case

Format

```c
#include <wctype.h>
wint_t towlower(wint_t wc);
wint_t towupper(wint_t wc);
```

Language Level

ANSI

Threading

Yes
Locale Sensitive

The behavior of these functions might be affected by the LC_CTYPE category of the current locale if LOCALETYPE(*LOCALE) is specified on the compilation command. The behavior of these functions might also be affected by the LC_UNI_CTYPE category of the current locale if either the LOCALETYPE(*LOCALEUCS2) or LOCALETYPE(*LOCALEUTF) option is specified on the compilation command. These functions are not available when LOCALETYPE(*CLD) is specified on the compilation command. For more information, see “Understanding CCSIDs and Locales” on page 565.

Wide Character Function

See “Wide Characters” on page 568 for more information.

Description

The towupper() function converts the lowercase character wc to the corresponding uppercase letter. The towlower() function converts the uppercase character wc to the corresponding lowercase letter.

Return 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, the argument is returned unchanged.

Example

This example uses towlower() and towupper() to convert characters between 0 and 0x7f.

```c
#include <wctype.h>
#include <stdio.h>

int main(void)
{
    wint_t w_ch;
    for (w_ch = 0; w_ch <= 0xff; w_ch++) {
        printf ("towupper : %#04x %#04x, ", w_ch, towupper(w_ch));
        printf ("towlower : %#04x %#04x\n", w_ch, towlower(w_ch));
    }
    return 0;
}

/*===============================================================================================*/
The output should be similar to:
:
: towupper : 0xc1 0xc1, towlower : 0xc1 0x81
: towupper : 0xc2 0xc2, towlower : 0xc2 0x82
: towupper : 0xc3 0xc3, towlower : 0xc3 0x83
: towupper : 0xc4 0xc4, towlower : 0xc4 0x84
: towupper : 0xc5 0xc5, towlower : 0xc5 0x85
:
: towupper : 0x81 0xc1, towlower : 0x81 0x81
: towupper : 0x82 0xc2, towlower : 0x82 0x82
: towupper : 0x83 0xc3, towlower : 0x83 0x83
: towupper : 0x84 0xc4, towlower : 0x84 0x84
: towupper : 0x85 0xc5, towlower : 0x85 0x85
:
:===============================================================================================*/

Related Information

• “iswalnum() – iswxdigit() — Test Wide Integer Value” on page 194
• “tolower() – toupper() — Convert Character Case” on page 447
• “<wctype.h>” on page 17
_ultoa() — Convert Unsigned Long Integer to String

Format

```c
#include <stdlib.h>
char *_ultoa(unsigned long value, char *string, int radix);
```

Note: The _ultoa function is supported only for C++, not for C.

Language Level

Extension

Threadsafe

Yes

Description

_ultoa converts the digits of the given unsigned long `value` to a character string that ends with a null character and stores the result in `string`. The `radix` argument specifies the base of `value`; it must be in the range 2 to 36.

Note: The space allocated for `string` must be large enough to hold the returned string. The function can return up to 33 bytes including the null character (\0).

Return Value

_ultoa returns a pointer to `string`. There is no error return value.

When the string argument is NULL or the `radix` is outside the range 2 to 36, errno will be set to EINVAL.

Example

This example converts the integer value 255 to a decimal, binary, and hexadecimal representation.

```c
#include <stdio.h>
#include <stdlib.h>
int main(void)
{
    char buffer[35];
    char *p;
    p = _ultoa(255UL, buffer, 10);
    printf("The result of _ultoa(255) with radix of 10 is %s\n", p);
    p = _ultoa(255UL, buffer, 2);
    printf("The result of _ultoa(255) with radix of 2\n is %s\n", p);
    p = _ultoa(255UL, buffer, 16);
    printf("The result of _ultoa(255) with radix of 16 is %s\n", p);
    return 0;
}
```

The output should be:

```
The result of _ultoa(255) with radix of 10 is 255
The result of _ultoa(255) with radix of 2
 is 11111111
The result of _ultoa(255) with radix of 16 is ff
```

Related Information

- “_gcvt() — Convert Floating-Point to String” on page 173
- “_itoa() — Convert Integer to String” on page 198
- “_ltoa() — Convert Long Integer to String” on page 214
ungetc() — Push Character onto Input Stream

Format

```c
#include <stdio.h>
int ungetc(int c, FILE *stream);
```

Language Level

ANSI

Threadsafe

Yes

Description

The `ungetc()` function pushes the unsigned character `c` back onto the given input `stream`. However, only one consecutive character is guaranteed to be pushed back onto the input stream if you call `ungetc()` consecutively. The `stream` must be open for reading. A subsequent read operation on the `stream` starts with `c`. The character `c` cannot be the EOF character.

Characters placed on the stream by `ungetc()` will be erased if `fseek()`, `fsetpos()`, `rewind()`, or `fflush()` is called before the character is read from the `stream`.

Return Value

The `ungetc()` function returns the integer argument `c` converted to an unsigned char, or EOF if `c` cannot be pushed back.

The value of `errno` may be set to:

<table>
<thead>
<tr>
<th>Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>ENOTREAD</td>
<td>The file is not open for read operations.</td>
</tr>
<tr>
<td>EIOERROR</td>
<td>A non-recoverable I/O error occurred.</td>
</tr>
<tr>
<td>EIORECERR</td>
<td>A recoverable I/O error occurred.</td>
</tr>
</tbody>
</table>

The `ungetc()` function is not supported for files opened with `type=record`.

Example

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 non-digit character appears before the end of the file, `ungetc()` replaces it in the input stream so that later input functions can process it.
```c
#include <stdio.h>
#include <ctype.h>

int main(void) {
    FILE *stream;
    int ch;
    unsigned int result = 0;
    while ((ch = getc(stream)) != EOF && isdigit(ch))
        result = result * 10 + ch - '0';
    if (ch != EOF)
        ungetc(ch, stream); /* Put the nondigit character back */
    printf("The result is: %d\n", result);
    if ((ch = getc(stream)) != EOF)
        printf("The character is: %c\n", ch);
}
```

**Related Information**
- “getc() – getchar() — Read a Character” on page 174
- “fflush() — Write Buffer to File” on page 121
- “fseek() – fseeko() — Reposition File Position” on page 157
- “fsetpos() — Set File Position” on page 159
- “putc() – putchar() — Write a Character” on page 263
- “rewind() — Adjust Current File Position” on page 303
- “<stdio.h>” on page 13

## ungetwc() — Push Wide Character onto Input Stream

**Format**
```
#include <wchar.h>
#include <stdio.h>

wint_t ungetwc(wint_t wc, FILE *stream);
```

**Language Level**
ANSI

**Threadsafe**
Yes

**Locale Sensitive**
The behavior of this function might be affected by the LC_CTYPE category of the current locale. The behavior might also be affected by the LC_UNI_CTYPE category of the current locale if LOCALETYPE(*LOCALEUCS2) or LOCALETYPE(*LOCALEUTF) is specified on the compilation command. This function is not available when LOCALETYPE(*CLD) is specified on the compilation command. For more information, see “Understanding CCSIDs and Locales” on page 565.

**Integrated File System Interface**
This function is not available when SYSIFCOPT(*NOIFSIO) is specified on the compilation command.

**Wide Character Function**
See “Wide Characters” on page 568 for more information.
**Description**

The `ungetwc()` function pushes the wide character `wc` back onto the input 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 (on the 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` is 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. However, only one consecutive wide character is guaranteed to be pushed back onto the input stream if you call `ungetwc()` consecutively.

For a text stream, the file position indicator is backed up by one wide character. This affects the `ftell()`, `fflush()`, `fseek()` (with SEEK_CUR), and `fgetpos()` function. 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 the `ftell()`, `fseek()` (with SEEK_CUR), `fgetpos()`, and `fflush()` function.

**Return Value**

The `ungetwc()` function returns the wide character pushed back after conversion, or WEOF if the operation fails.

**Example**

```c
#include <wchar.h>
#include <wctype.h>
#include <stdio.h>
#include <stdlib.h>

int main(void) {
    FILE         *stream;
    wint_t       wc,
    wc2;

    unsigned int result = 0;
    if (NULL == (stream = fopen("ungetwc.dat", "r+"))) {
        printf("Unable to open file.\n");
        exit(EXIT_FAILURE);
    }

    while (WEOF != (wc = fgetwc(stream)) && iswdigit(wc))
        result = result * 10 + wc - L'0';

    if (WEOF != wc)
        ungetwc(wc, stream);    /* Push the nondigit wide character back */

    /* get the pushed back character */
    if (WEOF != (wc2 = fgetwc(stream))) {
        if (wc != wc2) {
            printf("Subsequent fgetwc does not get the pushed back character.\n");
            exit(EXIT_FAILURE);
        }
        printf("The digits read are '%i'\n" , result, wc2);
    }

    return 0;
}
```

Assuming the file `ungetwc.dat` contains:

12345ABCDE67890XYZ

The output should be similar to:

The digits read are '12345'
The character being pushed back is 'A'

------------------------------------------
va_arg() – va_copy() – va_end() – va_start() — Handle Variable Argument List

Format

```c
#include <stdarg.h>

var_type va_arg(va_list arg_ptr, var_type);
void va_copy(va_list dest, va_list src);
void va_end(va_list arg_ptr);
void va_start(va_list arg_ptr, variable_name);
```

Language Level

ANSI

Thesafes

Yes

Description

The va_arg(), va_copy(), va_end(), and va_start() functions 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.

va_start() initializes the arg_ptr pointer for subsequent calls to va_arg(), va_copy() and va_end().

The argument variable_name is the identifier of the rightmost named parameter in the parameter list (preceding,...). Use va_start() before va_arg(). Corresponding va_start() and va_end() macros must be in the same function.

va_copy() initializes dest as a copy of src, as if va_start() had been applied to dest followed by the same sequence of uses of va_arg() as had previously been used to reach the present state of src. Neither va_copy() nor va_start() shall be called to reinitialize dest without an intervening call to va_end() for the same dest.

va_arg() function 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() function can retrieve arguments from the list any number of times within the function. The var_type argument must be one of int, long, decimal, double, struct, union, or pointer, or a typedef of one of these types.

The va_end() function is needed to indicate the end of parameter scanning. Each call of va_start() and va_copy() must be matched by a corresponding call to va_end() in the same function.

Because it is not always possible for the called function to determine how many arguments there are, the calling function should communicate the number of arguments to the called function. To determine the number of arguments, a function can use a null pointer to signal the end of the list or pass the count of the
optional arguments as one of the required arguments. The `printf()` function, for instance, can tell how many arguments there are through the `format-string` argument.

**Return Value**

The `va_arg()` function returns the current argument. The `va_copy()`, `va_end()` and `va_start()` functions do not return a value.

**Example**

This example passes a variable number of arguments to a function which prints each argument twice.

```c
#include <stdio.h>
#include <stdarg.h>

int vout(int max, ...);

int main(void)
{
    vout(2, "Sat", "Sun");
    printf("\n");
    vout(3, "Mon", "Tues", "Wed");
}

int vout(int max, ...)
{
    va_list arg_ptr;
    va_list args_copy;
    int args;
    char *day;
    va_start(arg_ptr, max);
    va_copy(args_copy, arg_ptr);
    args = 0;
    while(args < max)
    {
        day = va_arg(arg_ptr, char *);
        printf("Day: %s\n", day);
        args++;
    }
    va_end(arg_ptr);

    args = 0;
    while(args < max)
    {
        day = va_arg(args_copy, char *);
        printf("Day: %s\n", day);
        args++;
    }
    va_end(args_copy);
}

/******************  Output should be similar to:  *****************/
Day: Sat
Day: Sun
Day: Sat
Day: Sun

Day: Mon
Day: Tues
Day: Wed
Day: Mon
Day: Tues
Day: Wed
*/
```

**Related Information**

- “`vfprintf()` — Print Argument Data to Stream” on page 457
- “`vprintf()` — Print Argument Data” on page 464
- “vfprintf() — Format Argument Data as Wide Characters and Write to a Stream” on page 460
- “`vsprintf()` — Print Argument Data to Buffer” on page 468
- “`<stdarg.h>`” on page 11
vfprintf() — Print Argument Data to Stream

Format

```c
#include <stdarg.h>
#include <stdio.h>
int vfprintf(FILE *stream, const char *format, va_list arg_ptr);
```

Language Level

ANSI

Threadsafe

Yes

Locale Sensitive

The behavior of this function might be affected by the LCCTYPE and LCNUMERIC categories of the current locale. The behavior might also be affected by the LC_CTYPE category of the current locale if LOCALETYPE(*LOCALEUCS2) or LOCALETYPE(*LOCALEUTF) is specified on the compilation command. This function is not available when LOCALETYPE(*CLD) is specified on the compilation command. For more information, see “Understanding CCSIDs and Locales” on page 565.

Description

The `vfprintf()` function formats and writes a series of characters and values to the output stream. The `vfprintf()` function works just like the `fprintf()` function, 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, the `fprintf()` function can have a list of arguments, but the number of arguments in that list is fixed when you compile the program.

The `vfprintf()` function converts each entry in the argument list according to the corresponding format specifier in `format`. The `format` has the same form and function as the format string for the `printf()` function.

Return Value

If successful, `vfprintf()` returns the number of bytes written to `stream`. If an error occurs, the function returns a negative value.

Example

This example prints out a variable number of strings to the file `myfile`.

```c
#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("mylib/myfile", "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 should be similar to:  ****************
Sat  Sun  Mon
*/

Related Information
• “fprintf() — Write Formatted Data to a Stream” on page 141
• “printf() — Print Formatted Characters” on page 251
• “va_arg() – va_copy() – va_end() – va_start() — Handle Variable Argument List” on page 455
• “vfprintf() — Print Argument Data” on page 464
• “vsprintf() — Print Argument Data to Buffer” on page 468
• “vwprintf() — Format Argument Data as Wide Characters and Print ” on page 475
• “<stdarg.h>” on page 11
• “<stdio.h>” on page 13

vfscanf() — Read Formatted Data

Format
```c
#include <stdarg.h>
#include <stdio.h>

int vfscanf(FILE *stream, const char *format, va_list arg_ptr);
```
This function is not available when LOCALETYPEx(CLD) is specified on the compilation command. For more information, see “Understanding CCSIDs and Locales” on page 565.

Description

The \texttt{vfscanf()} function reads data from a stream into locations specified by a variable number of arguments. The \texttt{vfscanf()} function works just like the \texttt{fscanf()} function, except that \texttt{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 \texttt{va_start} for each call. In contrast, the \texttt{fscanf()} function can have a list of arguments, but the number of arguments in that list is fixed when you compile the program.

Each argument must be a pointer to a variable with a type that corresponds to a type specifier in format-string. The \texttt{format} has the same form and function as the format string for the \texttt{scanf()} function.

Return Value

The \texttt{vfscanf()} function returns the number of fields that were successfully converted and assigned. The return value does not include fields that were read but not assigned. The return value is EOF for an attempt to read at end-of-file if no conversion was performed. A return value of 0 means that no fields were assigned.

Example

This example opens the file \textit{myfile} for input, and then scans this file for a string, a long integer value, and a floating-point value.

```c
#include <stdio.h>
#include <stdarg.h>

int vread(FILE *stream, char *fmt, ...) {
    int rc;
    va_list arg_ptr;
    va_start(arg_ptr, fmt);
    rc = vfscanf(stream, fmt, arg_ptr);
    va_end(arg_ptr);
    return(rc);
}

#define MAX_LEN 80
int main(void) {
    FILE *stream;
    long l;
    float fp;
    char s[MAX_LEN + 1];
    char c;
    stream = fopen("mylib/myfile", "r");
    /* Put in various data. */
    vread(stream, "%s", &s[0]);
    vread(stream, "%ld", &l);
    vread(stream, "%c", &c);
    vread(stream, "%f", &fp);
    printf("string = %s\n", s);
    printf("long double = %ld\n", l);
    printf("char = %c\n", c);
    printf("float = %f\n", fp);
}

/******************** If myfile contains ********************
**************** abcdefghijklmnopqrstuvwxyz 343.2 ********************
******************** expected output is: ********************
string = abcdefghijklmnopqrstuvwxyz
long double = 343
char = .
float = 2.000000
/****
vfwprintf() — Format Argument Data as Wide Characters and Write to a Stream

Format
```
#include <stdarg.h>
#include <stdio.h>
#include <wchar.h>
int vfwprintf(FILE *stream, const wchar_t *format, va_list arg);
```

Language Level
ANSI

Threadsafe
Yes

Locale Sensitive
The behavior of this function might be affected by the LC_CTYPE and LC_NUMERIC categories of the current locale. The behavior might also be affected by the LC_UNI_CTYPE and LC_UNI_NUMERIC categories of the current locale if LOCALETYPE(*LOCALEUCS2) or LOCALETYPE(*LOCALEUTF) is specified on the compilation command. This function is not available when LOCALETYPE(*CLD) is specified on the compilation command. For more information, see “Understanding CCSIDs and Locales” on page 565.

Integrated File System Interface
This function is not available when SYSIFCOPT(*NOIFSIO) is specified on the compilation command.

Wide Character Function
See “Wide Characters” on page 568 for more information.

Description
The vfwprintf() function is equivalent to the fwprintf() function, except that the variable argument list is replaced by arg, which the va_start macro (and possibly subsequent va_arg calls) will have initialized. The vfwprintf() function does not invoke the va_end macro.

Because the functions vfwprintf(), vswprintf(), and vwprintf() invoke the va_arg macro, the value of arg after the return is unspecified.

Return Value
The vfwprintf() function returns the number of wide characters that are written to the output buffer, not counting the ending null wide character or a negative value if an error was encountered. If n or more wide characters are requested to be written, a negative value is returned.
Example

This example prints the wide character a to a file. The printing is done from the vout() function, which takes a variable number of arguments and uses vwprintf() to print them to a file.

```c
#include <wchar.h>
#include <stdarg.h>
#include <locale.h>

void vout (FILE *stream, wchar_t *fmt, ...);

const char ifs_path [] = "/tmp/myfile";

int main(void) {
    FILE *stream;
    wchar_t format [] = L"%lc";
    setlocale(LC_ALL, "POSIX");
    if ((stream = fopen (ifs_path, "w")) == NULL) {
        printf("Could not open file.\n");
        return (-1);
    }
    vout (stream, format, L'a');
    fclose (stream);
    
    /*******************************************************************************
     The contents of output file /tmp/myfile should be a wide char 'a' which in the "POSIX" locale is '0081'x.
     */
    return (0);
}

void vout (FILE *stream, wchar_t *fmt, ...)
{
    va_list arg_ptr;
    va_start (arg_ptr, fmt);
    vwprintf (stream, fmt, arg_ptr);
    va_end (arg_ptr);
}
```

Related Information

- “printf() — Print Formatted Characters” on page 251
- “fprintf() — Write Formatted Data to a Stream” on page 141
- “vfprintf() — Print Argument Data to Stream” on page 457
- “vprintf() — Print Argument Data” on page 464
- “btowc() — Convert Single Byte to Wide Character” on page 78
- “mbrtowc() — Convert a Multibyte Character to a Wide Character (Restartable)” on page 223
- “fwprintf() — Format Data as Wide Characters and Write to a Stream” on page 165
- “vswprintf() — Format and Write Wide Characters to Buffer” on page 471
- “vwprintf() — Format Argument Data as Wide Characters and Print ” on page 475
- “<stdarg.h>” on page 11
- “<stdio.h>” on page 13
- “<wchar.h>” on page 16
vfwscanf() — Read Formatted Wide Character Data

Format

```c
#include <stdarg.h>
#include <stdio.h>
int vfwscanf(FILE *stream, const wchar_t *format, va_list arg_ptr);
```

Language Level

ANSI

Threatsafe

Yes

Locale Sensitive

The behavior of this function might be affected by the LC_CTYPE and LC_NUMERIC categories of the current locale. The behavior might also be affected by the LC_UNI_CTYPE and LC_UNI_NUMERIC categories of the current locale if LOCALETYPE(*LOCALEUCS2) or LOCALETYPE(*LOCALEUTF) is specified on the compilation command. This function is not available when LOCALETYPE(*CLD) is specified on the compilation command. For more information, see “Understanding CCSIDs and Locales” on page 565.

Integrated File System Interface

See “Wide Characters” on page 568 for more information.

Wide Character Function

This function is not available when SYSIFCOPT(*NOIFSIO) is specified on the compilation command.

Description

The `vfwscanf()` function reads wide data from a stream into locations specified by a variable number of arguments. The `vfwscanf()` function works just like the `fwscanf()` function, 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, the `fwscanf()` function can have a list of arguments, but the number of arguments in that list is fixed when you compile the program.

Each argument must be a pointer to a variable with a type that corresponds to a type specifier in format-string. The `format` has the same form and function as the format string for the `fwscanf()` function.

Return Value

The `vfwscanf()` function returns the number of fields that were successfully converted and assigned. The return value does not include fields that were read but not assigned. The return value is EOF for an attempt to read at end-of-file if no conversion was performed. A return value of 0 means that no fields were assigned.

Example

This example opens the file `myfile` for input, and then scans this file for a string, a long integer value, and a floating-point value.
#include <stdio.h>
#include <stdarg.h>
#include <wchar.h>

int vread(FILE *stream, wchar_t *fmt, ...)
{
    int rc;
    va_list arg_ptr;
    va_start(arg_ptr, fmt);
    rc = vfwscanf(stream, fmt, arg_ptr);
    va_end(arg_ptr);
    return(rc);
}

#define MAX_LEN 80
int main(void)
{
    FILE *stream;
    long l;
    float fp;
    char s[MAX_LEN + 1];
    char c;
    stream = fopen("mylib/myfile", "r");
    /* Put in various data. */
    vread(stream, L"%s", &s[0]);
    vread(stream, L"%ld", &l);
    vread(stream, L"%c", &c);
    vread(stream, L"%f", &fp);
    printf("string = %s\n", s);
    printf("long double = %ld\n", l);
    printf("char = %c\n", c);
    printf("float = %f\n", fp);
}

/*************** If myfile contains ************************
**************** abcdefghijklmnopqrstuvwxyz 343.2 ***********
********************** expected output is: ********************
string = abcdefghijklmnopqrstuvwxyz
long double = 343
char = .
float = 2.000000
*/

Related Information
• “fscanf() — Read Formatted Data” on page 155
• “fwprintf() — Format Data as Wide Characters and Write to a Stream” on page 165
• “fwscanf() — Read Data from Stream Using Wide Character” on page 169
• “scanf() — Read Data” on page 358
• “sscanf() — Read Data” on page 382
• “swprintf() — Format and Write Wide Characters to Buffer” on page 435
• “swscanf() — Read Wide Character Data” on page 437
• “vfscanf() — Read Formatted Data” on page 458
• “vfwscanf() — Read Formatted Wide Character Data” on page 462
• “vscanf() — Read Formatted Data” on page 465
• “vsscanf() — Read Formatted Data” on page 470
• “vswscanf() — Read Formatted Wide Character Data” on page 473
• “vwscanf() — Read Formatted Wide Character Data” on page 476
• “wprintf() — Format Data as Wide Characters and Print” on page 538
• “wscanf() — Read Data Using Wide-Character Format String” on page 539
• “<wchar.h>” on page 16
vprintf() — Print Argument Data

Format

#include <stdarg.h>
#include <stdio.h>
int vprintf(const char *format, va_list arg_ptr);

Language Level

ANSI

Threadsafe

Yes

Locale Sensitive

The behavior of this function might be affected by the LC_CTYPE and LC_NUMERIC categories of the current locale. The behavior might also be affected by the LC_UNI_CTYPE category of the current locale if LOCALETYPE(*LOCALEUCS2) or LOCALETYPE(*LOCALEUTF) is specified on the compilation command. For more information, see “Understanding CCSIDs and Locales” on page 565.

Description

The vprintf() function formats and prints a series of characters and values to stdout. The vprintf() function works just like the printf() function, 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, the printf() function can have a list of arguments, but the number of arguments in that list is fixed when you compile the program.

The vprintf() function converts each entry in the argument list according to the corresponding format specifier in format. The format has the same form and function as the format string for the printf() function.

Return Value

If successful, the vprintf() function returns the number of bytes written to stdout. If an error occurs, the vprintf() function returns a negative value. The value of errno may be set to ETRUNC.

Example

This example prints out a variable number of strings to stdout.
#include <stdarg.h>
#include <stdio.h>

void vout(char *fmt, ...);
char fmt1[] = "%s %s %s %s %s 
"

int main(void)
{
    FILE *stream;
    stream = fopen("mylib/myfile", "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 should be similar to:  *********************/
Mon Tues Wed Thurs Fri
*/

Related Information
• “printf() — Print Formatted Characters” on page 251
• “va_arg() – va_copy() – va_end() – va_start() — Handle Variable Argument List” on page 455
• “vfprintf() — Print Argument Data to Stream” on page 457
• “vscanf() — Print Argument Data to Buffer” on page 468
• “<stdarg.h>” on page 11
• “<stdio.h>” on page 13

vscanf() — Read Formatted Data

Format
#include <stdarg.h>
#include <stdio.h>
int vscanf(const char *format, va_list arg_ptr);

Language Level
ANSI

Threading Safe
Yes

Locale Sensitive
The behavior of this function might be affected by the LC_CTYPE and LC_NUMERIC categories of the current locale. The behavior might also be affected by the LC_UNI_CTYPE category of the current locale if LOCALETYPE(*LOCALEUCS2) or LOCALETYPE(*LOCALEUTF) is specified on the compilation command. For more information, see “Understanding CCSIDs and Locales” on page 565.
Description
The vscanf() function reads data from stdin into locations specified by a variable number of arguments. The vscanf() function works just like the scanf() function, 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, the scanf() function can have a list of arguments, but the number of arguments in that list is fixed when you compile the program.

Each argument must be a pointer to a variable with a type that corresponds to a type specifier in format-string. The format has the same form and function as the format string for the scanf() function.

Return Value
The vscanf() function returns the number of fields that were successfully converted and assigned. The return value does not include fields that were read but not assigned. The return value is EOF for an attempt to read at end-of-file if no conversion was performed. A return value of 0 means that no fields were assigned.

Example
This example uses the vscanf() function to read an integer, a floating-point value, a character, and a string from stdin and then displays these values.

```c
#include <stdio.h>
#include <stdarg.h>
int vread(char *fmt, ...) {
    int rc;
    va_list arg_ptr;
    va_start(arg_ptr, fmt);
    rc = vscanf(fmt, arg_ptr);
    va_end(arg_ptr);
    return(rc);
}
int main(void) {
    int i, rc;
    float fp;
    char c, s[81];
    printf("Enter an integer, a real number, a character "
           "and a string : 
"n");
    rc = vread("%d %f %c %s", &i, &fp, &c, s);
    if (rc != 4)
        printf("Not all fields are assigned\n");
    else
    {
        printf("integer = %d\n", i);
        printf("real number = %f\n", fp);
        printf("character = %c\n", c);
        printf("string = %s\n", s);
    }
}*************** If input is: 12 2.5 a yes, ***************
*************** then output should be similar to: ***************
Enter an integer, a real number, a character and a string :
integer = 12
real number = 2.500000
character = a
string = yes
*
```

Related Information
- “fscanf() — Read Formatted Data” on page 155
- “fwprintf() — Format Data as Wide Characters and Write to a Stream” on page 165
- “fwscanf() — Read Data from Stream Using Wide Character” on page 169
- “scanf() — Read Data” on page 358
- “sscanf() — Read Data” on page 382
vsnprintf() — Print Argument Data to Buffer

Format

#include <stdarg.h>
#include <stdio.h>
int vsnprintf(char *target-string, size_t n, const char *format, va_list arg_ptr);

Language Level

ANSI

Threaatmse

Yes

Locale Sensitive

The behavior of this function might be affected by the LC_CTYPE and LC_NUMERIC categories of the current locale. The behavior might also be affected by the LC_UNI_CTYPE category of the current locale if LOCALETYPE(*LOCALEUCS2) or LOCALETYPE(*LOCALEUTF) is specified on the compilation command. For more information, see “Understanding CCSIDs and Locales” on page 565.

Description

The vsnprintf() function formats and stores a series of characters and values in the buffer target-string. The vsnprintf() function works just like the snprintf() function, 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 the va_start function for each call. In contrast, the snprintf() function can have a list of arguments, but the number of arguments in that list is fixed when you compile the program.

The vsnprintf() function converts each entry in the argument list according to the corresponding format specifier in format. The format has the same form and function as the format string for the printf() function.

Return Value

The vsnprintf() function returns the number of bytes that are written in the array, not counting the ending null character.
Example
This example assigns a variable number of strings to `string` and prints the resultant string.

```c
#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);
    vsnprintf(string, 8, fmt, arg_ptr);
    va_end(arg_ptr);
}

/******************  Output should be similar to: ****************
The string is: Sat Su
*/
```

Related Information
• “printf() — Print Formatted Characters” on page 251
• “sprintf() — Print Formatted Data to Buffer” on page 379
• “snprintf() — Print Formatted Data to Buffer” on page 378
• “va_arg() – va_copy() – va_end() – va_start() — Handle Variable Argument List” on page 455
• “vfprintf() — Print Argument Data to Stream” on page 457
• “vsprintf() — Print Argument Data to Buffer” on page 468
• “<stdarg.h>” on page 11
• “<stdio.h>” on page 13

vsprintf() — Print Argument Data to Buffer

Format

```c
#include <stdarg.h>
#include <stdio.h>

int vsprintf(char *target-string, const char *format, va_list arg_ptr);
```

Language Level
ANSI

Threading
Yes

Locale Sensitive
The behavior of this function might be affected by the LC_CTYPE and LC_NUMERIC categories of the current locale. The behavior might also be affected by the LC_UNI_CTYPE category of the current locale if
LOCALETYPE(*LOCALEUCS2) or LOCALETYPE(*LOCALEUTF) is specified on the compilation command. For more information, see “Understanding CCSIDs and Locales” on page 565.

Description

The vsprintf() function formats and stores a series of characters and values in the buffer target-string. The vsprintf() function works just like the sprintf() function, 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 the va_start function for each call. In contrast, the sprintf() function can have a list of arguments, but the number of arguments in that list is fixed when you compile the program.

The vsprintf() function converts each entry in the argument list according to the corresponding format specifier in format. The format has the same form and function as the format string for the printf() function.

Return Value

If successful, the vsprintf() function returns the number of bytes written to target-string. If an error occurs, the vsprintf() function returns a negative value.

Example

This example assigns a variable number of strings to string and prints the resultant string.

```c
#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 should be similar to:  **************/
The string is:  Sat  Sun  Mon
/*

Related Information

• “printf() — Print Formatted Characters” on page 251
• “sprintf() — Print Formatted Data to Buffer” on page 379
• “va_arg() – va_copy() – va_end() – va_start() — Handle Variable Argument List” on page 455
• “vfprintf() — Print Argument Data to Stream” on page 457
• “vprintf() — Print Argument Data” on page 464
• “vswprintf() — Format and Write Wide Characters to Buffer” on page 471
• “<stdarg.h>” on page 11
• “<stdio.h>” on page 13
vsscanf() — Read Formatted Data

Format

```c
#include <stdarg.h>
#include <stdio.h>
int vsscanf(const char *buffer, const char *format, va_list arg_ptr);
```

Language Level

ANSI

Threadsafe

Yes

Locale Sensitive

The behavior of this function might be affected by the LC_CTYPE and LC_NUMERIC categories of the current locale. The behavior might also be affected by the LC_UNI_CTYPE category of the current locale if LOCALETYPE("LOCALEUCS2") or LOCALETYPE("LOCALEUTF") is specified on the compilation command. For more information, see “Understanding CCSIDs and Locales” on page 565.

Description

The vsscanf() function reads data from a buffer into locations specified by a variable number of arguments. The vsscanf() function works just like the sscanf() function, 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, the sscanf() function can have a list of arguments, but the number of arguments in that list is fixed when you compile the program.

Each argument must be a pointer to a variable with a type that corresponds to a type specifier in format-string. The format has the same form and function as the format string for the scanf() function.

Return Value

The vsscanf() function returns the number of fields that were successfully converted and assigned. The return value does not include fields that were read but not assigned. The return value is EOF for an attempt to read at end-of-file if no conversion was performed. A return value of 0 means that no fields were assigned.

Example

This example uses vsscanf() to read various data from the string tokenstring and then displays that data.
```c
#include <stdio.h>
#include <stdarg.h>
#include <stddef.h>

int vread(const char *buffer, char *fmt, ...) {
    int rc;
    va_list arg_ptr;
    va_start(arg_ptr, fmt);
    rc = vsscanf(buffer, fmt, arg_ptr);
    va_end(arg_ptr);
    return(rc);
}

int main(void) {
    char *tokenstring = "15 12 14";
    wchar_t * idestring = L"ABC Z";
    wchar_t ws[81];
    wchar_t wc;
    int i;
    float fp;
    char s[81];
    char c;
    /* Input various data */
    /* In the first invocation of vsscanf, the format string is */
    /* "%s %c%d%f". If there were no space between %s and %c, */
    /* vsscanf would read the first character following the */
    /* string, which is a blank space. */
    vread(tokenstring, "%s %c%d%f", s, &c, &i, &fp);
    vread((char *) idestring, "%S %C", ws,&wc);
    /* Display the data */
    printf("string = %s
",s);
    printf("character = %c
",c);
    printf("integer = %d
",i);
    printf("floating-point number = %f
",fp);
    printf("wide-character string = %S
", ws);
    printf("wide-character = %C
", wc);
}

Related Information

- “fscanf() — Read Formatted Data” on page 155
- “fwscanf() — Read Data from Stream Using Wide Character” on page 169
- “scanf() — Read Data” on page 358
- “sscanf() — Read Data” on page 382
- “sprintf() — Print Formatted Data to Buffer” on page 379
- “<stdio.h>” on page 13
- “swscanf() — Read Wide Character Data” on page 437
- “wscanf() — Read Data Using Wide-Character Format String” on page 539

vswprintf() — Format and Write Wide Characters to Buffer

Format
```
#include <stdarg.h>
#include <wchar.h>

int vswprintf(wchar_t *wcsbuffer, size_t n, const wchar_t *format, va_list argptr);
```
Locale Sensitive

The behavior of this function might be affected by the LC_CTYPE and LC_NUMERIC categories of the current locale. It might also be affected by the LC_UNI_CTYPE and LC_UNI_NUMERIC categories of the current locale if LOCALETYPE(*LOCALEUCS2) or LOCALETYPE(*LOCALEUTF) is specified on the compilation command. This function is not available when LOCALETYPE(*CLD) is specified on the compilation command. For more information, see “Understanding CCSIDs and Locales” on page 565.

Wide Character Function

See “Wide Characters” on page 568 for more information.

Description

The vswprintf() function formats and stores a series of wide characters and values in the buffer wcsbuffer. The vswprintf() function works just like the swprintf() function, except that argptr points to a list of wide-character arguments whose number can vary from call to call. These arguments should be initialized by va_start for each call. In contrast, the swprintf() function can have a list of arguments, but the number of arguments in that list are fixed when you compile the program.

The value n specifies the maximum number of wide characters to be written, including the ending null character. The vswprintf() function converts each entry in the argument list according to the corresponding wide-character format specifier in format. The format has the same form and function as the format string for the printf() function, with the following exceptions:

• %c (without an l prefix) converts an integer argument to wchar_t, as if by calling the mbtowc() function.
• %lc converts a wint_t to wchar_t.
• %s (without an l prefix) converts an array of multibyte characters to an array of wchar_t, as if by calling the mbrtowc() function. The array is written up to, but not including, the ending null character, unless the precision specifies a shorter output.
• %ls writes an array of wchar_t. The array is written up to, but not including, the ending null character, unless the precision specifies a shorter output.

A null wide character is added to the end of the wide characters written; the null wide character is not counted as part of the returned value. If copying takes place between objects that overlap, the behavior is undefined.

Return Value

The vswprintf() function returns the number of bytes written in the array, not counting the ending null wide character.

Example

This example creates a function vout() that takes a variable number of wide-character arguments and uses vswprintf() to print them to wcstr.
```c
#include <stdio.h>
#include <stdarg.h>
#include <wchar.h>

wchar_t *format3 = L"%ls  %d  %ls";
wchar_t *format5 = L"%ls  %d  %ls  %d  %ls";

void vout(wchar_t *wcs, size_t n, wchar_t *fmt, ...)
{
    va_list arg_ptr;
    va_start(arg_ptr, fmt);
    vswprintf(wcs, n, fmt, arg_ptr);
    va_end(arg_ptr);
    return;
}

int main(void)
{
    wchar_t wcstr[100];
    vout(wcstr, 100, format3, L"ONE", 2L, L"THREE");
    printf("%ls\n", wcstr);
    vout(wcstr, 100, format5, L"ONE", 2L, L"THREE", 4L, L"FIVE");
    printf("%ls\n", wcstr);
    return 0;
}

/************************************************************
The output should be similar to:
ONE  2  THREE
ONE  2  THREE  4  FIVE
************************************************************/

Related Information
• “swprintf() — Format and Write Wide Characters to Buffer” on page 435
• “vfprintf() — Print Argument Data to Stream” on page 457
• “vprintf() — Print Argument Data” on page 464
• “vsprintf() — Print Argument Data to Buffer” on page 468
• “<stdarg.h>” on page 11
• “<wchar.h>” on page 16

vswscanf() — Read Formatted Wide Character Data

Format
```c
#include <stdarg.h>
#include <wchar.h>

int vswscanf(const wchar_t *buffer, const wchar_t *format, va_list arg_ptr);
```
current locale if LOCALETYPE(*LOCALECUS2) or LOCALETYPE(*LOCALEUTF) is specified on the compilation command. This function is not available when LOCALETYPE(*CLD) is specified on the compilation command. For more information, see “Understanding CCSIDs andLocales” on page 565.

Wide Character Function
See “Wide Characters” on page 568 for more information.

Description
The vswscanf() function reads wide data from a buffer into locations specified by a variable number of arguments. The vswscanf() function works just like the swscanf() function, 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, the swscanf() function can have a list of arguments, but the number of arguments in that list is fixed when you compile the program.

Each argument must be a pointer to a variable with a type that corresponds to a type specifier in format-string. The format has the same form and function as the format string for the swscanf() function.

Return Value
The vswscanf() function returns the number of fields that were successfully converted and assigned. The return value does not include fields that were read but not assigned. The return value is EOF for an attempt to read at end-of-file if no conversion was performed. A return value of 0 means that no fields were assigned.

Example
This example uses the vswscanf() function to read various data from the string tokenstring and then displays that data.

```
#include <stdio.h>
#include <stdarg.h>
#include <wchar.h>

int vread(const wchar_t *buffer, wchar_t *fmt, ...)
{
    int rc;
    va_list arg_ptr;
    va_start(arg_ptr, fmt);
    rc = vswscanf(buffer, fmt, arg_ptr);
    va_end(arg_ptr);
    return(rc);
}

int main(void)
{
    wchar_t *tokenstring = L"15 12 14";
    char s[81];
    char c;
    int i;
    float fp;

    /* Input various data */
    vread(tokenstring, L"%s %c%d%f", s, &c, &i, &fp);

    /* Display the data */
    printf("string = %s
", s);
    printf("character = %c
", c);
    printf("integer = %d
", i);
    printf("floating-point number = %f
", fp);
}
```

/******************** Output should be similar to: ********************
string = 15
character = 1
integer = 2
floating-point number = 14.000000

************************************************************************/

474 IBM i: ILE C/C++ Runtime Library Functions
vwprintf() — Format Argument Data as Wide Characters and Print

**Format**

```c
#include <stdarg.h>
#include <wchar.h>
int vwprintf(const wchar_t *format, va_list arg);
```

**Language Level**

ANSI

**Threadsafe**

Yes

**Locale Sensitive**

The behavior of this function might be affected by the LC_CTYPE and LC_NUMERIC categories of the current locale. It might also be affected by the LC_UNI_CTYPE and LC_UNI_NUMERIC categories of the current locale if LOCALETYPE(*LOCALEUCS2) or LOCALETYPE(*LOCALEUTF) is specified on the compilation command. This function is not available when LOCALETYPE(*CLD) is specified on the compilation command. For more information, see “Understanding CCSIDs and Locales” on page 565.

**Integrated File System Interface**

This function is not available when SYSIFCOPT(*NOIFSIO) is specified on the compilation command.

**Wide Character Function**

See “Wide Characters” on page 568 for more information.

**Description**

The vwprintf() function is equivalent to the wprintf() function, except that the variable argument list is replaced by arg, which the va_start macro (and possibly subsequent va_arg calls) will have initialized. The vwprintf() function does not invoke the va_end macro.

**Return Value**

The vwprintf() function returns the number of wide characters transmitted. If an output error occurred, the vwprintf() returns a negative value.
Example

This example prints the wide character a. The printing is done from the vout() function, which takes a variable number of arguments and uses the vwprintf() function to print them to stdout.

```c
#include <wchar.h>
#include <stdarg.h>
#include <locale.h>

void vout (wchar_t *fmt, ...);

int main(void) {
    FILE *stream;
    wchar_t format[] = L "%lc";
    setlocale(LC_ALL, "POSIX");
    vout (format, L'a');
    return(0);
}

/* A long a is written to stdout, if stdout is written to the screen
   it may get converted back to a single byte 'a'. */
}

void vout (wchar_t *fmt, ...) {
    va_list arg_ptr;
    va_start (arg_ptr, fmt);
    vwprintf (fmt, arg_ptr);
    va_end (arg_ptr);
}
```

Related Information

- “printf() — Print Formatted Characters” on page 251
- “vfprintf() — Print Argument Data to Stream” on page 457
- “vprintf() — Print Argument Data” on page 464
- “btowc() — Convert Single Byte to Wide Character” on page 78
- “mbtowc() — Convert a Multibyte Character to a Wide Character (Restartable)” on page 223
- “fwprintf() — Format Data as Wide Characters and Write to a Stream” on page 165
- “vswprintf() — Format and Write Wide Characters to Buffer” on page 471
- “vfwprintf() — Format Argument Data as Wide Characters and Write to a Stream” on page 460
- “<stdarg.h>” on page 11
- “<wchar.h>” on page 16

vwscanf() — Read Formatted Wide Character Data

**Format**

```c
#include <stdarg.h>
#include <stdio.h>

int vwscanf(const wchar_t *format, va_list arg_ptr);
```

**Language Level**

ANSI

**Threadsafe**

Yes
Locale Sensitive
The behavior of this function might be affected by the LC_CTYPE and LC_NUMERIC categories of the current locale. It might also be affected by the LC_UNI_CTYPE and LC_UNI_NUMERIC categories of the current locale if LOCALETYPE(*LOCALEUCS2) or LOCALETYPE(*LOCALEUTF) is specified on the compilation command. This function is not available when LOCALETYPE(*CLD) is specified on the compilation command. For more information, see “Understanding CCSIDs and Locales” on page 565.

Integrated File System Interface
This function is not available when SYSIFCOPT(*NOIFSIO) is specified on the compilation command.

Wide Character Function
See “Wide Characters” on page 568 for more information.

Description
The vwscanf() function reads data from stdin into locations specified by a variable number of arguments. The vwscanf() function works just like the wscanf() function, 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, the wscanf() function can have a list of arguments, but the number of arguments in that list is fixed when you compile the program.

Each argument must be a pointer to a variable with a type that corresponds to a type specifier in format-string. The format has the same form and function as the format string for the wscanf() function.

Return Value
The vwscanf() function returns the number of fields that were successfully converted and assigned. The return value does not include fields that were read but not assigned. The return value is EOF for an attempt to read at end-of-file if no conversion was performed. A return value of 0 means that no fields were assigned.

Example
This example scans various types of data from stdin.
#include <stdio.h
#include <stdarg.h

int vread(wchar_t *fmt, ...)
{
    int rc;
    va_list arg_ptr;
    va_start(arg_ptr, fmt);
    rc = vwscanf(fmt, arg_ptr);
    va_end(arg_ptr);
    return(rc);
}

int main(void)
{
    int i, rc;
    float fp;
    char c, s[81];
    printf("Enter an integer, a real number, a character "
        "and a string : \n");
    rc = vread(L "%d %f %c %s", &i, &fp, &c, s);
    if (rc != 4)
        printf("Not all fields are assigned\n");
    else
    {
        printf("integer = %d\n", i);
        printf("real number = %f\n", fp);
        printf("character = %c\n", c);
        printf("string = %s\n", s);
    }
}  

/**************************************** If input is: 12 2.5 a yes, ********************
************** then output should be similar to: *************************
Enter an integer, a real number, a character and a string :
integer = 12
real number = 2.500000
character = a
string = yes
*/

**Related Information**

- “fscanf() — Read Formatted Data” on page 155
- “scanf() — Read Data” on page 358
- “sscanf() — Read Data” on page 382
- “swscanf() — Read Wide Character Data” on page 437
- “fwscanf() — Read Data from Stream Using Wide Character” on page 169
- “wscanf() — Read Data Using Wide-Character Format String” on page 539
- “sprintf() — Print Formatted Data to Buffer” on page 379
- “<stdio.h>” on page 13

**wcrtomb() — Convert a Wide Character to a Multibyte Character**
**Restartable**

**Format**

```c
#include <wchar.h>
size_t wcrtomb (char *s, wchar_t wc, mbstate_t *ps);
```

**Language Level**

ANSI
**Threadsafe**
Yes, except when `ps` is NULL.

**Locale Sensitive**
The behavior of this function might be affected by the LC_CTYPE category of the current locale. The behavior might also be affected by the LC_UNI_CTYPE category of the current locale if

- `LOCALETYPE(*LOCALEUCS2)` or `LOCALETYPE(*LOCALEUTF)` is specified on the compilation command.
- `LOCALETYPE(*CLD)` is specified on the compilation command. For more information, see “Understanding CCSIDs and Locales” on page 565.

**Wide Character Function**
See “Wide Characters” on page 568 for more information.

**Description**
This function is the restartable version of the `wctomb()` function.

The `wcrtomb()` function converts a wide character to a multibyte character.

- 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 will be the initial conversion stated.

- 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 `wc` (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 will be stored. If `wc` is a null wide character, the resulting state described will be the initial conversions state.

This function differs from its corresponding internal-state multibyte character function in that it has an extra parameter, `ps` of type pointer to `mbstate_t` that points to an object that can completely describe the current conversion state of the associated multibyte character sequence. If `ps` is NULL, an internal static variable will be used to keep track of the conversion state. Using the internal static variable is not threadsafe.

**Return Value**
- If `s` is a null pointer, the `wcrtomb()` function returns the number of bytes needed to enter the initial shift state. The value returned will not be greater than that of the `MB_CUR_MAX` macro.

- If `s` is not a null pointer, the `wcrtomb()` function returns the number of bytes stored in the array object (including any shift sequences) when `wc` is a valid wide character; otherwise (when `wc` is not a valid wide character), an encoding error occurs, the value of the macro `EILSEQ` shall be stored in `errno` and `-1` will be returned, but the conversion state will be unchanged.

If a conversion error occurs, `errno` may be set to `ECONVERT`.

**Examples**
This program is compiled with `LOCALETYPE(*LOCALE)` and `SYSIFCOPT(*IFSIO)`: 

```c
#include <stdio.h>
#include <locale.h>
#include <wchar.h>
#include <errno.h>
#define  STRLENGTH   10
#define  LOCNAME     "/qsys.lib/JA_JP.locale"
#define  LOCNAME_EN  "/qsys.lib/EN_US.locale"
int main(void)
{
    char       string[STRLENGTH];
...
int length, sl = 0;
wchar_t wc = 0x4171;
wchar_t wc2 = 0x00C1;
wchar_t wc_string[10];
mbstate_t ps = 0;
memset(string, '"\0', STRLENGTH);
wc_string[0] = 0x00C1;
wc_string[1] = 0x4171;
wc_string[2] = 0x4172;
wc_string[3] = 0x00C2;
wc_string[4] = 0x0000;

/* In this first example we will convert a wide character */
/* to a single byte character.  We first set the locale   */
/* to a single byte locale.  We choose a locale with      */
/* CCSID 37.  For single byte cases the state will always */
/* remain in the initial state 0 */

if (setlocale(LC_ALL, LOCNAME_EN) == NULL)
    printf("setlocale failed.\n");
length = wcrtomb(string, wc, &ps);
/* In this case since wc > 256 hex, length is -1 and */
/* errno is set to EILSEQ (3492) */
printf("errno = %d, length = %d\n", errno, length);
length = wcrtomb(string, wc2, &ps);
/* In this case wc2 00C1 is converted to C1 */
printf("string = %s\n", string);

/* Now lets try a multibyte example.  We first must set the */
/* locale to a multibyte locale.  We choose a locale with */
/* CCSID 5026 */
if (setlocale(LC_ALL, LOCNAME) == NULL)
    printf("setlocale failed.\n");
length = wcrtomb(string, wc_string[0], &ps);
/* The first character is < 256 hex so is converted to */
/* single byte and the state is still the initial state 0 */
printf("length = %d, state = %d\n", length, ps);
sl += length;

length = wcrtomb(&string[sl], wc_string[1], &ps);
/* The next character is > 256 hex so we get a shift out */
/* 0x0e followed by the double byte character.  State is */
/* changed to double byte state.  Length is 3. */
printf("length = %d, state = %d\n", length, ps);
sl += length;

length = wcrtomb(&string[sl], wc_string[2], &ps);
/* The next character is > 256 hex so we get another */
/* double byte character.  The state is left in */
/* double byte state.  Length is 2. */
printf("length = %d, state = %d\n", length, ps);
sl += length;

length = wcrtomb(&string[sl], wc_string[3], &ps);
/* The next character is < 256 hex so we close off the */
/* double byte characters with a shift in 0x0f and then */
/* get a single byte character.  Length is 2. */
/* The hex look at string would now be: */
/* C10E417141720FC2 */
/* You would need a device capable of displaying multibyte */
/* characters to see this string. */
printf("length = %d, state = %d\n", length, ps);
/* In the last example we will show what happens if NULL */
/* is passed in for the state. */
memset(string, '\0', STRLENGTH);
length = wcrtomb(string, wc_string[1], NULL);

/* The second character is > 256 hex so a shift out */
/* followed by the double character is produced but since */
/* the state is NULL, the double byte character is closed */
/* off with a shift in right away. So string we look */
/* like this: 0E41710F and length is 4 and the state is */
/* left in the initial state. */
printf("length = %d, state = %d\n" , length, ps);

} /* The output should look like this:
errno = 3492, length = -1
string = A
length = 1, state = 0
length = 3, state = 2
length = 2, state = 2
length = 2, state = 0
length = 4, state = 0 */

This program is compiled with LOCALETYPE(*LOCALEUCS2) and SYSIFOPT(*IFSIO):

#include <stdio.h>
#include <locale.h>
#include <wchar.h>
#include <errno.h>

#define STRLENGTH 10
#define LOCNAME    "/qsys.lib/JA_JP.locale"
#define LOCNAME_EN "/qsys.lib/EN_US.locale"

int main(void) {
    char     string[STRLENGTH];
    int      length, sl = 0;
    wchar_t  wc = 0x4171;
    wchar_t  wc2 = 0x0041;
    wchar_t  wc_string[10];
    mbstate_t ps = 0;
    memset(string, '\0', STRLENGTH);
    wc_string[0] = 0x0041;
    wc_string[1] = 0xFF31;
    wc_string[2] = 0xFF32;
    wc_string[3] = 0x0042;
    wc_string[4] = 0x0000;
    /* In this first example we will convert a UNICODE character */
    /* to a single byte character. We first set the locale */
    /* to a single byte locale. We choose a locale with */
    /* CCSID 37. For single byte cases the state will always */
    /* remain in the initial state 0 */
    if (setlocale(LC_ALL, LOCNAME_EN) == NULL)
        printf("setlocale failed.\n");
    length = wcrtomb(string, wc2, &ps);
    /* In this case wc2 0041 is converted to C1 */
    /* 0041 is UNICODE A, C1 is CCSID 37 A */
    printf("string = %s\n", string);
    /* Now lets try a multibyte example. We first must set the */
    /* locale to a multibyte locale. We choose a locale with */
    /* CCSID 5026 */
    if (setlocale(LC_ALL, LOCNAME) == NULL)
        printf("setlocale failed.\n");
    length = wcrtomb(string, wc_string[0], &ps);
/* The first character UNICODE character is converted to a */
/* single byte and the state is still the initial state 0 */
printf("length = %d, state = %d\n\n", length, ps);
sl += length;
length = wcrtomb(string[sl], wc_string[1], ps);
/* The next UNICODE character is converted to a shift out */
/* 0x0e followed by the double byte character. State is */
/* changed to double byte state. Length is 3. */
printf("length = %d, state = %d\n\n", length, ps);
sl += length;
length = wcrtomb(string[sl], wc_string[2], ps);
/* The UNICODE character is converted to another */
/* double byte character. The state is left in */
/* double byte state. Length is 2. */
printf("length = %d, state = %d\n\n", length, ps);
sl += length;
length = wcrtomb(string[sl], wc_string[3], ps);
/* The next UNICODE character converts to single byte so */
/* we close off the */
/* double byte characters with a shiftin 0x0f and then */
/* get a single byte character. Length is 2. */
/* The hex look at string would now be: */
/* C10E42D842D90FC2 */
/* You would need a device capable of displaying multibyte */
/* characters to see this string. */
printf("length = %d, state = %d\n\n", length, ps);
}

/* The output should look like this: */
string = A
length = 1, state = 0
length = 3, state = 2
length = 2, state = 2
length = 2, state = 0

Related Information
• “mblen() — Determine Length of a Multibyte Character” on page 219
• “mbrlen() — Determine Length of a Multibyte Character (Restartable)” on page 221
• “mbtowc() — Convert a Multibyte Character to a Wide Character (Restartable)” on page 223
• “mbstowcs() — Convert a Multibyte String to a Wide Character String (Restartable)” on page 227
• “wcsrtombs() — Convert Wide Character String to Multibyte String (Restartable)” on page 505
• “wctomb() — Convert Wide Character to Multibyte Character” on page 527
• “<wchar.h>” on page 16
wcscat() — Concatenate Wide-Character Strings

Format

```c
#include <wchar.h>
wchar_t *wcscat(wchar_t *string1, const wchar_t *string2);
```

Language Level

XPG4

Threadsafe

Yes

Wide Character Function

See “Wide Characters” on page 568 for more information.

Description

The `wcscat()` function 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-ended wchar_t strings. The string arguments to this function should contain a wchar_t null character marking the end of the string. Boundary checking is not performed.

Return Value

The `wcscat()` function returns a pointer to the concatenated `string1`.

Example

This example creates the wide character string "computer program" using the `wcscat()` function.

```c
#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 should be similar to: */

```
buffer1 = computer program
```

Related Information

- “strcat() — Concatenate Strings” on page 385
- “strncat() — Concatenate Strings” on page 406
- “wcschr() — Search for Wide Character” on page 484
wcschr() — Search for Wide Character

Format

```
#include <wchar.h>
wchar_t *wcschr(const wchar_t *string, wchar_t character);
```

Language Level

XPG4

Threading Safe

Yes

Wide Character Function

See “Wide Characters” on page 568 for more information.

Description

The `wcschr()` function searches the wide-character string for the occurrence of character. The character can be a wchar_t null character (\0); the wchar_t null character at the end of string is included in the search.

The `wcschr()` function operates on null-ended wchar_t strings. The string argument to this function should contain a wchar_t null character marking the end of the string.

Return Value

The `wcschr()` function returns a pointer to the first occurrence of character in string. If the character is not found, a NULL pointer is returned.

Example

This example finds the first occurrence of the character "p" in the wide-character string "computer program". 
#include <stdio.h>
#include <wchar.h>

#define SIZE 40

int main(void)
{
    wchar_t buffer1[SIZE] = L"computer program";
    wchar_t * ptr;
    wchar_t ch = L'p';

    ptr = wcschr( buffer1, ch );
    printf( "The first occurrence of %lc in '%ls' is '%ls\n",
            ch, buffer1, ptr );
}

/****************  Output should be similar to: ******************
The first occurrence of p in 'computer program' is 'puter program' */

Related Information

• “strchr() — Search for Character” on page 386
• “strcsppn() — Find Offset of First Character Match” on page 393
• “strpbrk() — Find Characters in String” on page 413
• “strchr() — Locate Last Occurrence of Character in String” on page 418
• “strspn() — Find Offset of First Non-matching Character” on page 419
• “wcscat() — Concatenate Wide-Character Strings” on page 483
• “wcsncmp() — Compare Wide-Character Strings” on page 485
• “wcscpy() — Copy Wide-Character Strings” on page 488
• “wcscspn() — Find Offset of First Wide-Character Match” on page 489
• “wcslen() — Calculate Length of Wide-Character String” on page 494
• “wcsncmp() — Compare Wide-Character Strings” on page 497
• “wcsrchr() — Locate Wide Characters in String” on page 501
• “wcsrchr() — Locate Last Occurrence of Wide Character in String” on page 504
• “wcspbrk() — Locate First Non-matching Wide Character” on page 507
• “wcswcs() — Locate Wide-Character Substring” on page 522
• “<wchar.h>” on page 16

wscmp() — Compare Wide-Character Strings

Format

#include <wchar.h>
int wcscmp(const wchar_t *string1, const wchar_t *string2);

Language Level

ANSI

Threading

Yes
Wide Character Function

See “Wide Characters” on page 568 for more information.

Description

The `wcscmp()` function compares two wide-character strings. The `wcscmp()` function operates on null-ended wchar_t strings; string arguments to this function should contain a wchar_t null character marking the end of the string. Boundary checking is not performed when a string is added to or copied.

Return Value

The `wcscmp()` function returns a value indicating the relationship between the two strings, as follows:

<table>
<thead>
<tr>
<th>Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>Less than 0</td>
<td><code>string1</code> less than <code>string2</code></td>
</tr>
<tr>
<td>0</td>
<td><code>string1</code> identical to <code>string2</code></td>
</tr>
<tr>
<td>Greater than 0</td>
<td><code>string1</code> greater than <code>string2</code></td>
</tr>
</tbody>
</table>

Example

This example compares the wide-character string `string1` to `string2` using `wcscmp()`.

```c
#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\n" 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 should be similar to:  ******************
"abcdef" is less than "abcdefg"
*/
```

Related Information

- “`strcmp()` — Compare Strings” on page 388
- “`strncmp()` — Compare Strings” on page 408
- “`wcscat()` — Concatenate Wide-Character Strings” on page 483
- “`wcschr()` — Search for Wide Character” on page 484
- “`wcsncpy()` — Copy Wide-Character Strings” on page 488
- “`wcscspn()` — Find Offset of First Wide-Character Match” on page 489
- “`wcslen()` — Calculate Length of Wide-Character String” on page 494
- “`wcscmp()` — Compare Wide-Character Strings” on page 497
- “__`wcsicmp()` — Compare Wide Character Strings without Case Sensitivity” on page 492
- “__`wcsnicmp()` — Compare Wide Character Strings without Case Sensitivity” on page 500
wcscoll() — Language Collation String Comparison

Format

```c
#include <wchar.h>
int wcscoll (const wchar_t *wcs1, const wchar_t *wcs2);
```

Language Level

XPG4

Threadsafe

Yes

Locale Sensitive

The behavior of this function might be affected by the LC_COLLATE category of the current locale if LOCALETYPE("LOCALE") is specified on the compilation command. The behavior of this function might also be affected by the LC_UNI_COLLATE category of the current locale if LOCALETYPE("LOCALEUCS2") or LOCALETYPE("LOCALEUTF") is specified on the compilation command. This function is not available when LOCALETYPE("CLD") is specified on the compilation command. For more information, see “Understanding CCSIDs and Locales” on page 565.

Wide Character Function

See “Wide Characters” on page 568 for more information.

Description

The wcscoll() function compares the wide-character strings pointed to by wcs1 and wcs2, both interpreted as appropriate to the LC_COLLATE category of the current locale (or the LC_UNI_COLLATE category if a UNICODE LOCALETYPE was specified).

Return Value

The wcscoll() function returns an integer value indicating the relationship between the strings, as follows:

<table>
<thead>
<tr>
<th>Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>Less than 0</td>
<td>wcs1 less than wcs2</td>
</tr>
<tr>
<td>0</td>
<td>wcs1 equivalent to wcs2</td>
</tr>
<tr>
<td>Greater than 0</td>
<td>wcs1 greater than wcs2</td>
</tr>
</tbody>
</table>

If wcs1 or wcs2 contain characters outside the domain of the collating sequence, the wcscoll() function sets errno to EINVAL. If an error occurs, the wcscoll() function sets errno to an nonzero value. There is no error return value.

Example

This example uses the default locale.
```c
#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("""%S" is identical to "%S"
", wcs1, wcs2);
    else if (result < 0)
        printf("""%S" is less than "%S"
", wcs1, wcs2);
    else
        printf("""%S" is greater than "%S"
", wcs1, wcs2);
}
```

Related Information

• “strcoll() — Compare Strings” on page 391
• “setlocale() — Set Locale” on page 366
• “<wchar.h>” on page 16

**wcsncpy() — Copy Wide-Character Strings**

**Format**

```c
#include <wchar.h>
wchar_t *wcsncpy(wchar_t *string1, const wchar_t *string2);
```

**Language Level**

XPG4

**Threadsafe**

Yes

**Wide Character Function**

See “Wide Characters” on page 568 for more information.

**Description**

The wcsncpy() function copies the contents of string2 (including the ending wchar_t null character) into string1.

The wcsncpy() function operates on null-ended wchar_t strings; string arguments to this function should contain a wchar_t null character marking the end of the string. Only string2 needs to contain a null character. Boundary checking is not performed.

**Return Value**

The wcsncpy() function returns a pointer to string1.

**Example**

This example copies the contents of source to destination.
```c
#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"
", destination );
    return_string = wcscpy( destination, source );
    printf( "After wcscpy, destination becomes "%ls"
", destination );
}
```

Related Information

- “strcpy() — Copy Strings” on page 392
- “strncpy() — Copy Strings” on page 409
- “wcsnccat() — Concatenate Wide-Character Strings” on page 483
- “wcschr() — Search for Wide Character” on page 484
- “wcsncmp() — Compare Wide-Character Strings” on page 485
- “wcsnscspn() — Find Offset of First Wide-Character Match” on page 489
- “wcsnslen() — Calculate Length of Wide-Character String” on page 494
- “wcsnscpy() — Copy Wide-Character Strings” on page 499
- “<wchar.h>” on page 16

---

**wcscspn() — Find Offset of First Wide-Character Match**

**Format**

```c
#include <wchar.h>
size_t wcscspn(const wchar_t *string1, const wchar_t *string2);
```

**Language Level**

XPG4

**Threading Safe**

Yes

**Wide Character Function**

See “Wide Characters” on page 568 for more information.

**Description**

The `wcscspn()` function determines the number of wchar_t 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-ended wchar_t strings; string arguments to this function should contain a wchar_t null character marking the end of the string.
Return Value
The `wcscspn()` function returns the number of wchar_t characters in the segment.

Example
This example uses `wcscspn()` to find the first occurrence of any of the characters a, x, l, or e in string.

```c
#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 should be similar to: ******************
The first 10 characters in the string "This is the source string" are not in the string "axle"
*/
```

Related Information
- “strcspsn() — Find Offset of First Character Match” on page 393
- “strspn() — Find Offset of First Non-matching Character” on page 419
- “wcscat() — Concatenate Wide-Character Strings” on page 483
- “wcschr() — Search for Wide Character” on page 484
- “wcsncmp() — Compare Wide-Character Strings” on page 485
- “wcsncpy() — Copy Wide-Character Strings” on page 488
- “wcslen() — Calculate Length of Wide-Character String” on page 494
- “wcsspsn() — Find Offset of First Non-matching Wide Character” on page 507
- “wcsrscs() — Locate Wide-Character Substring” on page 522
- “<wchar.h>” on page 16

**wcsftime() — Convert to Formatted Date and Time**

Format
```c
#include <wchar.h>
size_t wcsftime(wchar_t *wdest, size_t maxsize,
    const wchar_t *format, const struct tm *timeptr);
```

Language Level
ANSI

Threadsafe
Yes
Locale Sensitive

The behavior of this function might be affected by the LC_CTYPE, LC_TIME, and LC_TOD categories of the current locale if LOCALETYPE(*LOCALE) is specified on the compilation command. The behavior of this function might also be affected by the LC_UNI_CTYPE, LC_UNI_TIME, and LC_UNI_TOD categories of the current locale if LOCALETYPE(*LOCALEUCS2) or LOCALETYPE(*LOCALEUTF) is specified on the compilation command. This function is not available when LOCALETYPE(*CLD) is specified on the compilation command. For more information, see “Understanding CCSIDs and Locales” on page 565.

Wide Character Function

See “Wide Characters” on page 568 for more information.

Description

The wcsftime() function converts the time and date specification in the timeptr structure into a wide-character string. It then stores the null-ended string in the array pointed to by wdest according to the format string pointed to by format. The maxsize value specifies the maximum number of wide characters that can be copied into the array. This function is equivalent to strftime(), except that it uses wide characters.

The wcsftime() function works just like the strftime() function, except that it uses wide characters. The format string is a wide-character character string that contains:

- Conversion-specification characters.
- Ordinary wide characters, which are copied into the array unchanged.

This function uses the time structure pointed to by timeptr, and if the specifier is locale sensitive, then it will also use the LC_TIME category of the current locale to determine the appropriate replacement value of each valid specifier. The time structure pointed to by timeptr is usually obtained by calling the gmtime() or localtime() function.

Return Value

If the total number of wide characters in the resulting string, including the ending null wide character, does not exceed maxsize, wcsftime() returns the number of wide characters placed into wdest, not including the ending null wide character. Otherwise, the wcsftime() function returns 0 and the contents of the array are indeterminate.

If a conversion error occurs, errno may be set to ECONVERT.

Example

This example obtains the date and time using localtime(), formats the information with the wcsftime(), and prints the date and time.

```c
#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), L" Today is %A, "
                  L" %b %d. \n Time: %I:%M %p", timeptr);
    printf("%d characters placed in string to make:\n%ls\n", rc, dest);
    return 0;
}
```

The output should be similar to:
Today is Thursday, Nov 10.
Time: 04:56 PM
********************************************************************/

Related Information

• “ctime() — Convert Time to Character String” on page 96
• “ctime64() — Convert Time to Character String” on page 98
• “ctime64_r() — Convert Time to Character String (Restartable)” on page 101
• “ctime_r() — Convert Time to Character String (Restartable)” on page 100
• “gmtime() — Convert Time” on page 183
• “gmtime64() — Convert Time” on page 185
• “gmtime64_r() — Convert Time (Restartable)” on page 189
• “gmtime_r() — Convert Time (Restartable)” on page 187
• “localtime() — Convert Time” on page 207
• “localtime64() — Convert Time” on page 208
• “localtime64_r() — Convert Time (Restartable)” on page 211
• “localtime_r() — Convert Time (Restartable)” on page 210
• “strftime() — Convert Date/Time to String” on page 399
• “strptime() — Convert String to Date/Time” on page 414
• “time() — Determine Current Time” on page 441
• “time64() — Determine Current Time” on page 443
• “<wchar.h>” on page 16

__wcsicmp() — Compare Wide Character Strings without Case Sensitivity

Format

```c
#include <wchar.h>
int __wcsicmp(const wchar_t *string1, const wchar_t *string2);
```

Language Level

Extension

Threadsafe

Yes

Locale Sensitive

The behavior of this function might be affected by the LC_CTYPE category of the current locale if LOCALETYPE(*LOCALE) is specified on the compilation command. The behavior of this function might also be affected by the LC_UNI_CTYPE category of the current locale if LOCALETYPE(*LOCALEUCS2) or LOCALETYPE(*LOCALEUTF) is specified on the compilation command. This function is not available when LOCALETYPE(*CLD) is specified on the compilation command. For more information, see “Understanding CCSIDs and Locales” on page 565.
Wide Character Function

See “Wide Characters” on page 568 for more information.

Description

The `__wcsicmp()` function compares `string1` and `string2` without sensitivity to case. All alphabetic wide characters in `string1` and `string2` are converted to lowercase before comparison. The function operates on null terminated wide character strings. The string arguments to the function are expected to contain a `wchar_t` null character (L'\0') marking the end of the string.

Return Value

The `__wcsicmp()` function returns a value indicating the relationship between the two strings as follows:

<table>
<thead>
<tr>
<th>Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>Less than 0</td>
<td><code>string1</code> less than <code>string2</code></td>
</tr>
<tr>
<td>0</td>
<td><code>string1</code> equivalent to <code>string2</code></td>
</tr>
<tr>
<td>Greater than 0</td>
<td><code>string1</code> greater than <code>string2</code></td>
</tr>
</tbody>
</table>

Example

This example uses `__wcsicmp()` to compare two wide character strings.

```c
#include <stdio.h>
#include <wchar.h>

int main(void)
{
    wchar_t *str1 = L"STRING";
    wchar_t *str2 = L"string";
    int result;

    result = __wcsicmp(str1, str2);
    if (result == 0)
        printf("Strings compared equal.\n");
    else if (result < 0)
        printf(""%ls" is less than "%ls".\n", str1, str2);
    else
        printf(""%ls" is greater than "%ls".\n", str1, str2);
    return 0;
}

/******** The output should be similar to: ************/
Strings compared equal.
**************
```

Related Information

- “strcmp() — Compare Strings” on page 388
- “strncmp() — Compare Strings” on page 408
- “wcscat() — Concatenate Wide-Character Strings” on page 483
- “wcschr() — Search for Wide Character” on page 484
- “wcscspn() — Find Offset of First Wide-Character Match” on page 489
- “wcslen() — Calculate Length of Wide-Character String” on page 494
wcslen() — Calculate Length of Wide-Character String

Format

```c
#include <wchar.h>
size_t wcslen(const wchar_t *string);
```

Language Level

XPG4

Threading

Yes

Wide Character Function

See “Wide Characters” on page 568 for more information.

Description

The `wcslen()` function computes the number of wide characters in the string pointed to by `string`.

Return Value

The `wcslen()` function returns the number of wide characters in `string`, excluding the ending wchar_t null character.

Example

This example computes the length of the wide-character string `string`.

```c
#include <stdio.h>
#include <wchar.h>

int main(void)
{
    wchar_t * string = L"abcdef";
    printf( "Length of "%ls" is %i\n", string, wcslen( string ));
}
```

Example Output

```
/********************  Output should be similar to:  ******************
Length of "abcdef" is 6
*/
```

Related Information

- “mblen() — Determine Length of a Multibyte Character” on page 219
- “strlen() — Determine String Length” on page 404
- “wcsncat() — Concatenate Wide-Character Strings” on page 496
- “wcsncmp() — Compare Wide-Character Strings” on page 497
- “wcsncpy() — Copy Wide-Character Strings” on page 499
- “<wchar.h>” on page 16
wcslocaleconv() — Retrieve Wide Locale Information

Format

```c
#include <locale.h>
struct wcslconv *wcslocaleconv(void);
```

Language Level

Extended

Threadsafe

Yes

Locale Sensitive

The behavior of this function might be affected by the LC_UNI_NUMERIC and LC_UNI_MONETARY categories of the current locale. This function is only available when LOCALETYPE("LOCALEUCS2) or LOCALETYPE("LOCALEUTF) is specified on the compilation command. For more information, see "Understanding CCSIDs and Locales" on page 565.

Wide Character Function

See “Wide Characters” on page 568 for more information.

Description

The wcslocaleconv() function is the same as the localeconv() function, except that it returns a pointer to a wcslconv structure, which is the wide version of a lconv structure. These elements are determined by the LC_UNI_MONETARY and LC_UNI_NUMERIC categories of the current locale.

Return Value

The wcslocaleconv() function returns a pointer to a wcslconv structure.

Example

This example prints out the Unicode currency symbol for a French locale.
This example prints out the Unicode currency symbol for a French locale. You first must create a Unicode French locale. You can do this with this command:

```
CRTLOCALE LOCALE('/QSYS.LIB/MYLIB.LIB/LC_UNI_FR.LOCALE') +
SRCFILE('/QSYS.LIB/QSYSLOCALE.LIB/QLOCALESRC.FILE/ +
FR_FR.MBR') CCSID(13488)
```

Then you must compile your C program with `LOCALETYPE(*LOCALEUCS2)`

```c
#include <stdio.h>
#include <locale.h>

int main(void) {
    char * string;
    struct wcsConv * mylocale;
    if (NULL != (string = setlocale(LC_UNI_ALL,
                        "/QSYS.LIB/MYLIB.LIB/LC_UNI_FR.LOCALE"))) {
        mylocale = wcslocaleconv();
        /* Display the Unicode currency symbol in a French locale */
        printf("French Unicode currency symbol is a %ls\n",
               mylocale->currency_symbol);
    } else {
        printf("setlocale(LC_UNI_ALL, "/QSYS.LIB/MYLIB.LIB/LC_UNI_FR.LOCALE") \n returned <NULL>\n"n);
    }
    return 0;
}
```

**Related Information**

- “setlocale() — Set Locale” on page 366
- “<locale.h>” on page 5
- “localeconv() — Retrieve Information from the Environment” on page 202

### wcsncat() — Concatenate Wide-Character Strings

**Format**

```c
#include <wchar.h>
wchar_t *wcsncat(wchar_t *string1, const wchar_t *string2, size_t count);
```

**Language Level**

XPG4

**Threading**

Yes

**Wide Character Function**

See “Wide Characters” on page 568 for more information.

**Description**

The `wcsncat()` function appends up to `count` wide characters from `string2` to the end of `string1`, and appends a wchar_t null character to the result.

The `wcsncat()` function operates on null-ending wide-character strings; string arguments to this function should contain a wchar_t null character marking the end of the string.

**Return Value**

The `wcsncat()` function returns `string1`. 
Example
This example demonstrates the difference between the `wcscat()` and `wcsncat()` functions. The `wcscat()` function appends the entire second string to the first; the `wcsncat()` function appends only the specified number of characters in the second string to the first.

```c
#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"
", buffer1 );
    /* Reset buffer1 to contain just the string "computer" again */
    memset( buffer1, L'\0', sizeof( buffer1 ));
    ptr = wcsncpy( buffer1, L"computer" );
    /* Call wcsncat with buffer1 and " program" */
    ptr = wcsncat( buffer1, L" program", 3 );
    printf( "wcsncat: buffer1 = "%ls"
", buffer1 );
}

*********** Output should be similar to: ***********
wcsncat: buffer1 = "computer pr"
```

Related Information

- “`strcat()` — Concatenate Strings” on page 385
- “`strncat()` — Concatenate Strings” on page 406
- “`wcscat()` — Concatenate Wide-Character Strings” on page 483
- “`wcsncmp()` — Compare Wide-Character Strings” on page 497
- “`wcsncpy()` — Copy Wide-Character Strings” on page 499
- “`<wchar.h>`” on page 16

**wcsncmp() — Compare Wide-Character Strings**

**Format**

```c
#include <wchar.h>
int wcsncmp(const wchar_t *string1, const wchar_t *string2, size_t count);
```

**Language Level**

XPG4

**Threadsafe**

Yes

**Wide Character Function**

See “Wide Characters” on page 568 for more information.
Description

The `wcsncmp()` function compares up to `count` wide characters in `string1` to `string2`.

The `wcsncmp()` function operates on null-ended wide-character strings; string arguments to this function should contain a `wchar_t` null character marking the end of the string.

Return Value

The `wcsncmp()` function returns a value indicating the relationship between the two strings, as follows:

<table>
<thead>
<tr>
<th>Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>Less than 0</td>
<td><code>string1</code> less than <code>string2</code></td>
</tr>
<tr>
<td>0</td>
<td><code>string1</code> identical to <code>string2</code></td>
</tr>
<tr>
<td>Greater than 0</td>
<td><code>string1</code> greater than <code>string2</code></td>
</tr>
</tbody>
</table>

Example

This example demonstrates the difference between the `wcscmp()` function, which compares the entire strings, and the `wcsncmp()` function, which compares only a specified number of wide characters in the strings.

```c
#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 should be similar to: */
Comparison of each character
wcscmp: "abcdefg" is less than "abcfg"
Comparison of only the first 3 characters
wcscmp: "abcdefg" is identical to "abcfg"
*/
```

Related Information

- “strcmpr() — Compare Strings” on page 388
wcsncpy() — Copy Wide-Character Strings

Format

```c
#include <wchar.h>
wchar_t *wcsncpy(wchar_t *string1, const wchar_t *string2, size_t count);
```

Language Level

XPG4

Threadsafe

Yes

Wide Character Function

See “Wide Characters” on page 568 for more information.

Description

The `wcsncpy()` function 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 wchar_t null characters.

The `wcsncpy()` function operates on null-ended wide-character strings; string arguments to this function should contain a wchar_t null character marking the end of the string. Only `string2` needs to contain a null character.

Return Value

The `wcsncpy()` returns a pointer to `string1`.

Related Information

- “strcpy() — Copy Strings” on page 392
- “strncpy() — Copy Strings” on page 409
- “wcsncpy() — Copy Wide-Character Strings” on page 488
- “wcsncat() — Concatenate Wide-Character Strings” on page 496
- “wcsncmp() — Compare Wide-Character Strings” on page 497
- “<wchar.h>” on page 16
__wcsnicmp() — Compare Wide Character Strings without Case Sensitivity

**Format**

```c
#include <wchar.h>
int __wcsnicmp(const wchar_t *string1, const wchar_t *string2, size_t count);
```

**Language Level**

Extension

**Threadsafe**

Yes

**Locale Sensitive**

The behavior of this function might be affected by the LC_CTYPE category of the current locale if LOCALETYPE(*LOCALE) is specified on the compilation command. The behavior of this function might also be affected by the LC_UNI_CTYPE category of the current locale if LOCALETYPE(*LOCALEUCS2) or LOCALETYPE(*LOCALEUTF) is specified on the compilation command. This function is not available when LOCALETYPE(*CLD) is specified on the compilation command. For more information, see “Understanding CCSIDs and Locales” on page 565.

**Wide Character Function**

See “Wide Characters” on page 568 for more information.

**Description**

The __wcsnicmp() function compares up to count characters of string1 and string2 without sensitivity to case. All alphabetic wide characters in string1 and string2 are converted to lowercase before comparison.

The __wcsnicmp() function operates on null terminated wide character strings. The string arguments to the function are expected to contain a wchar_t null character (L'\0') marking the end of the string.

**Return Value**

The __wcsnicmp() function returns a value indicating the relationship between the two strings, as follows:

<table>
<thead>
<tr>
<th>Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>Less than 0</td>
<td>string1 less than string2</td>
</tr>
<tr>
<td>0</td>
<td>string1 equivalent to string2</td>
</tr>
<tr>
<td>Greater than 0</td>
<td>string1 greater than string2</td>
</tr>
</tbody>
</table>

**Example**

This example uses __wcsnicmp() to compare two wide character strings.

```c
#include <stdio.h>
#include <wchar.h>

int main(void)
{
    wchar_t *str1 = L"STRING ONE";
    wchar_t *str2 = L"string TWO";
```
```c
int result;
result = __wcsnicmp(str1, str2, 6);
if (result == 0)
  printf("Strings compared equal.\n");
ext if (result < 0)
  printf("\"%ls\" is less than \"%ls\".\n", str1, str2);
else
  printf("\"%ls\" is greater than \"%ls\".\n", str1, str2);
return 0;

/******** The output should be similar to: **************/
Strings compared equal.
*************************************************************************/
```

### Related Information
- “strcmp() — Compare Strings” on page 388
- “strncmp() — Compare Strings” on page 408
- “wcscat() — Concatenate Wide-Character Strings” on page 483
- “wcschr() — Search for Wide Character” on page 484
- “wcschrn() — Find Offset of First Wide-Character Match” on page 489
- “wcslen() — Calculate Length of Wide-Character String” on page 494
- “wcsncmp() — Compare Wide-Character Strings” on page 497
- “__wcsicmp() — Compare Wide Character Strings without Case Sensitivity ” on page 492
- “<wchar.h>” on page 16

### wcsbrk() — Locate Wide Characters in String

#### Format
```
#include <wchar.h>
wchar_t *wcsbrk(const wchar_t *string1, const wchar_t *string2);
```

#### Language Level
XPG4

#### Threadsafe
Yes

#### Wide Character Function
See “Wide Characters” on page 568 for more information.

#### Description
The `wcsbrk()` function locates the first occurrence in the string pointed to by `string1` of any wide character from the string pointed to by `string2`.

#### Return Value
The `wcsbrk()` function returns a pointer to the character. If `string1` and `string2` have no wide characters in common, the `wcsbrk()` function returns NULL.
Example

This example uses `wcsbPRIk()` to find the first occurrence of either "a" or "b" in the array `string`.

```c
#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 = wcsbPRIk( string, chars);
    printf("The first occurrence of any of the characters \"%ls\" in \"%ls\" is \"%ls\"
           \n", chars, string, result);
}

/****************  Output should be similar to:  ******************
The first occurrence of any of the characters "ab" in "The Blue Danube" is "anube"
******************************************
```

Related Information

- "strchr() — Search for Character" on page 386
- "strcspn() — Find Offset of First Character Match" on page 393
- "strpbrk() — Find Characters in String" on page 413
- "strspn() — Find Offset of First Non-matching Character" on page 419
- "wcschr() — Search for Wide Character" on page 484
- "wcscmp() — Compare Wide-Character Strings" on page 485
- "wcsncmp() — Compare Wide-Character Strings" on page 489
- "wcsncmpeq() — Compare Wide-Character Strings” on page 497
- "wcschr() — Locate Last Occurrence of Wide Character in String" on page 504
- "wcswcschr() — Locate Wide-Character Substring” on page 522
- "<wchar.h>” on page 16

`wcsptime()` — Convert Wide Character String to Date/Time

Format

```c
#include <wchar.h>
wchar_t *wcsptime(const wchar_t *buf, const wchar_t *format, struct tm *tm);
```

Language Level

Extended

Threatsafe

Yes

Locale Sensitive

The behavior of this function might be affected by the LC_UNI_CTYPE, LC_UNI_TIME, and LC_UNI_TOD categories of the current locale. This function is only available when LOCALETYPE(*LOCALE_UTF) is specified on the compilation command. For more information, see “Understanding CCSIDs and Locales” on page 565.
Wide Character Function

See “Wide Characters” on page 568 for more information.

Description

The `wcsftime()` function converts the wide character string pointed to by `buf` to values that are stored in the `tm` structure pointed to by `tm`, using the format specified by `format`. This function is equivalent to `strptime()`, except that it uses wide characters.

See “`strptime() — Convert String to Date/Time`” on page 414 for a description of the format string.

Return Value

On successful completion, the `wcsftime()` function returns a pointer to the character following the last wide character parsed. Otherwise, a null pointer is returned. The value of `errno` may be set to `ECONV` (conversion error).

Example

```c
#include <stdio.h>
#include <time.h>
#include <wchar.h>

int main(void)
{
    wchar_t buf[100];
    time_t t;
    struct tm *timeptr,result;
    t = time(NULL);
    timeptr = localtime(&t);
    wcsftime(buf, 100, L"%a %m/%d/%Y %r", timeptr);
    if (wcsftime(buf, L"%a %m/%d/%Y %r", &result) == NULL)
        printf("wcsftime failed\n");
    else
    {
        printf("tm_hour:  %d
",result.tm_hour);
        printf("tm_min:  %d
",result.tm_min);
        printf("tm_sec:  %d
",result.tm_sec);
        printf("tm_mon:  %d
",result.tm_mon);
        printf("tm_mday:  %d
",result.tm_mday);
        printf("tm_year:  %d
",result.tm_year);
        printf("tm_yday:  %d
",result.tm_yday);
        printf("tm_wday:  %d
",result.tm_wday);
    }
    return 0;
}
```

The output should be similar to:

```
tm_hour:  14
_tm_min:  25
_tm_sec:  34
_tm_mon:  7
_tm_mday:  19
_tm_year:  103
_tm_yday:  230
_tm_wday:  2
```

Related Information

- “`asctime() — Convert Time to Character String`” on page 63
- “`asctime_r() — Convert Time to Character String (Restartable)`” on page 65
- “`ctime() — Convert Time to Character String`” on page 96
- “`ctime64() — Convert Time to Character String`” on page 98
wcsrchr() — Locate Last Occurrence of Wide Character in String

**Format**

```c
#include <wchar.h>
wchar_t *wcsrchr(const wchar_t *string, wchar_t character);
```

**Language Level**

ANSI

**Threadsafe**

Yes

**Wide Character Function**

See “Wide Characters” on page 568 for more information.

**Description**

The `wcsrchr()` function locates the last occurrence of `character` in the string pointed to by `string`. The ending `wchar_t` null character is considered to be part of the string.

**Return Value**

The `wcsrchr()` function returns a pointer to the character, or a NULL pointer if `character` does not occur in the string.

**Example**

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.
```c
#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 wcsrchr */
    ptr = wcsrchr(buf, ch);
    printf("The last occurrence of %c in '%ls' is '%ls'\n", ch, buf, ptr);
}

/******* Output should be similar to: *******
The first occurrence of p in 'computer program' is 'puter program'
The last occurrence of p in 'computer program' is 'program'
*/
```

**Related Information**

- “strchr() — Search for Character” on page 386
- “strchr() — Locate Last Occurrence of Character in String” on page 418
- “strcspn() — Find Offset of First Character Match” on page 393
- “strspn() — Find Offset of First Non-matching Character” on page 419
- “wcschr() — Search for Wide Character” on page 484
- “wcscmp() — Compare Wide-Character Strings” on page 485
- “wcscspn() — Find Offset of First Wide-Character Match” on page 489
- “wcsncmp() — Compare Wide-Character Strings” on page 497
- “wcswcs() — Locate Wide-Character Substring” on page 522
- “wcspbrk() — Locate Wide Characters in String” on page 501
- “<wchar.h>” on page 16

**wcsrtombs() — Convert Wide Character String to Multibyte String (Restartable)**

**Format**

```c
#include <wchar.h>
size_t wcsrtombs (char *dst, const wchar_t **src, size_t len,
                  mbstate_t *ps);
```

**Language Level**

ANSI

**Thesafe**

Yes, if the fourth parameter, `ps`, is not NULL.
Locale Sensitive

The behavior of this function might be affected by the LC_CTYPE category of the current locale. The behavior might also be affected by the LC_UNI_CTYPE category of the current locale if LOCALETYPE(*LOCALEUCS2) or LOCALETYPE(*LOCALEUTF) is specified on the compilation command. This function is not available when LOCALETYPE(*CLD) is specified on the compilation command. For more information, see “Understanding CCSIDs and Locales” on page 565.

Wide Character Function

See “Wide Characters” on page 568 for more information.

Description

This function is the restartable version of wcstombs().

The wcsrtombs() function converts a sequence of wide characters from the array indirectly pointed to by src into a sequence of corresponding multibyte characters that begins 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 ending null wide character, which is also stored. Conversion will stop earlier in two cases: when a code is reached that does not correspond to a valid multibyte character, or (if dst is not a null pointer) 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 place as if by a call to wcrtomb().

If dst is not a null pointer, the object pointed to by src will be assigned either a null pointer (if conversion stopped due to reaching a ending null character) or the address of the code just past the last wide character converted. If conversion stopped due to reaching a ending null wide character, the resulting state described will be the initial conversion state.

Return Value

If the first code is not a valid wide character, an encoding error will occur. wcsrtombs() stores the value of the macro EILSEQ in errno and returns (size_t) -1, but the conversion state will be unchanged. Otherwise it returns the number of bytes in the resulting multibyte character sequence, which is the same as the number of array elements changed when dst is not a null pointer.

If a conversion error occurs, errno may be set to ECONVERT.
Example

```c
#include <stdio.h>
#include <wchar.h>
#include <string.h>

#define SIZE 20

int main(void)
{
    char     dest[SIZE];
    wchar_t *wcs = L"string";
    wchar_t *ptr;
    size_t   count = SIZE;
    size_t   length;
    mbstate_t ps = 0;

    ptr = (wchar_t *) wcs;
    length = wcsrtombs(dest, ptr, count, &ps);
    printf("%d characters were converted.\n", length);
    printf("The converted string is \"%s\"\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, &ps);
    printf("%d characters were converted.\n", length);
    printf("The converted string is \"%s\"\n", dest);
}
```

Related Information

- “mblen() — Determine Length of a Multibyte Character” on page 219
- “mbrlen() — Determine Length of a Multibyte Character (Restartable)” on page 221
- “mbtowc() — Convert a Multibyte Character to a Wide Character (Restartable)” on page 223
- “mbstowcs() — Convert a Multibyte String to a Wide Character String (Restartable)” on page 227
- “wcrtomb() — Convert a Wide Character to a Multibyte Character (Restartable)” on page 478
- “wcstombs() — Convert Wide-Character String to Multibyte String” on page 517
- “<wchar.h>” on page 16

**wcspn() — Find Offset of First Non-matching Wide Character**

**Format**

```c
#include <wchar.h>
size_t wcspn(const wchar_t *string1, const wchar_t *string2);
```

**Language Level**

ANSI

**Threadsafe**

Yes
Wide Character Function
See “Wide Characters” on page 568 for more information.

Description
The wcsspn() function computes the number of wide characters in the initial segment of the string pointed to by string2, which consists entirely of wide characters from the string pointed to by string2.

Return Value
The wcsspn() function returns the number of wide characters in the segment.

Example
This example finds the first occurrence in the array string of a wide character that is not an a, b, or c. Because the string in this example is cabbage, the wcsspn() function returns 5, the index of the segment of cabbage before a character that is not an a, b, or c.

```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 should be similar to: *******
The first 5 characters of "cabbage" are found in "abc"
*/
```

Related Information
- “strchr() — Search for Character” on page 386
- “strcspn() — Find Offset of First Character Match” on page 393
- “strchr() — Search for Wide Character” on page 484
- “wcschr() — Search for Wide Character” on page 484
- “wcscat() — Concatenate Wide-Character Strings” on page 483
- “wcscmp() — Compare Wide-Character Strings” on page 485
- “wcsncmp() — Compare Wide-Character Strings” on page 487
- “wcscmp() — Find Offset of First Non-matching Wide Character” on page 489
- “wcscspn() — Find Offset of First Non-matching Wide Character” on page 504
- “wcschrb() — Locate Last Occurrence of Wide Character in String” on page 507
- “wcsstr() — Locate Last Occurrence of Wide Character in String” on page 504
- “wcscspn() — Locate Last Occurrence of Wide Character in String” on page 504
- “wcswcs() — Locate Wide-Character Substring” on page 522
- “<wchar.h>” on page 16
wcsstr() — Locate Wide-Character Substring

Format

```c
#include <wchar.h>
wchar_t *wcsstr(const wchar_t *wcs1, const wchar_t *wcs2);
```

Language Level

ANSI

Threadsafe

Yes

Wide Character Function

See “Wide Characters” on page 568 for more information.

Description

The wcsstr() function locates the first occurrence of wcs2 in wcs1.

Return Value

The wcsstr() function returns a pointer to the beginning of the first occurrence of wcs2 in wcs1. If wcs2 does not appear in wcs1, the wcsstr() function returns NULL. If wcs2 points to a wide-character string with zero length, it returns wcs1.

Example

This example uses the wcsstr() function to find the first occurrence of "hay" in the wide-character string "needle in a haystack".

```c
#include <stdio.h>
#include <wchar.h>

int main(void)
{
    wchar_t *wcs1 = L"needle in a haystack";
    wchar_t *wcs2 = L"hay";
    printf("result: "%ls\n", wcsstr(wcs1, wcs2));
    return 0;

    /********************
    The output should be similar to:
    result: "haystack"
    ********************/
}
```

Related Information

- “strstr() — Locate Substring” on page 421
- “wcschr() — Search for Wide Character” on page 484
- “wcsrchr() — Locate Last Occurrence of Wide Character in String” on page 504
- “wcscws() — Locate Wide-Character Substring” on page 522
- “<wchar.h>” on page 16
**wcstod() - wcstof() - wcstold() — Convert Wide-Character String to Double, Float, and Long Double**

**Format**

```c
#include <wchar.h>
double wcstod(const wchar_t *nptr, wchar_t **endptr);
float wcstof(const wchar_t *nptr, wchar_t **endptr);
long double wcstold(const wchar_t *nptr, wchar_t **endptr);
```

**Language Level**

XPG4

**Threadsafe**

Yes

**Locale Sensitive**

The behavior of this function might be affected by the LC_CTYPE and LC_NUMERIC categories of the current locale if LOCALETYPE(*LOCAL) is specified on the compilation command. The behavior of this function might also be affected by the LC_UNI_CTYPE and LC_UNI_NUMERIC categories of the current locale if LOCALETYPE(*LOCALUCS2) or LOCALETYPE(*LOCALUTF) is specified on the compilation command. This function is not available when LOCALETYPE(*CLD) is specified on the compilation command. For more information, see “Understanding CCSIDs and Locales” on page 565.

**Wide Character Function**

See “Wide Characters” on page 568 for more information.

**Description**

The `wcstod()`, `wcstof()`, `wcstold()` functions convert the initial portion of the wide-character string pointed to by `nptr` to a double, float or long double value. The `nptr` parameter points to a sequence of characters that can be interpreted as a numeric binary floating-point value. These functions stop reading the string at the first character that it cannot recognize as part of a number. This character can be the wchar_t null character at the end of the string.

The `wcstod()`, `wcstof()`, `wcstold()` functions expect `nptr` to point to a string with the following form:

```
whitespace [+ -]digits . digits e E [+ -]digits . digits
```

The first character that does not fit this form stops the scan. In addition, a sequence of INFINITY or NAN (ignoring case) is allowed.
If an exponent is specified with the hexadecimal digit form, the exponent is interpreted as a binary (base 2) exponent. If an exponent is specified with the decimal digit form, the exponent is interpreted as a decimal (base 10) exponent.

**Return Value**

The `wcstod()`, `wcstof()`, `wcstold()` functions return the converted double, float or long double value. If no conversion could be performed, these functions return 0. If the correct value is outside the range of representable values, these functions return `+HUGE_VAL` or `-HUGE_VAL` (according to the sign of the value), and set `errno` to `ERANGE`. If the correct value would cause underflow, these functions return 0 and set `errno` to `ERANGE`. If the string `nptr` points to is empty or does not have the expected form, no conversion is performed, and the value of `nptr` is stored in the object pointed to by `endptr`, provided that `endptr` is not a null pointer.

The `wcstod()`, `wcstof()`, `wcstold()` functions do not fail if a character other than a digit follows an E or e that is read as an exponent. For example, 100elf is converted to the floating-point value 100.0.

The value of `errno` may be set to `ERANGE`, range error.

A character sequence of `INFINITY` (ignoring case) yields a value of `INFINITY`. A character value of `NAN` yields a Quiet Not-A-Number (NAN) value.

**Example**

This example uses the `wcstod()`, `wcstof()`, `wcstold()` functions to convert the string `wcs` to double, float and long double values.

```c
#include <stdio.h>
#include <wchar.h>

int main(void)
{
    wchar_t *wcs = L"3.1415926This stopped it";
    wchar_t *stopwcs;

    printf("wcs = "L"\%ls\"\n", wcs);
    printf(" wcstod = %f\n", wcstod(wcs, &stopwcs));
    printf(" Stop scanning at "L"\%ls\"\n", stopwcs);

    printf(" wcstof = %f\n", wcstof(wcs, &stopwcs));
    printf(" Stop scanning at "L"\%ls\"\n", stopwcs);

    printf(" wcstold = %lf\n", wcstold(wcs, &stopwcs));
    printf(" Stop scanning at "L"\%ls\"\n", stopwcs);
    return 0;
}
```

The output should be similar to:

```
wcs = "3.1415926This stopped it"
wcestod = 3.141593
Stop scanning at "This stopped it"
wcestof = 3.141593
Stop scanning at "This stopped it"
wcestold = 3.141593
Stop scanning at "This stopped it"
```

**Related Information**

- “`strtod() - strtof() - strtold()` — Convert Character String to Double, Float, and Long Double” on page 422
- “`strtod32() - strtod64() - strtod128()` — Convert Character String to Decimal Floating-Point” on page 425
- “`strtol() – strtoll()` — Convert Character String to Long and Long Long Integer” on page 430
- “`wcstod32() - wcstod64() - wcstod128()` — Convert Wide-Character String to Decimal Floating-Point” on page 512
- “`wcstol()` — Convert Wide Character String to Long and Long Long Integer” on page 515
wcstod32() - wcstod64() - wcstod128() — Convert Wide-Character String to Decimal Floating-Point

Format

```c
#define __STDC_WANT_DEC_FP__
#include <wchar.h>

__Decimal32 wcstod32(const wchar_t *nptr, wchar_t **endptr);
__Decimal64 wcstod64(const wchar_t *nptr, wchar_t **endptr);
__Decimal128 wcstod128(const wchar_t *nptr, wchar_t **endptr);
```

Language Level

XPG4

Threadsafe

Yes

Locale Sensitive

The behavior of these functions might be affected by the LC_CTYPE and LC_NUMERIC categories of the current locale if LOCALETYPE(*LOCALE) is specified on the compilation command. The behavior of these functions might also be affected by the LC_UNI_CTYPE and LC_UNI_NUMERIC categories of the current locale if LOCALETYPE(*LOCALEUCS2) or LOCALETYPE(*LOCALEUTF) is specified on the compilation command. These functions are not available when LOCALETYPE(*CLD) is specified on the compilation command. For more information, see “Understanding CCSIDs and Locales” on page 565.

Wide Character Function

See “Wide Characters” on page 568 for more information.

Description

The wcstod32(), wcstod64(), and wcstod128() functions convert the initial portion of the wide-character string pointed to by nptr to a single-precision, double-precision, or quad-precision decimal floating-point value. The parameter nptr points to a sequence of characters that can be interpreted as a numeric decimal floating-point value. The wcstod32(), wcstod64(), and wcstod128() functions stop reading the string at the first character that is not recognized as part of a number. This character can be the wchar_t null character at the end of the string. The endptr parameter is updated to point to this character, provided that endptr is not a NULL pointer.

The wcstod32(), wcstod64(), and wcstod128() functions expect nptr to point to a string with the following form:
The first character that does not fit this form stops the scan. In addition, a sequence of INFINITY or NAN (ignoring case) is allowed.

Return Value

The wcstod32(), wcstod64(), and wcstod128() functions return the value of the floating-point number, except when the representation causes an underflow or overflow. For an overflow, wcstod32() returns HUGE_VAL_D32 or -HUGE_VAL_D32; wcstod64() returns HUGE_VAL_D64 or -HUGE_VAL_D64; wcstod128() returns HUGE_VAL_D128 or -HUGE_VAL_D128. For an underflow, all functions return +0.E0.

In both the overflow and underflow cases, errno is set to ERANGE. If the string pointed to by nptr does not have the expected form, a value of +0.E0 is returned and the value of nptr is stored in the object pointed to by endptr, provided that endptr is not a NULL pointer.

The wcstod32(), wcstod64(), and wcstod128() functions do not fail if a character other than a digit follows an E or e that is read as an exponent. For example, 100elf is converted to the floating-point value 100.0.

A character sequence of INFINITY (ignoring case) yields a value of INFINITY. A character value of NAN (ignoring case) yields a Quiet Not-A-Number (NaN) value.

If necessary, the return value is rounded using the rounding mode Round to Nearest, Ties to Even.

Example

This example converts the string wcs to single-precision, double-precision, and quad-precision decimal floating-point values.

```c
#define __STDC_WANT_DEC_FP__
#include <stdio.h>
#include <wchar.h>

int main(void)
{
    wchar_t *wcs = L"3.1415926This stopped it";
    wchar_t *stopwcs;

    printf("wcs = "%ls"
        printf(" wcstod32 = %Hf
            printf(" Stopped scan at "%ls"
        printf(\"wcs = \"%ls\"\n            return wcstod32(wcs, &stopwcs));
        printf(" Stopped scan at \"%ls\"\n            return wcstod64(wcs, &stopwcs));
        printf(" Stopped scan at \"%ls\"\n            return wcstod128(wcs, &stopwcs));
    }

    /*************** Output should be similar to: ***************

    wcs = "3.1415926This stopped it"
    wcstod32 = 3.141593
    Stopped scan at "This stopped it"
    wcs = "3.1415926This stopped it"
    wcstod64 = 3.141593
    Stopped scan at "This stopped it"
```
Related Information

- “strtod() - strtof() - strtold() — Convert Character String to Double, Float, and Long Double” on page 422
- “strtod32() - strtod64() - strtod128() — Convert Character String to Decimal Floating-Point” on page 425
- “strtol() – strtoll() — Convert Character String to Long and Long Long Integer” on page 430
- “wcstod() - wcstof() - wcstold() — Convert Wide-Character String to Double, Float, and Long Double” on page 510
- “wcstol() – wcstoll() — Convert Wide Character String to Long and Long Long Integer” on page 515
- “wcstoul() – wcstoull() — Convert Wide Character String to Unsigned Long and Unsigned Long Long Integer” on page 520
- “<wchar.h>” on page 16

wcstok() — Tokenize Wide-Character String

Format

```c
#include <wchar.h>
wchar_t *wcstok(wchar_t *wcs1, const wchar_t *wcs2, wchar_t **ptr);
```

Language Level

ANSI

Threadsafe

Yes

Wide Character Function

See “Wide Characters” on page 568 for more information.

Description

The wcstok() function reads wcs1 as a series of zero or more tokens and wcs2 as the set of wide characters serving as delimiters for the tokens in wcs1. A sequence of calls to the wcstok() function locates the tokens inside wcs1. The tokens can be separated by one or more of the delimiters from wcs2. The third argument points to a wide-character pointer that you provide where the wcstok() function stores information necessary for it to continue scanning the same string.

When the wcstok() function is first called for the wide-character string wcs1, it searches for the first token in wcs1, skipping over leading delimiters. The wcstok() function returns a pointer to the first token. To read the next token from wcs1, call the wcstok() function with NULL as the first parameter (wcs1). This NULL parameter causes the wcstok() function to search for the next token in the previous token string. Each delimiter is replaced by a null character to end the token.

The wcstok() function always stores enough information in the pointer ptr so that subsequent calls, with NULL as the first parameter and the unmodified pointer value as the third, will start searching right after the previously returned token. You can change the set of delimiters (wcs2) from call to call.
Return Value

The `wcstok()` function returns a pointer to the first wide character of the token, or a null pointer if there is no token. In later calls with the same token string, the `wcstok()` function returns a pointer to the next token in the string. When there are no more tokens, the `wcstok()` function returns NULL.

Example

This example uses the `wcstok()` function to locate the tokens in the wide-character string `str1`.

```c
#include <stdio.h>
#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" */
    printf("t = '%ls'\n", t);
    t = wcstok(NULL, L",", &ptr1);    /* t points to the token L"b" */
    printf("t = '%ls'\n", t);
    t = wcstok(str2, L"\t \t", &ptr2); /* t is a null pointer */
    printf("t = '%ls'\n", t);
    t = wcstok(NULL, L"#", &ptr1);    /* t points to the token L"c" */
    printf("t = '%ls'\n", t);
    t = wcstok(NULL, L"?", &ptr2);   /* t is a null pointer */
    printf("t = '%ls'\n", t);
    return 0;
}
```

The output should be similar to:

```
t = 'a'
t = '?b'
t = '
nt = 'c'
t = '
```

Related Information

- “`strtok()` — Tokenize String” on page 428
- “`<wchar.h>`” on page 16

**wcstol() – wcstoll() — Convert Wide Character String to Long and Long Long Integer**

**Format (wcstol())**

```c
#include <wchar.h>
long int wcstol(const wchar_t *nptr, wchar_t **endptr, int base);
```

**Format (wcstoll())**

```c
#include <wchar.h>
long long int wcstoll(const wchar_t *nptr, wchar_t **endptr, int base);
```

**Language Level**

ANSI
Threadsafe
Yes

Locale Sensitive
The behavior of these functions might be affected by the LC_CTYPE category of the current locale if LOCALETYPE(*LOCALE) is specified on the compilation command. The behavior of these functions might also be affected by the LC_UNI_CTYPE category of the current locale if LOCALETYPE(*LOCALEUCS2) or LOCALETYPE(*LOCALEUTF) is specified on the compilation command. These functions are not available when LOCALETYPE(*CLD) is specified on the compilation command. For more information, see “Understanding CCSIDs and Locales” on page 565.

Wide Character Function
See “Wide Characters” on page 568 for more information.

Description
The wcstol() function converts the initial portion of the wide-character string pointed to by nptr to a long integer value. The nptr parameter points to a sequence of wide characters that can be interpreted as a numerical value of type long int. The wcstol() function stops reading the string at the first wide character that it cannot recognize as part of a number. This character can be the wchar_t null character at the end of the string. The ending character can also be the first numeric character greater than or equal to the base.

The wcstoll() function converts a wide-character string to a long long integer. The wide-character string is parsed to skip the initial space characters (as determined by the iswspace function). Any non-space character signifies the start of a subject string that may form a long long int in the radix specified by the base parameter. The subject sequence is defined to be the longest initial substring that is a long long int of the expected form.

If the value of the endptr parameter is not null, then a pointer to the character that ended the scan is stored in endptr. If a long long integer cannot be formed, the value of the endptr parameter is set to that of the nptr parameter.

If the base parameter is a value between 2 and 36, the subject sequence’s expected form is a sequence of letters and digits representing a long long integer whose radix is specified by the base parameter. This sequence optionally is preceded by a positive (+) or negative (-) sign. Letters from a (or A) to z (or Z) inclusive are ascribed the values 10 to 35; only letters whose ascribed values are less than that of the base parameter are permitted. If the base parameter has a value of 16, the characters 0x or 0X optionally precede the sequence of letters and digits, following the positive (+) or negative (-) sign, if present.

If the value of the base parameter is 0, the string determines the base. Therefore, after an optional leading sign, a leading 0 indicates octal conversion, and a leading 0x or 0X indicates hexadecimal conversion.

Return Value
The wcstol() function returns the converted long integer value. If no conversion could be performed, the wcstol() function returns 0. If the correct value is outside the range of representable values, the wcstol() function returns LONG_MAX or LONG_MIN (according to the sign of the value), and sets errno to ERANGE. If the string nptr points to is empty or does not have the expected form, no conversion is performed, and the value of nptr is stored in the object pointed to by endptr, provided that endptr is not a null pointer.

Upon successful completion, the wcstoll() function returns the converted value. If no conversion could be performed, 0 is returned, and the errno global variable is set to indicate the error. If the correct value is outside the range of representable values, the wcstoll() function returns a value of LONG_LONG_MAX or LONG_LONG_MIN.

The value of errno may be set to ERANGE (range error), or EINVAL (invalid argument).
Example
This example uses the \texttt{wcstol()} function to convert the wide-character string \texttt{wcs} to a long integer value.

```c
#include <stdio.h>
#include <wchar.h>

int main(void)
{
    wchar_t *wcs = L"10110134932";
    wchar_t *stopwcs;
    long l;
    int base;

    printf("wcs = \"%ls\"\n", wcs);
    for (base=2; base<=8; base*=2) {
        l = wcstol(wcs, &stop wcs, base);
        printf("   wcstol = %ld\n"
               "   Stopped scan at \"%ls\"\n", l, stop wcs);
    }
    return 0;
}
```

The output should be similar to:

- \texttt{wcs = \"10110134932\"
  wcstol = 45
  Stopped scan at \"34932\"
- \texttt{wcstol = 4423
  Stopped scan at \"4932\"
- \texttt{wcstol = 2134108
  Stopped scan at \"932\"

Related Information
- “\texttt{strtod()} - \texttt{strtof()} - \texttt{strtol()} — Convert Character String to Double, Float, and Long Double” on page 422
- “\texttt{strtod32()} - \texttt{strtod64()} - \texttt{strtod128()} — Convert Character String to Decimal Floating-Point” on page 425
- “\texttt{strtol()} - \texttt{strtoll()} — Convert Character String to Long and Long Long Integer” on page 430
- “\texttt{strtoul()} - \texttt{strtoull()} — Convert Character String to Unsigned Long and Unsigned Long Long Integer” on page 432
- “\texttt{wcstod()} - \texttt{wcstof()} - \texttt{wcstold()} — Convert Wide-Character String to Double, Float, and Long Double” on page 510
- “\texttt{wcstod32()} - \texttt{wcstod64()} - \texttt{wcstod128()} — Convert Wide-Character String to Decimal Floating-Point” on page 512
- “\texttt{wcstoul()} - \texttt{wcstoull()} — Convert Wide Character String to Unsigned Long and Unsigned Long Long Integer” on page 520
- “\texttt{<wchar.h>}” on page 16

\textbf{wcstombs()} — Convert Wide-Character String to Multibyte String

\textbf{Format}

```c
#include <stdlib.h>
size_t wcstombs(char *dest, const wchar_t *string, size_t count);
```

\textbf{Language Level}

ANSI
The behavior of this function might be affected by the LC_CTYPE category of the current locale. The behavior might also be affected by the LC_UNI_CTYPE category of the current locale if LOCALETYPE(*LOCALEUCS2) or LOCALETYPE(*LOCALEUTF) is specified on the compilation command. For more information, see “Understanding CCSIDs and Locales” on page 565.

Wide Character Function
See “Wide Characters” on page 568 for more information.

Description
The wcstombs() function 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 wchar_t null character is encountered.

Only complete multibyte characters are stored in dest. If the lack of space in dest would cause a partial multibyte character to be stored, wcstombs() stores fewer than n bytes and discards the invalid character.

Return Value
The wcstombs() function returns the length in bytes of the multibyte character string, not including a ending null character. The value (size_t)-1 is returned if an invalid multibyte character is encountered.

The value of errno may be set to EILSEQ (conversion stopped due to input character), or ECONVERT (conversion error).

Examples
This program is compiled with LOCALETYPE(*LOCALE) and SYSIFCOPT(*IFSIO):

```c
#include <stdio.h>
#include <stdlib.h>
#include <locale.h>
#include <wchar.h>

#define STRLENGTH 10
#define LOCNAM "/qsys.lib/JA_JP.locale"
#define LOCNAM_EN "/qsys.lib/EN_US.locale"

int main(void) {
    char string[STRLENGTH];
    int length, sl = 0;
    wchar_t wc2[] = L"ABC";
    wchar_t wc_string[10];
    mbstate_t ps = 0;
    memset(string, '\0', STRLENGTH);
    wc_string[0] = 0x00C1;
    wc_string[1] = 0x4171;
    wc_string[2] = 0x4172;
    wc_string[3] = 0x00C2;
    wc_string[4] = 0x0000;

    /* In this first example we will convert a wide character string */
    /* to a single byte character string. We first set the locale */
    /* to a single byte locale. We choose a locale with */
    /* CCSID 37. */
    if (setlocale(LC_ALL, LOCNAM_EN) == NULL)
        printf("setlocale failed.\n");
    length = wcstombs(string, wc2, 10);
    return 0;
}
```
/* In this case wide characters ABC are converted to */
/* single byte characters ABC, length is 3. */

printf("string = %s, length = %d\n\n", string, length);

/* Now lets try a multibyte example. We first must set the */
/* locale to a multibyte locale. We choose a locale with */
/* CCSID 5026 */

if (setlocale(LC_ALL, LOCNAME) == NULL)
  printf("setlocale failed.\n\n");

length = wcstombs(string, wc_string, 10);

/* The hex look at string would now be: */
/* C10E417141720FC2 length will be 8 */
/* You would need a device capable of displaying multibyte */
/* characters to see this string. */

printf("length = %d\n\n", length);

} /* The output should look like this:
string = ABC, length = 3
length = 8 */

This program is compiled with LOCALETYPE(*LOCALEUCS2) and SYSIFCOPT(*IFSIO):

#include <stdio.h>
#include <stdlib.h>
#include <locale.h>
#include <wchar.h>

#define STRLENGTH 10
#define LOCNAME "/qsys.lib/JA_JP.locale"
#define LOCNAME_EN "/qsys.lib/EN_US.locale"

int main(void)
{
  char string[STRLENGTH];
  int length, sl = 0;
  wchar_t wc2[] = L"ABC";
  wchar_t wc_string[10];
  mbstate_t ps = 0;
  memset(string, '\0', STRLENGTH);
  wc_string[0] = 0x0041;       /* UNICODE A */
  wc_string[1] = 0xFF41;
  wc_string[2] = 0xFF42;       /* UNICODE B */
  wc_string[3] = 0x0000;
  wc_string[4] = 0x0000;
  /* In this first example we will convert a wide character string */
  /* to a single byte character string. We first set the locale */
  /* to a single byte locale. We choose a locale with */
  /* CCSID 37. */

  if (setlocale(LC_ALL, LOCNAME_EN) == NULL)
    printf("setlocale failed.\n\n");

  length = wcstombs(string, wc2, 10);

  /* In this case wide characters ABC are converted to */
  /* single byte characters ABC, length is 3. */
  printf("string = %s, length = %d\n\n", string, length);

  /* Now lets try a multibyte example. We first must set the */
  /* locale to a multibyte locale. We choose a locale with */
  /* CCSID 5026 */

  if (setlocale(LC_ALL, LOCNAME) == NULL)
    printf("setlocale failed.\n\n");

  length = wcstombs(string, wc_string, 10);
/* The hex look at string would now be: */
/* C10E428142820FC2   length will be 8 */
/* You would need a device capable of displaying multibyte */
/* characters to see this string. */

printf("length = %d\n\n", length);

} /* The output should look like this:
string = ABC, length = 3
length = 8 */

Related Information

• “mbstowcs() — Convert a Multibyte String to a Wide Character String” on page 229
• “wcslen() — Calculate Length of Wide-Character String” on page 494
• “wcsrtombs() — Convert Wide Character String to Multibyte String (Restartable)” on page 505
• “wctomb() — Convert Wide Character to Multibyte Character” on page 527
• “<stdlib.h>” on page 15

wcstoul() – wcstoull() — Convert Wide Character String to Unsigned Long and Unsigned Long Long Integer

Format (wcstoul())

#include <wchar.h>
unsigned long int wcstoul(const wchar_t *nptr, wchar_t **endptr, int base);

Format (wcstoull())

#include <wchar.h>
unsigned long long int wcstoull(const wchar_t *nptr, wchar_t **endptr, int base);

Language Level

ANSI

Threadsafe

Yes

Locale Sensitive

The behavior of these functions might be affected by the LC_CTYPE category of the current locale if LOCALETYPE(*LOCALE) is specified on the compilation command. The behavior of these functions might also be affected by the LC_UNI_CTYPE category of the current locale if LOCALETYPE(*LOCALEUCS2) or LOCALETYPE(*LOCALEUTF) is specified on the compilation command. These functions are not available when LOCALETYPE(*CLD) is specified on the compilation command. For more information, see “Understanding CCSIDs and Locales” on page 565.

Wide Character Function

See “Wide Characters” on page 568 for more information.
The `wcstoul()` function converts the initial portion of the wide-character string pointed to by `nptr` to an unsigned long integer value. The `nptr` parameter points to a sequence of wide characters that can be interpreted as a numerical value of type unsigned long int. The `wcstoul()` function stops reading the string at the first wide character that it cannot recognize as part of a number. This character can be the wchar_t null character at the end of the string. The ending character can also be the first numeric character greater than or equal to the base.

The `wcstoull()` function converts a wide-character string to an unsigned long long integer. The wide-character string is parsed to skip the initial space characters (as determined by the iswspace function). Any non-space character signifies the start of a subject string that may form an unsigned long long int in the radix specified by the `base` parameter. The subject sequence is defined to be the longest initial substring that is an unsigned long long int of the expected form.

If the value of the `endptr` parameter is not null, then a pointer to the character that ended the scan is stored in `endptr`. If an unsigned long long integer cannot be formed, the value of the `endptr` parameter is set to that of the `nptr` parameter.

If the `base` parameter is a value between 2 and 36, the subject sequence’s expected form is a sequence of letters and digits representing an unsigned long long integer whose radix is specified by the `base` parameter. This sequence optionally is preceded by a positive (+) or negative (-) sign. Letters from a (or A) to z (or Z) inclusive are ascribed the values 10 to 35; only letters whose ascribed values are less than that of the `base` parameter are permitted. If the `base` parameter has a value of 16, the characters 0x or 0X optionally precede the sequence of letters and digits, following the positive (+) or negative (-) sign, if present.

If the value of the `base` parameter is 0, the string determines the base. Therefore, after an optional leading sign, a leading 0 indicates octal conversion, and a leading 0x or 0X indicates hexadecimal conversion.

The value of `errno` may be set to `EINVAL` (endptr is null, no numbers are found, or base is invalid), or `ERANGE` (converted value is outside the range).

**Return Value**

The `wcstoul()` function returns the converted unsigned long integer value. If no conversion could be performed, the `wcstoul()` function returns 0. If the correct value is outside the range of representable values, the `wcstoul()` function returns ULONG_MAX and sets `errno` to `ERANGE`. If the string `nptr` points to is empty or does not have the expected form, no conversion is performed, and the value of `nptr` is stored in the object pointed to by `endptr`, provided that `endptr` is not a null pointer.

Upon successful completion, the `wcstoull()` function returns the converted value. If no conversion could be performed, 0 is returned, and the `errno` global variable is set to indicate the error. If the correct value is outside the range of representable values, `wcstoull()` function returns a value of ULONG_LONG_MAX.

**Example**

This example uses the `wcstoul()` function to convert the string `wcs` to an unsigned long integer value.

```c
#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"
           "The unsigned long int produced: %lu\n"
           "The substring of the input wide string that was not"
```
Related Information

- “strtod() - strtof() - strtold() — Convert Character String to Double, Float, and Long Double” on page 422
- “strtod32() - strtod64() - strtod128() — Convert Character String to Decimal Floating-Point” on page 425
- “strtol() – strtoll() — Convert Character String to Long and Long Long Integer” on page 430
- “wcstod() - wcstof() - wcstold() — Convert Wide-Character String to Double, Float, and Long Double” on page 510
- “wcstod32() - wcstod64() - wcstod128() — Convert Wide-Character String to Decimal Floating-Point” on page 512
- “wcstol() – wcstoll() — Convert Wide Character String to Long and Long Long Integer” on page 515
- “<wchar.h>” on page 16

wcswcs() — Locate Wide-Character Substring

Format

```
#include <wchar.h>
wchar_t *wcswcs(const wchar_t *string1, const wchar_t *string2);
```

Language Level

XPG4

Threading

Yes

Wide Character Function

See “Wide Characters” on page 568 for more information.

Description

The wcswcs() function locates the first occurrence of string2 in the wide-character string pointed to by string1. In the matching process, the wcswcs() function ignores the wchar_t null character that ends string2.

Return Value

The wcswcs() function returns a pointer to the located string or NULL if the string is not found. If string2 points to a string with zero length, wcswcs() returns string1.

Example

This example finds the first occurrence of the wide character string pr in buffer1.
```c
#include <stdio.h>
#include <wchar.h>

#define SIZE 40

int main(void)
{
    wchar_t buffer1[SIZE] = L"computer program";
    wchar_t * ptr;
    wchar_t * wch = L"pr";

    ptr = wcscws( buffer1, wch );
    printf( "The first occurrence of %ls in '%ls' is '%ls'
               wch, buffer1, ptr );
}
/******************** Output should be similar to: ******************
The first occurrence of pr in 'computer program' is 'program' */
```

**Related Information**

- “strchr() — Search for Character” on page 386
- “strcspn() — Find Offset of First Character Match” on page 393
- “strchr() — Find Characters in String” on page 413
- “strchr() — Locate Last Occurrence of Character in String” on page 418
- “strspn() — Find Offset of First Non-matching Character” on page 419
- “strstr() — Locate Substring” on page 421
- “wcschr() — Search for Wide Character” on page 484
- “wcsncmp() — Compare Wide-Character Strings” on page 485
- “wcscspn() — Find Offset of First Wide-Character Match” on page 489
- “wcsbrk() — Locate Wide Characters in String” on page 501
- “wcsrchr() — Locate Last Occurrence of Wide Character in String” on page 504
- “wcsspn() — Find Offset of First Non-matching Wide Character” on page 507
- “<wchar.h>” on page 16

### wcswidth() — Determine the Display Width of a Wide Character String

**Format**

```c
#include <wchar.h>
int wcswidth (const wchar_t *wcs, size_t n);
```

**Language Level**

XPG4

**Threadsafe**

Yes

**Locale Sensitive**

The behavior of this function might be affected by the LC_CTYPE category of the current locale if LOCALETYPE("getLocale") is specified on the compilation command. The behavior of this function might also be affected by the LC_UNI_CTYPE category of the current locale if LOCALETYPE("getLocaleUCS2") or LOCALETYPE("getLocaleUTF") is specified on the compilation command. This function is not available when
LOCALETYPE(*CLD) is specified on the compilation command. For more information, see “Understanding CCSIDs and Locales” on page 565.

Wide Character Function
See “Wide Characters” on page 568 for more information.

Description
The wcswidth() function 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 string pointed to by wcs occupies on a display device. The number is independent of its location on the device.

The value of \( \text{errno} \) may be set to \textbf{EINVAL} (non-printing wide character).

Return Value
The wcswidth() function either returns:

- 0, if wcs points to a null wide character; or
- the number of printing positions occupied by the wide string pointed to by wcs; or
- -1, if any wide character in the wide string pointed to by wcs is not a printing wide character.

Example
```c
#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));
}
```

/*
 *            wcs has a width of: 3
 */

Related Information
- “wcswidth() — Determine the Display Width of a Wide Character String” on page 523
- “<wchar.h>” on page 16

\textbf{wcxfrm() — Transform a Wide-Character String}

Format
```c
#include <wchar.h>
size_t wcxfrm (wchar_t *wcs1, const wchar_t *wcs2, size_t n);
```

Language Level
XPG4

Threading Safe
Yes
Locale Sensitive

The behavior of this function might be affected by the LC_COLLATE category of the current locale if LOCALETYPE(*LOCALE) is specified on the compilation command. The behavior of this function might also be affected by the LC_COLLATE category of the current locale if LOCALETYPE(*LC_COLLATE_) is specified on the compilation command. This function is not supported when LOCALETYPE(*LOCALEUCS2) is specified on the compilation command. This function is not available when LOCALETYPE(*CLD) is specified on the compilation command. For more information, see “Understanding CCSIDs and Locales” on page 565.

Wide Character Function

See “Wide Characters” on page 568 for more information.

Description

The wcsxfrm() function 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.

Return Value

The wcsxfrm() function returns the length of the transformed wide-character string (not including the ending null wide character code). If the value returned is n or more, the contents of the array pointed to by wcs1 are indeterminate.

If wcsxfrm() is unsuccessful, errno is changed. The value of errno may be set to EINVAL (the wcs1 or wcs2 arguments contain characters which are not available in the current locale).

Example

```c
#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 = fgets(buffer, 80, stdin);
    length = wcsxfrm(NULL, wcs, 0);
    printf("You would need a %d element array to hold the wide string\n", length);
    printf("translated according to this program’s locale. \n");
}
```

Related Information

• “strxfrm() — Transform String” on page 434
• “<wchar.h>” on page 16

wctob() — Convert Wide Character to Byte

Format

```c
#include <stdio.h>
#include <wchar.h>
int wctob(wint_t wc);
```
Locale Sensitive

The behavior of this function might be affected by the LC_CTYPE category of the current locale. The behavior might also be affected by the LC_UNI_CTYPE category of the current locale if LOCALETYPE(LLOCALEUCS2) or LOCALETYPE(LLOCALEUTF) is specified on the compilation command. This function is not available when LOCALETYPE(LCLD) is specified on the compilation command. For more information, see “Understanding CCSIDs and Locales” on page 565.

Wide Character Function

See “Wide Characters” on page 568 for more information.

Description

The wctob() function determines whether wc corresponds to a member of the extended character set, whose multibyte character has a length of 1 byte when in the initial shift state.

Return Value

If c corresponds to a multibyte character with a length of 1 byte, the wctob() function returns the single-byte representation. Otherwise, it returns EOF.

If a conversion error occurs, errno may be set to ECONVERT.

Example

This example uses the wctob() function to test if the wide character A is a valid single-byte character.

```c
#include <stdio.h>
#include <wchar.h>

int main(void)
{
    wint_t wc = L'A';
    if (wctob(wc) == wc)
        printf("%lc is a valid single byte character\n", wc);
    else
        printf("%lc is not a valid single byte character\n", wc);
    return 0;
}
```

The output should be similar to:

```
A is a valid single byte character
```

Related Information

- “mbtowc() — Convert Multibyte Character to a Wide Character” on page 233
- “wctomb() — Convert Wide Character to Multibyte Character” on page 527
- “wcstombs() — Convert Wide-Character String to Multibyte String” on page 517
- “<wchar.h>” on page 16
wctomb() — Convert Wide Character to Multibyte Character

Format

```c
#include <stdlib.h>
int wctomb(char *string, wchar_t character);
```

Language Level

ANSI

Threadsafe

No

Use wcrtomb() instead.

Locale Sensitive

The behavior of this function might be affected by the LC_CTYPE category of the current locale. The behavior might also be affected by the LC_UNI_CTYPE category of the current locale if LOCALETYPE(*LOCALEUCS2) or LOCALETYPE(*LOCALEUTF) is specified on the compilation command. For more information, see “Understanding CCSIDs and Locales” on page 565.

Wide Character Function

See “Wide Characters” on page 568 for more information.

Description

The `wctomb()` function 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, the `wctomb()` function stores MB_CUR_MAX characters in `string`.

The conversion of the wide character is the same as described in `wcstombs()`. See this function for a Unicode example.

Return Value

The `wctomb()` function returns the length in bytes of the multibyte character. The value -1 is returned if `character` is not a valid multibyte character. If `string` is a NULL pointer, the `wctomb()` function returns nonzero if shift-dependent encoding is used, or 0 otherwise.

If a conversion error occurs, `errno` may be set to `ECONVERT`.

Example

This example converts the wide character c to a multibyte character.
```c
#include <stdio.h>
#include <stdlib.h>
#include <wchar.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 should be similar to: ******************
The number of bytes that comprise the multibyte character is 1
And the converted string is "c"
*/
```

**Related Information**

- “mbtowc() — Convert Multibyte Character to a Wide Character” on page 233
- “wcslen() — Calculate Length of Wide-Character String” on page 494
- “wcrtomb() — Convert a Wide Character to a Multibyte Character (Restartable)” on page 478
- “wcstombs() — Convert Wide-Character String to Multibyte String” on page 505
- “wcsrtombs() — Convert Wide Character String to Multibyte String (Restartable)” on page 505
- “<stdlib.h>” on page 15

**wctrans() — Get Handle for Character Mapping**

**Format**

```c
#include <wctype.h>
wctrans_t wctrans(const char *property);
```

**Language Level**

ANSI

**Threadsafe**

Yes

**Description**

The `wctrans()` function returns a value with type `wctrans_t`. This value describes a mapping between wide characters. The string argument `property` is a wide character mapping name. The `wctrans_t` equivalent of the wide character mapping name is returned by this function. The `toupper` and `tolower` wide character mapping names are defined in all locales.

**Return Value**

If `property` is a valid wide character mapping name, the `wctrans()` function returns a nonzero value that is valid as the second argument to the `towctrans()` function. Otherwise, it returns 0.
**Example**

This example translates the lowercase alphabet to uppercase, and back to lowercase.

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <wchar.h>
#include <wctype.h>

int main() {
    char *alpha = "abcdefghijklmnopqrstuvwxyz";
    char *tocase[2] = {"toupper", "tolower"};
    wchar_t *wcalpha;
    int i, j;
    size_t alphalen;
    alphalen = strlen(alpha) + 1;
    wcalpha = (wchar_t *)malloc(sizeof(wchar_t) * alphalen);
    mbstowcs(wcalpha, alpha, 2 * alphalen);
    for (i = 0; i < 2; ++i) {
        printf("Input string: %ls\n", wcalpha);
        for (j = 0; j < strlen(alpha); ++j) {
            wcalpha[j] = (wchar_t)towctrans((wint_t)wcalpha[j], wctrans(tocase[i]));
        }
        printf("Output string: %ls\n", wcalpha);
        printf("\n");
    }
    return 0;
}

/****************  Output should be similar to:  ******************
Input string: abcdefghijklmnopqrstuvwxyz
Output string: ABCDEFGHIJKLMNOPQRSTUVWXYZ
Input string: ABCDEFGHIJKLMNOPQRSTUVWXYZ
Output string: abcdefghijklmnopqrstuvwxyz
*******************************************************************/
```

**Related Information**

- “towctrans() — Translate Wide Character” on page 448
- “<wctype.h>” on page 17

**wctype() — Get Handle for Character Property Classification**

**Format**

```c
#include <wctype.h>
wctype_t wctype(const char *property);
```

**Language Level**

XPG4

**Threading safe**

Yes

**Description**

The `wctype()` function is defined for valid character class names. The `property` is a string that identifies a generic character class. These character class names are defined in all locales: `alnum`, `alpha`, `blank`, `cntrl`, `graph`, `lower`, `print`, `shx`, `space`, `upper`, `xdigit`, `xdigit`. This function returns a handle that can be used by other wide-character functions to perform operations on characters. For example, `towctrans()` can be used with the handle returned by `wctype()` to translate characters from one class to another.
cntrl, digit, graph, lower, print, punct, space, upper, xdigit. The function returns a value of
type wctype_t, which can be used as the second argument to a call of the iswctype() function.

The wctype() function determines values of wctype_t according to rules of the coded character set that
are defined by character type information in the program's locale (category LC_CTYPE). Values that are
returned by the wctype() are valid until a call to setlocale() that changes the category LC_CTYPE.

Return Value
The wctype() function returns zero if the given property name is not valid. Otherwise, it returns a value
of type wctype_t that can be used in calls to iswctype().

Example

```c
#include <wchar.h>
#define UPPER_LIMIT 0xFF
int main(void)
{
  int wc;
  for (wc = 0; wc <= UPPER_LIMIT; wc++) {
    printf("%#4x ", wc);
    printf("%c", iswctype(wc, wctype("print")) ? wc : ' ');
    printf("%c", iswctype(wc, wctype("alnum")) ? "AN" : " ");
    printf("%c", iswctype(wc, wctype("alpha")) ? "A" : " ");
    printf("%c", iswctype(wc, wctype("blank")) ? "B" : " ");
    printf("%c", iswctype(wc, wctype("cntrl")) ? "C" : " ");
    printf("%c", iswctype(wc, wctype("digit")) ? "D" : " ");
    printf("%c", iswctype(wc, wctype("graph")) ? "G" : " ");
    printf("%c", iswctype(wc, wctype("lower")) ? "L" : " ");
    printf("%c", iswctype(wc, wctype("punct")) ? "PU" : " ");
    printf("%c", iswctype(wc, wctype("space")) ? "S" : " ");
    printf("%c", iswctype(wc, wctype("print")) ? "PR" : " ");
    printf("%c", iswctype(wc, wctype("upper")) ? "U" : " ");
    printf("%c", iswctype(wc, wctype("xdigit")) ? "X" : " ");
    putwchar('n');
  }
  return 0;
}
/* *************************************************************************/
The output should be similar to :
0x1f C
0x20 B S PR
0x21 ! G PU PR
0x22 " G PU PR
0x23 # G PU PR
0x24 $ G PU PR
0x25 % G PU PR
0x26 & G PU PR
0x27 ' G PU PR
0x28 ( G PU PR
0x29 ) G PU PR
0x2a * G PU PR
0x2b + G PU PR
0x2c , G PU PR
0x2d - G PU PR
0x2e / G PU PR
0x30 0 AN D G PR X
0x31 1 AN D G PR X
0x32 2 AN D G PR X
0x33 3 AN D G PR X
0x34 4 AN D G PR X
0x35 5 AN D G PR X
:* ***********************************************************************/

Related Information
- “<wchar.h>” on page 16
- “<wctype.h>” on page 17


**Format**

```c
#include <wchar.h>
int wcwidth (const wint_t wc);
```

**Language Level**

XPG4

**Threadsafe**

Yes

**Locale Sensitive**

The behavior of this function might be affected by the LC_CTYPE category of the current locale. The behavior might also be affected by the LC_UNI_CTYPE category of the current locale if LOCALETYPE("LOCALEUCS2") or LOCALETYPE("LOCALEUTF") is specified on the compilation command. This function is not available when LOCALETYPE("CLD") is specified on the compilation command. For more information, see “Understanding CCSIDs and Locales” on page 565.

**Wide Character Function**

See “Wide Characters” on page 568 for more information.

**Description**

The `wcwidth()` function 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.

The value of `errno` may be set to `EINVAL` (non-printing wide character).

**Return Value**

The `wcwidth()` function either returns:

- 0, if `wc` is a null wide character; or
- the number of printing position occupied by `wc`; or
- -1, if `wc` is not a printing wide character.

**Example**

```c
#include <stdio.h>
#include <wchar.h>

int main(void)
{
    wint_t wc = L'A';
    printf("%lc has a width of %d\n", wc, wcwidth(wc));
    return 0;
}

/**************************************************************************
The output should be similar to :
A has a width of 1
**************************************************************************/
```
wfopen() — Open Files

Format

```c
#include <ifs.h>
FILE * wfopen(const wchar_t *filename,const wchar_t *mode);
```

Language Level

ILE C Extension

Threading

Yes

Locale Sensitive

This function is only available when LOCALETYPE(*LOCALEUCS2) or LOCALETYPE(*LOCALEUTF) is specified on the compilation command. For more information, see “Understanding CCSIDs and Locales” on page 565.

Integrated File System Interface

This function is not available when SYSIFCOPT(*NOIFSIO) is specified on the compilation command.

Wide Character Function

See “Wide Characters” on page 568 for more information.

Description

The `wfopen()` function works like the `fopen()` function, except:

- `wfopen()` accepts file name and mode as wide characters.
- The default CCSID for files opened with `wfopen()` (used when the ccsid=value, o_ccsid=value, and codepage=value keywords are not specified) is UCS2 when LOCALETYPE(*LOCALEUCS2) is specified on the compilation command. The default CCSID is UTF-32 when LOCALETYPE(*LOCALEUTF) is specified on the compilation command.

wmemchr() — Locate Wide Character in Wide-Character Buffer

Format

```c
#include <wchar.h>
wchar_t * wmemchr(const wchar_t *s, wchar_t c, size_t n);
```

Language Level

ANSI
Threadsafe
Yes

Wide Character Function
See “Wide Characters” on page 568 for more information.

Description
The wmemchr() function locates the first occurrence of c in the initial n wide characters of the object pointed to by s. If n has the value 0, the wmemchr() function finds no occurrence of c, and returns a NULL pointer.

Return Value
The wmemchr() function returns a pointer to the located wide character, or a NULL pointer if the wide character does not occur in the object.

Example
This example finds the first occurrence of 'A' in the wide-character string.

```c
#include <stdio.h>
#include <wchar.h>

main()
{
    wchar_t *in = L"1234ABCD";
    wchar_t *ptr;
    wchar_t fnd = L'A';
    printf("nEXPECTED: ABCD\n");
    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
- “memchr() — Search Buffer” on page 234
- “strchr() — Search for Character” on page 386
- “wcschr() — Search for Wide Character” on page 484
- “wmemcmp() — Compare Wide-Character Buffers” on page 533
- “wmemcpy() — Copy Wide-Character Buffer” on page 535
- “wmemmove() — Copy Wide-Character Buffer” on page 536
- “wmemset() — Set Wide Character Buffer to a Value” on page 537
- “<wchar.h>” on page 16

wmemcmp() — Compare Wide-Character Buffers

Format

```c
#include <wchar.h>
int wmemcmp(const wchar_t *s1, const wchar_t *s2, size_t n);
```
**Wide Character Function**

See “Wide Characters” on page 568 for more information.

**Description**

The `wmemcmp()` function compares the first \( n \) wide characters of the object pointed to by `s1` to the first \( n \) wide characters of the object pointed to by `s2`. If \( n \) has the value 0, the `wmemcmp()` function returns 0.

**Return Value**

The `wmemcmp()` function returns a value according to the relationship between the two strings, `s1` and `s2`:

<table>
<thead>
<tr>
<th>Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>Less than 0</td>
<td><code>s1</code> less than <code>s2</code></td>
</tr>
<tr>
<td>0</td>
<td><code>s1</code> identical to <code>s2</code></td>
</tr>
<tr>
<td>Greater than 0</td>
<td><code>s1</code> greater than <code>s2</code></td>
</tr>
</tbody>
</table>

**Example**

This example compares the wide-character string in to out using the `wmemcmp()` function.

```c
#include <wchar.h>
#include <stdio.h>
#include <locale.h>

main()
{
  int rc;
  wchar_t *in = L"12345678";
  wchar_t *out = L"12AAAAAB";
  setlocale(LC_ALL, "POSIX");

  printf("\nGREATER is the expected result\n");
  rc = wmemcmp(in, out, 3);
  if (rc == 0)
    printf("\nArrays are EQUAL %ls %ls \n", in, out);
  else
    {
      if (rc > 0)
        printf("\nArray %ls GREATER than %ls \n", in, out);
      else
        printf("\nArray %ls LESS than %ls \n", in, out);
    }
}

/*****************************************************/
GREATER is the expected result
Array 12345678 GREATER than 12AAAAAB
*****************************************************/
```

**Related Information**

- “`memcmp()` — Compare Buffers” on page 235
wmemcpy() — Copy Wide-Character Buffer

**Format**

```c
#include <wchar.h>
wchar_t *wmemcpy(wchar_t *s1, const wchar_t *s2, size_t n);
```

**Language Level**

ANSI

**Threadsafe**

Yes

**Wide Character Function**

See “Wide Characters” on page 568 for more information.

**Description**

The `wmemcpy()` function copies `n` wide characters from the object pointed to by `s2` to the object pointed to by `s1`. If `s1` and `s2` overlap, the result of the copy is unpredictable. If `n` has the value 0, the `wmemcpy()` function copies 0 wide characters.

**Return Value**

The `wmemcpy()` function returns the value of `s1`.

**Example**

This example copies the first four characters from `out` to `in`. In the expected output, the first four characters in both strings will be "ABCD".
```c
#include <wchar.h>
#include <stdio.h>

main()
{
    wchar_t *in = L"12345678";
    wchar_t *out = L"ABCDEFGH";
    wchar_t *ptr;

    printf("Expected result: First 4 chars of in change and are the same as first 4 chars of out");
    ptr = wmemcpy(in, out, 4);
    if (ptr == in)
        printf("Array in %ls array out %ls \n", in, out);
    else
    {
        printf("n*** ERROR ***");
        printf(" returned pointer wrong");
    }
}
```

**Related Information**

- “memcpy() — Copy Bytes” on page 236
- “strcpy() — Copy Strings” on page 392
- “strncpy() — Copy Strings” on page 409
- “wcscpy() — Copy Wide-Character Strings” on page 488
- “wcsncpy() — Copy Wide-Character Strings” on page 499
- “wmemchr() — Locate Wide Character in Wide-Character Buffer” on page 532
- “wmemcmp() — Compare Wide-Character Buffers” on page 533
- “wmemmove() — Copy Wide-Character Buffer” on page 536
- “wmemset() — Set Wide Character Buffer to a Value” on page 537
- “<wchar.h>” on page 16

**wmemmove() — Copy Wide-Character Buffer**

**Format**

```
#include <wchar.h>
wchar_t *wmemmove(wchar_t *s1, const wchar_t *s2, size_t n);
```

**Language Level**

ANSI

**Threadsafe**

Yes

**Wide Character Function**

See “Wide Characters” on page 568 for more information.

**Description**

The `wmemmove()` function copies `n` wide characters from the object pointed to by `s2` to the object pointed to by `s1`. Copying takes place as if the `n` wide characters from the object pointed to by `s2` are first copied into a temporary array, of `n` wide characters, that does not overlap the objects pointed to by `s1` or `s2`. 

536  IBM i: ILE C/C++ Runtime Library Functions
Then, the `wmemmove()` function copies the \( n \) wide characters from the temporary array into the object pointed to by \( s1 \). If \( n \) has the value 0, the `wmemmove()` function copies 0 wide characters.

**Return Value**
The `wmemmove()` function returns the value of \( s1 \).

**Example**
This example copies the first five characters in a string to overlay the last five characters in the same string. Since the string is only nine characters long, the source and target overlap.

```c
#include <wchar.h>
#include <stdio.h>

void main()
{
    wchar_t *theString = L"ABCDEFGHI";
    printf("\nThe original string: %ls \n", theString);
    wmemmove(theString+4, theString, 5);
    printf("\nThe string after wmemmove: %ls \n", theString);
    return;
}

/******************************************************************************
The output should be:
The original string: ABCDEFGHI
The string after wmemmove: ABCDABCDE
*******************************************************************************/
```

**Related Information**
- “memmove() — Copy Bytes” on page 239
- “wmemchr() — Locate Wide Character in Wide-Character Buffer” on page 532
- “wmemcpy() — Copy Wide-Character Buffer” on page 535
- “wmemcmp() — Compare Wide-Character Buffers” on page 533
- “wmemset() — Set Wide Character Buffer to a Value” on page 537
- “<wchar.h>” on page 16

**`wmemset()` — Set Wide Character Buffer to a Value**

**Format**

```c
#include <wchar.h>
wchar_t *wmemset(wchar_t *s, wchar_t c, size_t n);
```

**Language Level**
ANSI

**Threading**
Yes

**Wide Character Function**
See “Wide Characters” on page 568 for more information.
Description
The wmemset() function copies the value of c into each of the first n wide characters of the object pointed to by s. If n has the value 0, the wmemset() function copies 0 wide characters.

Return Value
The wmemset() function returns the value of s.

Example
This example sets the first 6 wide characters to the wide character 'A'.

```c
#include <wchar.h>
#include <stdio.h>

void main()
{
    wchar_t *in = L"1234ABCD";
    wchar_t *ptr;
    printf("\nEXPECTED: AAAAAACD\n");
    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
• “memset() — Set Bytes to Value” on page 240
• “wmemchr() — Locate Wide Character in Wide-Character Buffer” on page 532
• “wmemcpy() — Copy Wide-Character Buffer” on page 535
• “wmemcmp() — Compare Wide-Character Buffers” on page 533
• “wmemmove() — Copy Wide-Character Buffer” on page 536
• “<wchar.h>” on page 16

wprintf() — Format Data as Wide Characters and Print

Format
```c
#include <stdio.h>
int wprintf(const wchar_t *format,...);
```

Language Level
ANSI

Threadsafe
Yes

Locale Sensitive
The behavior of this function might be affected by the LC_CTYPE and LC_NUMERIC categories of the current locale. The behavior might also be affected by the LC_UNI_CTYPE and LC_UNI_NUMERIC categories of the current locale if LOCALETYPE(*LOCALEUCS2) or LOCALETYPE(*LOCALEUTF) is specified.
on the compilation command. This function is not available when LOCALETYPE(*CLD) is specified on the
compilation command. For more information, see “Understanding CCSIDs and Locales” on page 565.

Integrated File System Interface
This function is not available when SYSIFCOPT(*NOIFSIO) is specified on the compilation command.

Wide Character Function
See “Wide Characters” on page 568 for more information.

Description
A wprintf(format, ...) is equivalent to fwprintf(stdout, format, ...).

Return Value
The wprintf() function returns the number of wide characters transmitted. If an output error occurred,
the wprintf() function returns a negative value.

Example
This example prints the wide character a. Date and time may be formatted according to your locale's
representation. The output goes to stdout.

```c
#include <wchar.h>
#include <stdarg.h>
#include <locale.h>

int main(void)
{
    setlocale(LC_ALL, "POSIX");
    wprintf (L"%C
", L'a');
    return(0);
}
/* A long 'a' is written to stdout */
```

Related Information
• “printf() — Print Formatted Characters” on page 251
• “btowc() — Convert Single Byte to Wide Character” on page 78
• “mbrtowc() — Convert a Multibyte Character to a Wide Character (Restartable)” on page 223
• “vfwprintf() — Format Argument Data as Wide Characters and Write to a Stream ” on page 460
• “fwprintf() — Format Data as Wide Characters and Write to a Stream” on page 165
• “vswprintf() — Format and Write Wide Characters to Buffer” on page 471
• “<wchar.h>” on page 16

wscanf() — Read Data Using Wide-Character Format String

Format

```
#include <stdio.h>
int wscanf(const wchar_t *format,...);
```

Language Level
ANSI
Threadsafe
Yes

Locale Sensitive
The behavior of this function might be affected by the LC_CTYPE and LC_NUMERIC categories of the current locale. The behavior might also be affected by the LC_UNI_CTYPE and LC_UNI_NUMERIC categories of the current locale if LOCALETYPE(*LOCALEUCS2) or LOCALETYPE(*LOCALEUTF) is specified on the compilation command. This function is not available when LOCALETYPE(*CLD) is specified on the compilation command. For more information, see “Understanding CCSIDs and Locales” on page 565.

Integrated File System Interface
This function is not available when SYSIFCOPT(*NOIFSIO) is specified on the compilation command.

Wide Character Function
See “Wide Characters” on page 568 for more information.

Description
The wscanf() function is equivalent to the fscanf() function with the argument stdin interposed before the arguments of the wscanf() function.

Return Value
If an input failure occurs before any conversion, the wscanf() function returns the value of the macro EOF.

Otherwise, the wscanf() function returns the number of input items assigned. It can be fewer than provided for, or even zero, in the event of an early matching failure.

Example
This example scans various types of data.

```c
#include <stdio.h>
#include <wchar.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 (wscanf(L"%d %f %c %s", &i, &fp,&c, s) != 4)
        printf("Some fields were not assigned\n");
    else {
        printf("integer = %d\n", i);
        printf("real number = %f\n", fp);
        printf("character = %c\n", c);
        printf("string = %s\n", s);
    }
    return 0;
}

/***************************************************************************/
The output should be similar to:
Enter an integer, a real number, a character and a string :
12 2.5 a yes
integer = 12
real number = 2.500000
character = a
string = yes
/***************************************************************************/
Related Information

- “fscanf() — Read Formatted Data” on page 155
- “fwprintf() — Format Data as Wide Characters and Write to a Stream” on page 165
- “fwsprintf() — Read Data from Stream Using Wide Character” on page 169
- “scanf() — Read Data” on page 358
- “sscanf() — Read Data” on page 382
- “swprintf() — Format and Write Wide Characters to Buffer” on page 435
- “swscanf() — Read Wide Character Data” on page 437
- “vfscanf() — Read Formatted Data” on page 458
- “vfwsprintf() — Read Formatted Wide Character Data” on page 462
- “vscanf() — Read Formatted Data” on page 465
- “vsscanf() — Read Formatted Data” on page 470
- “vswscanf() — Read Formatted Wide Character Data” on page 473
- “vwscanf() — Read Formatted Wide Character Data” on page 476
- “wprintf() — Format Data as Wide Characters and Print” on page 538
- “<wchar.h>” on page 16
Runtime Considerations

This topic provides the following information:

- Exception and condition management
- Interlanguage data type compatibility
- CCSID (Coded Character Set Identifier) source file conversion
- Heap memory

errno Macros

The following table lists which error macros the ILE C library functions can set.

<table>
<thead>
<tr>
<th>Error Macro</th>
<th>Description</th>
<th>Set by Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>EBADDATA</td>
<td>The message data is not valid.</td>
<td>perror, strerror</td>
</tr>
<tr>
<td>EBADF</td>
<td>The catalog descriptor is not valid.</td>
<td>catclose, catgets, clearerr, fgetc, fgetpos, fgets, fileno, freopen, fseek, fsetpos, getc, rewind</td>
</tr>
<tr>
<td>EBADKEYLN</td>
<td>The key length specified is not valid.</td>
<td>_Rreadk, _Rlocate</td>
</tr>
<tr>
<td>EBADMODE</td>
<td>The file mode specified is not valid.</td>
<td>fopen, freopen, _Ropen</td>
</tr>
<tr>
<td>EBADNAME</td>
<td>Bad file name specified.</td>
<td>fopen, freopen, _Ropen</td>
</tr>
<tr>
<td>EBADPOS</td>
<td>The position specified is not valid.</td>
<td>fsetpos</td>
</tr>
<tr>
<td>EBADSEEK</td>
<td>Bad offset for a seek operation.</td>
<td>fgetpos, fseek</td>
</tr>
<tr>
<td>EBUSY</td>
<td>The record or file is in use.</td>
<td>perror, strerror</td>
</tr>
<tr>
<td>ECONVERT</td>
<td>Conversion error.</td>
<td>wcstomb, wcswick</td>
</tr>
<tr>
<td>EDOM</td>
<td>Domain error in math function.</td>
<td>acos, asin, atan2, cos, exp, fmod, gamma, hypot, j0, j1, jn, y0, y1, yn, log, log10, pow, sin, strtol, strtoul, sqrt, tan</td>
</tr>
<tr>
<td>EGETANDPUT</td>
<td>An illegal read operation occurred after a write operation.</td>
<td>fgetc, fread, getc, getchar</td>
</tr>
<tr>
<td>EILSEQ</td>
<td>The character sequence does not form a valid multibyte character.</td>
<td>fgetwc, fgetws, getwc, mblen,mbrlen, mbtowc, mbsrtowcs, mbsowcs, mbtowc, printf, scanf, ungetwc, wcrtomb, wcscrtombs, wcstombs, wctomb, wcswidth, wcwidth</td>
</tr>
<tr>
<td>EINVAL</td>
<td>The signal is not valid.</td>
<td>printf, scanf, signal, swprintf, swscanf, wcstol, wcstoll, wcstoul, wcstoull</td>
</tr>
</tbody>
</table>

© Copyright IBM Corp. 1998, 2013
<table>
<thead>
<tr>
<th>Error Macro</th>
<th>Description</th>
<th>Set by Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>EIO</td>
<td>Consecutive calls of I/O occurred.</td>
<td>I/O</td>
</tr>
<tr>
<td>EIOERROR</td>
<td>A non-recoverable I/O error occurred.</td>
<td>All I/O functions</td>
</tr>
<tr>
<td>EIORECERR</td>
<td>A recoverable I/O error occurred.</td>
<td>All I/O functions</td>
</tr>
<tr>
<td>ENODEV</td>
<td>Operation attempted on a wrong device.</td>
<td>fgetpos, fsee, ftell, rewind</td>
</tr>
<tr>
<td>ENOENT</td>
<td>File or library is not found.</td>
<td>perror, strerror</td>
</tr>
<tr>
<td>ENOPOS</td>
<td>No record at specified position.</td>
<td>fsetpos</td>
</tr>
<tr>
<td>ENOREC</td>
<td>Record not found.</td>
<td>fread, perror, strerror</td>
</tr>
<tr>
<td>ENOTDILT</td>
<td>File is not opened for delete operations.</td>
<td>_Rdelete</td>
</tr>
<tr>
<td>ENOTOPEN</td>
<td>File is not opened.</td>
<td>clearerr, fclose, fflush, fgetpos, fopen, freopen, ftell, setbuf, setvbuf, _Ropen, _Rclose</td>
</tr>
<tr>
<td>ENOTREAD</td>
<td>File is not opened for read operations.</td>
<td>fgetc, fread, ungetc, _Rread, _Rreadf, _Rreadind, _Rreadk, _Rreadd, _Rreadf, _Rreadindv, _Rreadk, _Rreadf, _Rreadnc, _Rreadp, _Rreads, _Rlocate</td>
</tr>
<tr>
<td>ENOTUPD</td>
<td>File is not opened for update operations.</td>
<td>_Rrlslck, _Rupdate</td>
</tr>
<tr>
<td>ENOTWRITE</td>
<td>File is not opened for write operations.</td>
<td>fputc, fwrite, _Rwrite, _Rwrited, _Rwriterd</td>
</tr>
<tr>
<td>ENUMMBRS</td>
<td>More than 1 member.</td>
<td>ftell</td>
</tr>
<tr>
<td>ENUMRECS</td>
<td>Too many records.</td>
<td>ftell</td>
</tr>
<tr>
<td>EPAD</td>
<td>Padding occurred on a write operation.</td>
<td>fwrite</td>
</tr>
<tr>
<td>EPERM</td>
<td>Insufficient authorization for access.</td>
<td>perror, strerror</td>
</tr>
<tr>
<td>EPUTANDGET</td>
<td>An illegal write operation occurred after a read operation.</td>
<td>fputc, fwrite, fputs, putc, putchar</td>
</tr>
<tr>
<td>ERANGE</td>
<td>Range error in math function.</td>
<td>cos, cosh, gamma, exp, j0, j1, jn, y0, y1, yn, log, log10, ldexp, pow, sin, sinh, strtd, strtl, strtoul, tan, wcstol, wcstoul, wcstoul, wcstoul, wcstoul, wcstold</td>
</tr>
<tr>
<td>ERECIO</td>
<td>File is opened for record I/O, so character-at-a-time processing functions cannot be used.</td>
<td>fgetc, fgetpos, fputc, fread, fseek, fsetpos, ftell</td>
</tr>
<tr>
<td>ESTDERR</td>
<td>stderr cannot be opened.</td>
<td>feof, ferror, fgetpos, fputc, fseek, fsetpos, ftell, fwrite</td>
</tr>
<tr>
<td>ESTDIN</td>
<td>stdin cannot be opened.</td>
<td>fgetc, fgetpos, fread, fseek, fsetpos, ftell</td>
</tr>
</tbody>
</table>
### Table 22. errno Macros (continued)

<table>
<thead>
<tr>
<th>Error Macro</th>
<th>Description</th>
<th>Set by Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>ESTDOUT</td>
<td>stdout cannot be opened.</td>
<td>fgetpos, fputc, fseek, fsetpos, ftell, fwrite</td>
</tr>
<tr>
<td>ETRUNC</td>
<td>Truncation occurred on I/O operation.</td>
<td>Any IO function that reads or writes a record sets errno to ETRUNC.</td>
</tr>
</tbody>
</table>

#### errno Values for Integrated File System Enabled C Stream I/O

The following table describes the possible settings when using integrated file system enabled stream I/O.

<table>
<thead>
<tr>
<th>C Stream Function</th>
<th>Possible errno Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>clearerr</td>
<td>E_BADF</td>
</tr>
<tr>
<td>fclose</td>
<td>EAGAIN, E_BADF, E_EIO, ESCANFAILURE, EINVAL</td>
</tr>
<tr>
<td>feof</td>
<td>E_BADF</td>
</tr>
<tr>
<td>ferror</td>
<td>E_BADF</td>
</tr>
<tr>
<td>fflush</td>
<td>E_ACCES, EAGAIN, E_BADF, E_BUSY, E_DAMAGE, EFAULT, EFBIG, EINVAL, EIO, ENOMEM,</td>
</tr>
<tr>
<td></td>
<td>ENOSPC, E_TRUNC, EINVAL, EINVAL, ENOSYSRSC, ENOTWRITE, EPAD</td>
</tr>
<tr>
<td>fgetc</td>
<td>E_BADF, E_ACCES, EAGAIN, E_BUSY, E_DAMAGE, EFAULT, EINVAL, EIO, ENOMEM,</td>
</tr>
<tr>
<td></td>
<td>EINVAL, ENOSYSRSC, EPUTANDGET, EDOM, ENOTREAD,</td>
</tr>
<tr>
<td>fgetpos</td>
<td>E_ACCES, EAGAIN, E_BADF, E_BUSY, E_DAMAGE, EFAULT, EINVAL, EIO, ENOSYSRSC,</td>
</tr>
<tr>
<td></td>
<td>EINVAL, EPUTANDGET, EDOM, ENOTREAD,</td>
</tr>
<tr>
<td>fgets</td>
<td>E_BADF, E_ACCES, EAGAIN, E_BUSY, E_DAMAGE, EFAULT, EINVAL, EIO, ENOMEM,</td>
</tr>
<tr>
<td></td>
<td>EINVAL, EPUTANDGET, EDOM, ENOTREAD,</td>
</tr>
<tr>
<td>fgetwc</td>
<td>E_BADF, E_ILSEQ</td>
</tr>
<tr>
<td>fgetws</td>
<td>E_BADF, E_ILSEQ</td>
</tr>
<tr>
<td>fopen</td>
<td>E_AGAIN, E_BADV, E_CONVERT, E_DAMAGE, EEXITS, EFAULT, EINVAL, EINVAL, E_EISDIR,</td>
</tr>
<tr>
<td></td>
<td>ELOOP, ENOENT, ENOMEM, ENOSPC, ENOSYS, ENOSYSRSC, ENOTDIR, ESCANFAILURE</td>
</tr>
<tr>
<td>fprintf</td>
<td>E_ACCES, EAGAIN, E_BADF, E_BUSY, E_DAMAGE, EFAULT, EFBIG, EINVAL, EINVAL, EIO,</td>
</tr>
<tr>
<td></td>
<td>ENOMEM, ENOSPC, E_TRUNC, EINVAL, ENOSYSRSC, ENOTWRITE, EPAD</td>
</tr>
<tr>
<td>fputc</td>
<td>E_ACCES, EAGAIN, E_BADF, E_BUSY, E_DAMAGE, EFAULT, EFBIG, EINVAL, EINVAL, EIO,</td>
</tr>
<tr>
<td></td>
<td>ENOMEM, ENOSPC, E_TRUNC, EINVAL, ENOSYSRSC, ENOTWRITE, EPAD</td>
</tr>
<tr>
<td>fputs</td>
<td>E_ACCES, EAGAIN, E_BADF, E_BUSY, E_DAMAGE, EFAULT, EFBIG, EINVAL, EINVAL, EIO,</td>
</tr>
<tr>
<td></td>
<td>ENOMEM, ENOSPC, E_TRUNC, EINVAL, ENOSYSRSC, ENOTWRITE, EPAD</td>
</tr>
<tr>
<td>fread</td>
<td>E_BADF, E_ACCES, EAGAIN, E_BUSY, E_DAMAGE, EFAULT, EFBIG, EINVAL, EINVAL, EIO,</td>
</tr>
<tr>
<td></td>
<td>ENOMEM, EINVAL, ENOSYSRSC, ENOTDIR</td>
</tr>
<tr>
<td>freopen</td>
<td>E_AGAIN, E_BADV, E_CONVERT, E_DAMAGE, EEXITS, EFAULT, EINVAL, E_FILE, E_FILE,</td>
</tr>
<tr>
<td></td>
<td>E.asList, ENAME_TOO_LONG, ENFILE, ENOENT, ENOTDIR</td>
</tr>
</tbody>
</table>

Runtime Considerations 545
Table 23. errno Values for Integrated File System Enabled C Stream I/O (continued)

<table>
<thead>
<tr>
<th>C Stream Function</th>
<th>Possible errno Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>fscanf</td>
<td>E_BADF, E_ACESS, E_AGAIN, E_BUSY, E_DAMAGE, EFAULT, EINVAL, EIO, ENOMEM, EUKNOWN, EGETANDPUT, EDOM, ENOTREAD</td>
</tr>
<tr>
<td>fseek</td>
<td>E_ACESS, E_AGAIN, E_BADF, E_BUSY, E_DAMAGE, EINVAL, EIO, ENOENT, ENOSPC, ENOSYSRSC, ESPPIPE, EUKNOWN, EFAULT, EPERM, EUNATCH, EUKNOWN</td>
</tr>
<tr>
<td>fsetpos</td>
<td>E_ACESS, E_AGAIN, E_BADF, E_BUSY, E_DAMAGE, EINVAL, EUKNOWN, EFAULT, EPERM, EUNATCH, EUKNOWN</td>
</tr>
<tr>
<td>ftell</td>
<td>E_ACESS, E_AGAIN, E_BADF, E_BUSY, E_DAMAGE, EINVAL, EUKNOWN, EFAULT, EPERM, EUNATCH, EUKNOWN</td>
</tr>
<tr>
<td>fwrite</td>
<td>E_ACESS, E_AGAIN, E_BADF, E_BUSY, E_DAMAGE, EINVAL, EUKNOWN, EFAULT, EPERM, EUNATCH, EUKNOWN</td>
</tr>
<tr>
<td>getc</td>
<td>E_BADF, E_ACESS, E_AGAIN, E_BUSY, E_DAMAGE, EINVAL, EUKNOWN, EFAULT, EGETANDPUT, EDOM, ENOTREAD</td>
</tr>
<tr>
<td>getchar</td>
<td>E_BADF, E_ACESS, E_AGAIN, E_BUSY, E_DAMAGE, EINVAL, EUKNOWN, EFAULT, EGETANDPUT, EDOM, ENOTREAD</td>
</tr>
<tr>
<td>gets</td>
<td>E_BADF, E_ACESS, E_AGAIN, E_BUSY, E_DAMAGE, EINVAL, EUKNOWN, EFAULT, EGETANDPUT, EDOM, ENOTREAD</td>
</tr>
<tr>
<td>getwc</td>
<td>E_BADF, E_ILSEQ</td>
</tr>
<tr>
<td>perror</td>
<td>E_BADF</td>
</tr>
<tr>
<td>printf</td>
<td>E_ACESS, E_AGAIN, E_BADF, E_BUSY, E_DAMAGE, EINVAL, EIO, ENOMEM, ENOSPC, E_TRUNC, EUKNOWN, EPUTANDGET, ENOTWRITE, EPAD</td>
</tr>
<tr>
<td>putc</td>
<td>E_ACESS, E_AGAIN, E_BADF, E_BUSY, E_DAMAGE, EINVAL, E_TRUNC, EUKNOWN, EPUTANDGET, ENOTWRITE, EPAD</td>
</tr>
<tr>
<td>putchar</td>
<td>E_ACESS, E_AGAIN, E_BADF, E_BUSY, E_DAMAGE, EINVAL, E_TRUNC, EUKNOWN, EPUTANDGET, ENOTWRITE, EPAD</td>
</tr>
<tr>
<td>puts</td>
<td>E_ACESS, E_AGAIN, E_BADF, E_BUSY, EINVAL, ENOMEM, E_TRUNC, EUKNOWN, EPUTANDGET, ENOTWRITE, EPAD</td>
</tr>
<tr>
<td>remove</td>
<td>E_ACESS, E_AGAIN, E_BADNAME, E_BADF, EINVAL, E_TRUNC, EUKNOWN, EPUTANDGET, ENOTWRITE, EPAD</td>
</tr>
<tr>
<td>rename</td>
<td>E_ACESS, E_AGAIN, E_BADNAME, E_BUSY, E_TRUNC, EUKNOWN, EPUTANDGET, ENOTWRITE, EPAD</td>
</tr>
<tr>
<td>rewind</td>
<td>E_ACESS, E_AGAIN, E_BADF, E_BUSY, EINVAL, E_TRUNC, EUKNOWN, EPUTANDGET, ENOTWRITE, EPAD</td>
</tr>
<tr>
<td>scanf</td>
<td>E_BADF, E_ACESS, E_AGAIN, E_BUSY, EINVAL, EUKNOWN, EGETANDPUT, EDOM, ENOTREAD</td>
</tr>
<tr>
<td>setbuf</td>
<td>E_BADF, EINVAL, EIO</td>
</tr>
<tr>
<td>setvbuf</td>
<td>E_BADF, EINVAL, EIO</td>
</tr>
</tbody>
</table>
### Table 23. errno Values for Integrated File System Enabled C Stream I/O (continued)

<table>
<thead>
<tr>
<th>C Stream Function</th>
<th>Possible errno Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>tmpfile</td>
<td>EACCES, EAGAIN, EBADF, EBUSY, ECONVERT, EDAMAGE, EEXITS, EFAULT, EINVAL, EIO, EISDIR, ELOOP, EMFILE, ENAMEETOOLONG, ENFILE, ENOENT, ENOMEM, ENOSPC, ENOSYS, ENOSYSRSC, ENOTDIR, EPERM, EROOBJ, EUNKNOWN, EXDEV</td>
</tr>
<tr>
<td>tmppnam</td>
<td>EACCESS, EAGAIN, EBADF, EBUSY, EDAMAGE, EFAULT, EINVAL, EIO, ENOENT, ENOSYSRSC, EUNATCH, EUNKNOWN</td>
</tr>
<tr>
<td>ungetc</td>
<td>EBADF, EIO</td>
</tr>
<tr>
<td>ungetwc</td>
<td>EBADF, EILSEQ</td>
</tr>
<tr>
<td>vfprintf</td>
<td>EACCES, EAGAIN, EBADF, EBUSY, EDAMAGE, EFAULT, EFBIG, EINVAL, EIO, ENOMEM, ENOSPC, ETRUNC, EUNKNOWN, EPUTANDGET, ENOTWRITE, EPAD</td>
</tr>
<tr>
<td>vprintf</td>
<td>EACCES, EAGAIN, EBADF, EBUSY, EDAMAGE, EFAULT, EFBIG, EINVAL, EIO, ENOMEM, ENOSPC, ETRUNC, EUNKNOWN, EPUTANDGET, ENOTWRITE, EPAD</td>
</tr>
</tbody>
</table>

### Record Input and Output Error Macro to Exception Mapping

The following table describes what occurs if the signal SIGIO is raised. Only *ESCAPE, *NOTIFY, and *STATUS messages are monitored.

#### Table 24. Record Input and Output Error Macro to Exception Mapping

<table>
<thead>
<tr>
<th>Description</th>
<th>Messages (_EXCP_MSGID)</th>
<th>errno setting</th>
</tr>
</thead>
<tbody>
<tr>
<td>*STATUS and *NOTIFY</td>
<td>CPF4001 to CPF40FF, CPF4401 to CPF44FF, CPF4901 to CPF49FF, CPF5004</td>
<td>errno is not set, a default reply is returned to the operating system.</td>
</tr>
<tr>
<td>Recoverable I/O error</td>
<td>CPF4701 to CPF47FF, CPF4801 to CPF48FF, CPF5001 to CPF5003, CPF5005 to CPF50FF, CPF50FF</td>
<td>EIORECERR</td>
</tr>
<tr>
<td>Non-recoverable I/O error²</td>
<td>CPF4101 to CPF41FF, CPF4201 to CPF42FF, CPF4301 to CPF43FF, CPF4501 to CPF45FF, CPF4601 to CPF46FF, CPF5101 to CPF51FF, CPF5201 to CPF52FF, CPF5301 to CPF53FF, CPF5401 to CPF54FF, CPF5501 to CPF55FF, CPF5601 to CPF56FF</td>
<td>EIOERROR</td>
</tr>
<tr>
<td>Truncation occurred at I/O operation</td>
<td>C2M3003</td>
<td>ETRUNC</td>
</tr>
<tr>
<td>File is not opened</td>
<td>C2M3004</td>
<td>ENOTOPEN</td>
</tr>
<tr>
<td>File is not opened for read operations</td>
<td>C2M3005</td>
<td>ENOTREAD</td>
</tr>
<tr>
<td>File is not opened for write operations</td>
<td>C2M3009</td>
<td>ENOTWRITE</td>
</tr>
<tr>
<td>Bad file name specified</td>
<td>C2M3014</td>
<td>EBADNAME</td>
</tr>
</tbody>
</table>
Table 24. Record Input and Output Error Macro to Exception Mapping (continued)

<table>
<thead>
<tr>
<th>Description</th>
<th>Messages (_EXCP_MSGID)</th>
<th>errno setting</th>
</tr>
</thead>
<tbody>
<tr>
<td>The file mode specified is not valid</td>
<td>C2M3015</td>
<td>EBADMODE</td>
</tr>
<tr>
<td>File is not opened for update operations</td>
<td>C2M3041</td>
<td>ENOTUPD</td>
</tr>
<tr>
<td>File is not opened for delete operations</td>
<td>C2M3042</td>
<td>ENOTDLT</td>
</tr>
<tr>
<td>The key length specified is not valid</td>
<td>C2M3044</td>
<td>EBADKEYLN</td>
</tr>
<tr>
<td>A non-recoverable I/O error occurred</td>
<td>C2M3101</td>
<td>EIOERROR</td>
</tr>
<tr>
<td>A recoverable I/O error occurred</td>
<td>C2M3102</td>
<td>EIORECERR</td>
</tr>
</tbody>
</table>

Note:
- 1 The error is percolated to the user, therefore the user’s direct monitor handlers, ILE C condition handlers and signal handler may get control. The initial setting for SIGIO is SIG_IGN.
- 2 The type of device determines whether the error is recoverable or not recoverable. The following IBM publications contain information about recoverable and non-recoverable system exceptions for each specific file type:
  - ICF Programming
  - ADTS/400: Advanced Printer Function
  - Application Display Programming
  - Database Programming

Signal Handling Action Definitions

The following table shows the initial state of the C signal values and their handling action definitions when SYSIFCOPT(*NOASYNC_SIGNAL) is specified on the compilation command. SIG_DFL always percolates the condition to the handler. Resume indicates the exception is handled, and the application continues.

Table 25. Handling Action Definitions for Signal Values

<table>
<thead>
<tr>
<th>Signal Value</th>
<th>Initial State</th>
<th>SIG_DFL</th>
<th>SIG_IGN</th>
<th>Return from Handler</th>
</tr>
</thead>
<tbody>
<tr>
<td>SIGABRT¹</td>
<td>SIG_DFL</td>
<td>Percolate</td>
<td>Ignore</td>
<td>Resume</td>
</tr>
<tr>
<td>SIGALL²</td>
<td>SIG_DFL</td>
<td>Percolate</td>
<td>Ignore</td>
<td>Resume</td>
</tr>
<tr>
<td>SIGFPE</td>
<td>SIG_DFL</td>
<td>Percolate</td>
<td>Ignore³</td>
<td>Resume⁴</td>
</tr>
<tr>
<td>SIGILL</td>
<td>SIG_DFL</td>
<td>Percolate</td>
<td>Ignore³</td>
<td>Resume⁴</td>
</tr>
<tr>
<td>SIGINT</td>
<td>SIG_DFL</td>
<td>Percolate</td>
<td>Ignore</td>
<td>Resume</td>
</tr>
<tr>
<td>SIGIO</td>
<td>SIG_IGN</td>
<td>Percolate</td>
<td>Ignore</td>
<td>Resume</td>
</tr>
<tr>
<td>SIGOTHER</td>
<td>SIG_DFL</td>
<td>Percolate</td>
<td>Ignore³</td>
<td>Resume⁴</td>
</tr>
<tr>
<td>SIGSEGV</td>
<td>SIG_DFL</td>
<td>Percolate</td>
<td>Ignore³</td>
<td>Resume⁴</td>
</tr>
<tr>
<td>SIGTERM</td>
<td>SIG_DFL</td>
<td>Percolate</td>
<td>Ignore</td>
<td>Resume</td>
</tr>
</tbody>
</table>
### Table 25. Handling Action Definitions for Signal Values (continued)

<table>
<thead>
<tr>
<th>Signal Value</th>
<th>Initial State</th>
<th>SIG_DFL</th>
<th>SIG_IGN</th>
<th>Return from Handler</th>
</tr>
</thead>
<tbody>
<tr>
<td>SIGUSR1</td>
<td>SIG_DFL</td>
<td>Percolate</td>
<td>Ignore</td>
<td>Resume</td>
</tr>
<tr>
<td>SIGUSR2</td>
<td>SIG_DFL</td>
<td>Percolate</td>
<td>Ignore</td>
<td>Resume</td>
</tr>
</tbody>
</table>

**Note:**
- • 1 Can only be signaled by the `raise()` function or the `abort()` function
- • 2 `SIGALL` cannot be signaled by the `raise()` function.
- • 3 If the value of the signal is SIGFPE, SIGILL or SIGSEGV the behavior is undefined.
- • 4 If the signal is hardware-generated, then the behavior undefined.

The following table shows the initial state of the C signal values and their handling action definitions when `SYSIFCOPT(*ASYNC SIGNAL)` is specified on the compilation command.

### Table 26. Default Actions for Signal Values

<table>
<thead>
<tr>
<th>Value</th>
<th>Default Action</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>SIGABRT</td>
<td>2</td>
<td>Abnormal termination.</td>
</tr>
<tr>
<td>SIGFPE</td>
<td>2</td>
<td>Arithmetic exceptions that are not masked, such as overflow, division by zero, and incorrect operation.</td>
</tr>
<tr>
<td>SIGILL</td>
<td>2</td>
<td>Detection of an incorrect function image.</td>
</tr>
<tr>
<td>SIGINT</td>
<td>2</td>
<td>Interactive attention.</td>
</tr>
<tr>
<td>SIGSEGV</td>
<td>2</td>
<td>Incorrect access to storage.</td>
</tr>
<tr>
<td>SIGTERM</td>
<td>2</td>
<td>Termination request sent to the program.</td>
</tr>
<tr>
<td>SIGUSR1</td>
<td>2</td>
<td>Intended for use by user applications.</td>
</tr>
<tr>
<td>SIGUSR2</td>
<td>2</td>
<td>Intended for use by user applications.</td>
</tr>
<tr>
<td>SIGALRM</td>
<td>2</td>
<td>A timeout signal that is sent by <code>alarm()</code>.</td>
</tr>
<tr>
<td>SIGHUP</td>
<td>2</td>
<td>A controlling terminal is hung up, or the controlling process ended.</td>
</tr>
<tr>
<td>SIGKILL</td>
<td>1</td>
<td>A termination signal that cannot be caught or ignored.</td>
</tr>
<tr>
<td>SIGPIPE</td>
<td>3</td>
<td>A write to a pipe that is not being read.</td>
</tr>
<tr>
<td>SIGQUIT</td>
<td>2</td>
<td>A quit signal for a terminal.</td>
</tr>
<tr>
<td>SIGCHLD</td>
<td>3</td>
<td>An ended or stopped child process. SIGCLD is an alias name for this signal.</td>
</tr>
<tr>
<td>SIGCONT</td>
<td>5</td>
<td>If stopped, continue.</td>
</tr>
<tr>
<td>SIGSTOP</td>
<td>4</td>
<td>A stop signal that cannot be caught or ignored.</td>
</tr>
<tr>
<td>SIGTSTP</td>
<td>4</td>
<td>A stop signal for a terminal.</td>
</tr>
<tr>
<td>SIGTTIN</td>
<td>4</td>
<td>A background process attempted to read from a controlling terminal.</td>
</tr>
<tr>
<td>SIGTTOU</td>
<td>4</td>
<td>A background process attempted to write to a controlling terminal.</td>
</tr>
</tbody>
</table>
Table 26. Default Actions for Signal Values (continued)

<table>
<thead>
<tr>
<th>Value</th>
<th>Default Action</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>SIGIO</td>
<td>3</td>
<td>Completion of input or output.</td>
</tr>
<tr>
<td>SIGURG</td>
<td>3</td>
<td>High bandwidth data is available at a socket.</td>
</tr>
<tr>
<td>SIGPOLL</td>
<td>2</td>
<td>Pollable event.</td>
</tr>
<tr>
<td>SIGBUS</td>
<td>2</td>
<td>Specification exception.</td>
</tr>
<tr>
<td>SIGPRE</td>
<td>2</td>
<td>Programming exception.</td>
</tr>
<tr>
<td>SIGSYS</td>
<td>2</td>
<td>Bad system call.</td>
</tr>
<tr>
<td>SIGTRAP</td>
<td>2</td>
<td>Trace or breakpoint trap.</td>
</tr>
<tr>
<td>SIGPROF</td>
<td>2</td>
<td>Profiling timer expired.</td>
</tr>
<tr>
<td>SIGVTALRM</td>
<td>2</td>
<td>Virtual timer expired.</td>
</tr>
<tr>
<td>SIGXCPU</td>
<td>2</td>
<td>Processor time limit exceeded.</td>
</tr>
<tr>
<td>SIGXFSZ</td>
<td>2</td>
<td>File size limit exceeded.</td>
</tr>
<tr>
<td>SIGDANGER</td>
<td>2</td>
<td>System crash is imminent.</td>
</tr>
<tr>
<td>SIGPCANCEL</td>
<td>2</td>
<td>Thread termination signal that cannot be caught or ignored.</td>
</tr>
</tbody>
</table>

Default Actions:

1. End the process immediately.
2. End the request.
3. Ignore the signal.
4. Stop the process.
5. Continue the process if it is currently stopped. Otherwise, ignore the signal.

Signal to Exception Mapping

The following table shows the system exception messages that are mapped to a signal. All *ESCAPE exception messages are mapped to signals. The *STATUS and *NOTIFY messages that map to SIGIO as defined in Table 24 on page 547 are mapped to signals.

Table 27. Signal to Exception Mapping

<table>
<thead>
<tr>
<th>Signal</th>
<th>Message</th>
</tr>
</thead>
<tbody>
<tr>
<td>SIGABRT</td>
<td>C2M1601</td>
</tr>
<tr>
<td>SIGALL</td>
<td>C2M1610 (if explicitly raised)</td>
</tr>
<tr>
<td>SIGFPE</td>
<td>C2M1602, MCH1201 to MCH1204, MCH1206 to MCH1215, MCH1221 to MCH1224, MCH1838 to MCH1839</td>
</tr>
</tbody>
</table>
Table 27. Signal to Exception Mapping (continued)

<table>
<thead>
<tr>
<th>Signal</th>
<th>Message</th>
</tr>
</thead>
<tbody>
<tr>
<td>SIGILL</td>
<td>C2M1603, MCH0401, MCH1002, MCH1004, MCH1205, MCH1216 to MCH1219, MCH1801 to MCH1802, MCH1807 to MCH1808, MCH1819 to MCH1820, MCH1824 to MCH1825, MCH1832, MCH1837, MCH1852, MCH1854 to MCH1857, MCH1867, MCH2003 to MCH2004, MCH2202, MCH2602, MCH2604, MCH2808, MCH2810 to MCH2811, MCH3201 to MCH3203, MCH4201 to MCH4211, MCH4213, MCH4296 to MCH4298, MCH4401 to MCH4403, MCH4406 to MCH4408, MCH4421, MCH4427 to MCH4428, MCH4801, MCH4804 to MCH4805, MCH5001 to MCH5003, MCH5401 to MCH5402, MCH5601, MCH6001 to MCH6002, MCH6201, MCH6208, MCH6216, MCH6220, MCH6403, MCH6601 to MCH6602, MCH6609 to MCH6612</td>
</tr>
<tr>
<td>SIGINT</td>
<td>C2M1604</td>
</tr>
<tr>
<td>SIGIO</td>
<td>C2M1609, See Table 24 on page 547 for the exception mappings.</td>
</tr>
<tr>
<td>SIGOTHER</td>
<td>C2M1611 (if explicitly raised)</td>
</tr>
<tr>
<td>SIGSEGV</td>
<td>C2M1605, MCH0201, MCH0601 to MCH0606, MCH0801 to MCH0803, MCH1001, MCH1003, MCH1005 to MCH1006, MCH1220, MCH1401 to MCH1402, MCH1602, MCH1603 to MCH1605, MCH1668, MCH1803 to MCH1806, MCH1809 to MCH1811, MCH1873 to MCH1815, MCH1821 to MCH1823, MCH1826 to MCH1829, MCH1833, MCH1836, MCH1848, MCH1850, MCH1851, MCH1854 to MCH1857, MCH1864 to MCH1866, MCH1898, MCH2001 to MCH2002, MCH2005 to MCH2006, MCH2201, MCH2203 to MCH2205, MCH2401, MCH2601, MCH2603, MCH2604, MCH2605, MCH2801 to MCH2804, MCH2806 to MCH2809, MCH3001, MCH3401 to MCH3408, MCH3410, MCH3601 to MCH3602, MCH3603 to MCH3604, MCH3802, MCH4001 to MCH4002, MCH4010, MCH4212, MCH4404 to MCH4405, MCH4416 to MCH4420, MCH4422 to MCH4426, MCH4429 to MCH4437, MCH4601, MCH4802 to MCH4803, MCH4806 to MCH4812, MCH5201 to MCH5204, MCH5602 to MCH5603, MCH5801 to MCH5804, MCH6203 to MCH6204, MCH6206, MCH6217 to MCH6219, MCH6221 to MCH6222, MCH6401 to MCH6402, MCH6404, MCH6603 to MCH6608, MCH6801</td>
</tr>
<tr>
<td>SIGTERM</td>
<td>C2M1606</td>
</tr>
<tr>
<td>SIGUSR1</td>
<td>C2M1607</td>
</tr>
<tr>
<td>SIGUSR2</td>
<td>C2M1608</td>
</tr>
</tbody>
</table>

Cancel Handler Reason Codes

The following table lists the bits that are set in the reason code. If the activation group is to be stopped, then the **activation group is stopped** bit is also set in the reason code. These bits must be correlated to `_CNL_MASK_T` in `_CNL_Hndlr_Parms_T` in `<except.h>`. Column 2 contains the macro constant defined for the cancel reason mask in `<except.h>`.

Table 28. Determining Canceled Invocation Reason Codes

<table>
<thead>
<tr>
<th>Function</th>
<th>Bits set in reason code</th>
<th>Rationale</th>
</tr>
</thead>
<tbody>
<tr>
<td>Library functions</td>
<td></td>
<td></td>
</tr>
<tr>
<td>exit</td>
<td>_EXIT_VERB</td>
<td>The definition of exit is normal end of processing, and therefore invocations canceled by this function is done with a reason code of normal.</td>
</tr>
</tbody>
</table>
### Table 28. Determining Canceled Invocation Reason Codes (continued)

<table>
<thead>
<tr>
<th>Function</th>
<th>Bits set in reason code</th>
<th>Rationale</th>
</tr>
</thead>
<tbody>
<tr>
<td>abort</td>
<td>_ABNORMAL_TERM _EXIT_VERB</td>
<td>The definition of abort is abnormal end of processing, and therefore invocations canceled by this function are done with a reason code of abnormal.</td>
</tr>
<tr>
<td>longjmp</td>
<td>_JUMP</td>
<td>The general use of the longjmp() function is to return from an exception handler, although it may be used in non-exception situations as well. It is used as part of the &quot;normal&quot; path for a program, and therefore any invocations canceled because of it are cancelled with a reason code of normal.</td>
</tr>
<tr>
<td>Unhandled function check</td>
<td>_ABNORMAL_TERM UNHANDLED_EXCP</td>
<td>Not handling an exception which is an abnormal situation.</td>
</tr>
</tbody>
</table>

#### System APIs

| CEEMRCR                         | _ABNORMAL_TERM _EXCP_SENT           | This API is only used during exception processing. It is typically used to cancel invocations where a resume is not possible, or at least the behavior would be undefined if control was resumed in them. Also, these invocations have had a chance to handle the exception but did not do so. Invocations canceled by this API are done with reason code of abnormal. |
| QMHSNDPM / QMHRSNEM (escape messages) Message Handler APIs | _ABNORMAL_TERM _EXCP_SENT | All invocations down to the target invocation are canceled without any chance of handling the exception. The API topic contains information about these APIs. |

#### System commands

| Process end                     | _ABNORMAL_TERM _PROCESS_TERM _AG_TERMINATING | Any externally initiated shutdown of an activation group is considered abnormal.                                                           |
| RCLACTGRP                       | _ABNORMAL_TERM _RCLRSC                | The default is abnormal termination. The termination could be normal if a normal/abnormal flag is added to the command.                     |

### Table 29. Common Reason Code for Cancelling Invocations

<table>
<thead>
<tr>
<th>Bit</th>
<th>Description</th>
<th>Header File Constant &lt;except.h&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bits 0</td>
<td>Reserved</td>
<td></td>
</tr>
<tr>
<td>Bits 1</td>
<td>Invocation canceled due to sending exception message</td>
<td>_EXCP_SENT</td>
</tr>
<tr>
<td>Bits 2-15</td>
<td>Reserved</td>
<td></td>
</tr>
<tr>
<td>Bit 16</td>
<td>0 - normal end of process 1 - abnormal end of process</td>
<td>_ABNORMAL_TERM</td>
</tr>
</tbody>
</table>
Table 29. Common Reason Code for Cancelling Invocations (continued)

<table>
<thead>
<tr>
<th>Bit</th>
<th>Description</th>
<th>Header File Constant &lt;except.h&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bit 17</td>
<td>Activation Group is ending.</td>
<td>_AG_TERMINATING</td>
</tr>
<tr>
<td>Bit 18</td>
<td>Initiated by Reclaim Activation Group (RCLACTGRP)</td>
<td>_RCLRSC</td>
</tr>
<tr>
<td>Bit 19</td>
<td>Initiated by the process end.</td>
<td>_PROCESS_TERM</td>
</tr>
<tr>
<td>Bit 20</td>
<td>Initiated by an exit() function.</td>
<td>_EXIT_VERB</td>
</tr>
<tr>
<td>Bit 21</td>
<td>Initiated by an unhandled function check.</td>
<td>_UNHANDLED_EXCP</td>
</tr>
<tr>
<td>Bit 22</td>
<td>Invocation canceled due to a longjmp() function.</td>
<td>_JUMP</td>
</tr>
<tr>
<td>Bit 23</td>
<td>Invocation canceled due to a jump because of exception processing.</td>
<td>_JUMP_EXCP</td>
</tr>
</tbody>
</table>

Bits 24-31 Reserved (0)

## Exception Classes

In a CL program, you can monitor for a selected group of exceptions, or a single exception, based on the exception identifier. The only class2 values the exception handler will monitor for are _C2_MH_ESCAPE, _C2_MH_STATUS, _C2_MH_NOTIFY, and _C2_MH_FUNCTION_CHECK. For more information about using the #pragma exception handler directive, see the ILE C/C++ Compiler Reference. This table defines all the exception classes you can specify.

### Table 30. Exception Classes

<table>
<thead>
<tr>
<th>Bit position</th>
<th>Header File Constant in &lt;except.h&gt;</th>
<th>Exception class</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>_C1_BINARY_OVERFLOW</td>
<td>Binary overflow or divide by zero</td>
</tr>
<tr>
<td>1</td>
<td>_C1_DECIMAL_OVERFLOW</td>
<td>Decimal overflow or divide by zero</td>
</tr>
<tr>
<td>2</td>
<td>_C1_DECIMAL_DATA_ERROR</td>
<td>Decimal data error</td>
</tr>
<tr>
<td>3</td>
<td>_C1_FLOAT_OVERFLOW</td>
<td>Floating-point overflow or divide by zero</td>
</tr>
<tr>
<td>4</td>
<td>_C1_FLOAT_UNDERFLOW</td>
<td>Floating-point underflow or inexact result</td>
</tr>
<tr>
<td>5</td>
<td>_C1_INVALID_FLOAT_OPERAND</td>
<td>Floating-point invalid operand or conversion error</td>
</tr>
<tr>
<td>6</td>
<td>_C1_OTHER_DATA_ERROR</td>
<td>Other data error, for example edit mask</td>
</tr>
<tr>
<td>7</td>
<td>_C1_SPECIFICATION_ERROR</td>
<td>Specification (operand alignment) error</td>
</tr>
<tr>
<td>8</td>
<td>_C1_POINTER_NOT_VALID</td>
<td>Pointer not set指pointer type invalid</td>
</tr>
<tr>
<td>9</td>
<td>_C1_OBJECT_NOT_FOUND</td>
<td>Object not found</td>
</tr>
<tr>
<td>10</td>
<td>_C1_OBJECT_DESTROYED</td>
<td>Object destroyed</td>
</tr>
<tr>
<td>Bits 24-31</td>
<td>Reserved (0)</td>
<td></td>
</tr>
</tbody>
</table>
### Data Type Compatibility

Each high-level language has different data types. When you want to pass data between programs that are written in different languages, you must be aware of these differences.

Some data types in the ILE C programming language have no direct equivalent in other languages. However, you can simulate data types in other languages that use ILE C data types.

The following table shows the ILE C data type compatibility with ILE RPG.
<table>
<thead>
<tr>
<th>ILE C declaration in prototype</th>
<th>Free-form ILE RPG syntax</th>
<th>ILE RPG D spec, columns 33 to 39</th>
<th>Length</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>char[n]</td>
<td></td>
<td></td>
<td>n</td>
<td>An array of characters where n=1 to 16777210. If it is a null-terminated string parameter, code the prototyped parameter as a pointer with the VALUE and OPTIONS(*STRING) keywords.</td>
</tr>
<tr>
<td>char *</td>
<td></td>
<td></td>
<td>nA</td>
<td></td>
</tr>
<tr>
<td>char</td>
<td>IND</td>
<td>1N</td>
<td>1</td>
<td>An Indicator.</td>
</tr>
<tr>
<td>char[n]</td>
<td>ZONED(n)</td>
<td>nS 0</td>
<td>n</td>
<td>A zoned decimal.</td>
</tr>
<tr>
<td>_Packed struct {unsigned short i; char[n]}</td>
<td>VARCHAR(n)</td>
<td>nA VARYING</td>
<td>n+2</td>
<td>A variable length field where i is the intended length and n is the maximum length. Note: If n is greater than 65535, the first element of the struct is an unsigned int, and the length is n+4.</td>
</tr>
<tr>
<td>_Packed struct {unsigned short i; char[n]}</td>
<td>VARCHAR(n : 4)</td>
<td>nA VARYING(4)</td>
<td>n+4</td>
<td>A variable length field where i is the intended length and n is the maximum length.</td>
</tr>
<tr>
<td>wchar_t[n]</td>
<td>UCS2(n)</td>
<td>nC</td>
<td>2n</td>
<td>An array of UCS-2 characters. Note: The RPG UCS-2 type also supports UTF-16 data by specifying CCSID(1200).</td>
</tr>
<tr>
<td>_Packed struct {unsigned short i; wchar_t[n]}</td>
<td>VARUCS2(n)</td>
<td>nC VARYING</td>
<td>2n+2</td>
<td>A variable length UCS-2 field where i is the intended length and n is the maximum length. Note: If n is greater than 65535, the first element of the struct is an unsigned int, and the length is 2n+4. Note: The RPG UCS-2 type also supports UTF-16 data by specifying CCSID(1200).</td>
</tr>
<tr>
<td>_Packed struct {unsigned int i; wchar_t[n]}</td>
<td>VARUCS2(n : 4)</td>
<td>nC VARYING(4)</td>
<td>2n+4</td>
<td>A variable length UCS-2 field where i is the intended length and n is the maximum length. Note: The RPG UCS-2 type also supports UTF-16 data by specifying CCSID(1200).</td>
</tr>
<tr>
<td>wchar_t[n]</td>
<td>GRAPH(n)</td>
<td>nG</td>
<td>2n</td>
<td>An array of graphic characters.</td>
</tr>
<tr>
<td>ILE C declaration in prototype</td>
<td>Free-form ILE RPG syntax</td>
<td>ILE RPG D spec, columns 33 to 39</td>
<td>Length</td>
<td>Comments</td>
</tr>
<tr>
<td>-------------------------------</td>
<td>--------------------------</td>
<td>-----------------------------------</td>
<td>--------</td>
<td>----------</td>
</tr>
<tr>
<td><code>_Packed struct {unsigned short i; wchar_t[i]}</code></td>
<td>VARGRAPH(n)</td>
<td>nG VARYING</td>
<td>2n +2</td>
<td>A variable length graphic field where i is the intended length and n is the maximum length. Note: If n is greater than 65535, the first element of the struct is an unsigned int, and the length is 2n+4.</td>
</tr>
<tr>
<td><code>_Packed struct {unsigned int i; wchar_t[i]}</code></td>
<td>VARGRAPH(n : 4)</td>
<td>nG VARYING(4)</td>
<td>2n +4</td>
<td>A variable length graphic field where i is the intended length and n is the maximum length.</td>
</tr>
<tr>
<td>char[n]</td>
<td>DATE</td>
<td>D</td>
<td>8, 10</td>
<td>A date field.</td>
</tr>
<tr>
<td>char[n]</td>
<td>TIME</td>
<td>T</td>
<td>8</td>
<td>A time field.</td>
</tr>
<tr>
<td>char[n]</td>
<td>TIMESTAMP</td>
<td>Z</td>
<td>20-32</td>
<td>A time stamp field.</td>
</tr>
<tr>
<td>short int</td>
<td>INT(5)</td>
<td>5I 0</td>
<td>2</td>
<td>An integer field.</td>
</tr>
<tr>
<td>short unsigned int</td>
<td>UNS(5)</td>
<td>5U 0</td>
<td>2</td>
<td>An unsigned integer field.</td>
</tr>
<tr>
<td>int</td>
<td>INT(10)</td>
<td>10I 0</td>
<td>4</td>
<td>An integer field.</td>
</tr>
<tr>
<td>unsigned int</td>
<td>UNS(10)</td>
<td>10U 0</td>
<td>4</td>
<td>An unsigned integer field</td>
</tr>
<tr>
<td>long int</td>
<td>INT(10)</td>
<td>10I 0</td>
<td>4</td>
<td>An integer field.</td>
</tr>
<tr>
<td>long unsigned int</td>
<td>UNS(10)</td>
<td>10U 0</td>
<td>4</td>
<td>An unsigned integer field.</td>
</tr>
<tr>
<td>long long int</td>
<td>INT(20)</td>
<td>20I 0</td>
<td>8</td>
<td>An 8-byte integer field.</td>
</tr>
<tr>
<td>long long unsigned int</td>
<td>UNS(20)</td>
<td>20U 0</td>
<td>8</td>
<td>An 8-byte unsigned integer field.</td>
</tr>
<tr>
<td>struct {unsigned int : n}x;</td>
<td>Not supported.</td>
<td>Not supported.</td>
<td>4</td>
<td>A 4-byte unsigned integer, a bitfield.</td>
</tr>
<tr>
<td>float</td>
<td>FLOAT(4)</td>
<td>4F</td>
<td>4</td>
<td>A 4-byte floating point.</td>
</tr>
<tr>
<td>double</td>
<td>FLOAT(8)</td>
<td>8F</td>
<td>8</td>
<td>An 8-byte double.</td>
</tr>
<tr>
<td>long double</td>
<td>FLOAT(8)</td>
<td>8F</td>
<td>8</td>
<td>An 8-byte long double.</td>
</tr>
<tr>
<td><code>_Decimal32</code></td>
<td>Not supported</td>
<td>Not supported</td>
<td>4</td>
<td>A 4-byte decimal floating point.</td>
</tr>
<tr>
<td><code>_Decimal64</code></td>
<td>Not supported</td>
<td>Not supported</td>
<td>8</td>
<td>An 8-byte decimal floating point.</td>
</tr>
</tbody>
</table>
### Table 31. ILE C Data Type Compatibility with ILE RPG (continued)

<table>
<thead>
<tr>
<th>ILE C declaration in prototype</th>
<th>Free-form ILE RPG syntax</th>
<th>ILE RPG D spec, columns 33 to 39</th>
<th>Length</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>_Decimal128</td>
<td>Not supported</td>
<td>Not supported</td>
<td>16</td>
<td>A 16-byte decimal floating point.</td>
</tr>
</tbody>
</table>
| enum                          | INT(p)                   | pI                               | 1, 2, 4 | Enumeration. For n = 1,2,4: p = 3,5,10  
Define ILE RPG named constants with the enum values. |
| *                             | POINTER                  | *                                | 16     | A pointer. |
| decimal(n,p)                  | PACKED(n:p)              | n/2 + 1                          | A packed decimal. n must be less than or equal to 30. |
| union.element                 | Keyword POS(1)           | <type> with keyword OVERLAY(data structure name) | element length | An element of a union. |
| data_type[n]                  | <type> with keyword DIM(n) | <type> with keyword DIM(n)       | 16     | An array to which C passes a pointer. |
| struct                        | data structure           | data structure                   | n      | A structure. Use the _Packed qualifier on the struct if the RPG data structure does not have the ALIGN keyword.  
Use the ALIGN keyword on the RPG data structure if the C struct does not have the _Packed qualifier. |
| pointer to function           | POINTER(*PROC)           | *                                | 16     | A 16-byte pointer. |

The following table shows the ILE C data type compatibility with ILE COBOL.

### Table 32. ILE C Data Type Compatibility with ILE COBOL

<table>
<thead>
<tr>
<th>ILE C declaration in prototype</th>
<th>ILE COBOL LINKAGE SECTION</th>
<th>Length</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>char[n]</td>
<td>PIC X(n).</td>
<td>n</td>
<td>An array of characters where n=1 to 3,000,000</td>
</tr>
<tr>
<td>char</td>
<td>PIC 1 INDIC ..</td>
<td>1</td>
<td>An indicator.</td>
</tr>
<tr>
<td>char[n]</td>
<td>PIC S9(n) DISPLAY</td>
<td>n</td>
<td>A zoned decimal.</td>
</tr>
<tr>
<td>wchar_t[n]</td>
<td>PIC G(n)</td>
<td>2n</td>
<td>A graphic data type.</td>
</tr>
<tr>
<td>ILE C declaration in prototype</td>
<td>ILE COBOL LINKAGE SECTION</td>
<td>Length</td>
<td>Comments</td>
</tr>
<tr>
<td>-------------------------------</td>
<td>--------------------------</td>
<td>--------</td>
<td>----------</td>
</tr>
<tr>
<td>_Packed struct {short i; char[n]}</td>
<td>05 VL-FIELD. 10 i PIC S9(4) COMP-4. 10 char X(n).</td>
<td>n+2</td>
<td>A variable length field where i is the intended length and n is the maximum length.</td>
</tr>
<tr>
<td>char[n]</td>
<td>PIC X(n).</td>
<td>6</td>
<td>A date field.</td>
</tr>
<tr>
<td>char[n]</td>
<td>PIC X(n).</td>
<td>5</td>
<td>A day field.</td>
</tr>
<tr>
<td>char</td>
<td>PIC X.</td>
<td>1</td>
<td>A day-of-week field.</td>
</tr>
<tr>
<td>char[n]</td>
<td>PIC X(n).</td>
<td>8</td>
<td>A time field.</td>
</tr>
<tr>
<td>char[n]</td>
<td>PIC X(n).</td>
<td>26</td>
<td>A time stamp field.</td>
</tr>
<tr>
<td>short int</td>
<td>PIC S9(4) COMP-4.</td>
<td>2</td>
<td>A 2-byte signed integer with a range of -9999 to +9999.</td>
</tr>
<tr>
<td>short int</td>
<td>PIC S9(4) BINARY.</td>
<td>2</td>
<td>A 2-byte signed integer with a range of -9999 to +9999.</td>
</tr>
<tr>
<td>int</td>
<td>PIC S9(9) COMP-4.</td>
<td>4</td>
<td>A 4-byte signed integer with a range of -999999999 to +999999999.</td>
</tr>
<tr>
<td>int</td>
<td>PIC S9(9) BINARY.</td>
<td>4</td>
<td>A 4-byte signed integer with a range of -999999999 to +999999999.</td>
</tr>
<tr>
<td>int</td>
<td>USAGE IS INDEX</td>
<td>4</td>
<td>A 4-byte integer.</td>
</tr>
<tr>
<td>long int</td>
<td>PIC S9(9) COMP-4.</td>
<td>4</td>
<td>A 4-byte signed integer with a range of -999999999 to +999999999.</td>
</tr>
<tr>
<td>long int</td>
<td>PIC S9(9) BINARY.</td>
<td>4</td>
<td>A 4-byte signed integer with a range of -999999999 to +999999999.</td>
</tr>
<tr>
<td>struct {unsigned int : n}x;</td>
<td>PIC 9(9) COMP-4. PIC X(4).</td>
<td>4</td>
<td>Bitfields can be manipulated using hex literals.</td>
</tr>
<tr>
<td>float</td>
<td>USAGE IS COMP-1</td>
<td>4</td>
<td>A 4-byte floating point.</td>
</tr>
<tr>
<td>double</td>
<td>USAGE IS COMP-2</td>
<td>8</td>
<td>An 8-byte double.</td>
</tr>
<tr>
<td>long double</td>
<td>Not supported.</td>
<td>8</td>
<td>An 8-byte long double.</td>
</tr>
<tr>
<td>enum</td>
<td>Not supported.</td>
<td>1, 2, 4</td>
<td>Enumeration.</td>
</tr>
<tr>
<td>*</td>
<td>USAGE IS POINTER</td>
<td>16</td>
<td>A pointer.</td>
</tr>
<tr>
<td>decimal(n,p)</td>
<td>PIC S9(n-p)V9(p) COMP-3</td>
<td>n/2+1</td>
<td>A packed decimal.</td>
</tr>
<tr>
<td>decimal(n,p)</td>
<td>PIC S9(n-p) 9(p) PACKED-DECIMAL</td>
<td>n/2+1</td>
<td>A packed decimal.</td>
</tr>
<tr>
<td>union.element</td>
<td>REDEFINES</td>
<td>element length</td>
<td>An element of a union.</td>
</tr>
</tbody>
</table>
### Table 32. ILE C Data Type Compatibility with ILE COBOL (continued)

<table>
<thead>
<tr>
<th>ILE C declaration in prototype</th>
<th>ILE COBOL LINKAGE SECTION</th>
<th>Length</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>data_type[n]</td>
<td>OCCURS</td>
<td>16</td>
<td>An array to which C passes a pointer.</td>
</tr>
<tr>
<td>struct</td>
<td>01 record 05 field1 05 field2</td>
<td>n</td>
<td>A structure. Use the _Packed qualifier on the struct. Structures passed should be passed as a pointer to the structure if you want to change the contents of the structure.</td>
</tr>
<tr>
<td>pointer to function</td>
<td>PROCEDURE-POINTER</td>
<td>16</td>
<td>A 16 byte pointer to a procedure.</td>
</tr>
<tr>
<td>Not supported.</td>
<td>PIC S9(18) COMP-4.</td>
<td>8</td>
<td>An 8 byte integer.</td>
</tr>
<tr>
<td>Not supported.</td>
<td>PIC S9(18) BINARY.</td>
<td>8</td>
<td>An 8 byte integer.</td>
</tr>
</tbody>
</table>

The following table shows the ILE C data type compatibility with ILE CL.

### Table 33. ILE C Data Type Compatibility with ILE CL

<table>
<thead>
<tr>
<th>ILE C declaration in prototype</th>
<th>CL</th>
<th>Length</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>char[n]</td>
<td>*CHAR LEN(&amp;N)</td>
<td>n</td>
<td>An array of characters where n=1 to 32766. A null-terminated string. For example, CHGVAR &amp;V1 VALUE (&amp;V *TCAT X'00') where &amp;V1 is one byte bigger than &amp;V.</td>
</tr>
<tr>
<td>char *</td>
<td>*LGL</td>
<td>1</td>
<td>Holds '1' or '0'.</td>
</tr>
<tr>
<td>_Packed struct {short i; char[n]}</td>
<td>Not supported.</td>
<td>n+2</td>
<td>A variable length field where i is the intended length and n is the maximum length.</td>
</tr>
<tr>
<td>integer types</td>
<td>*INT LEN(&amp;N)</td>
<td>2, 4, 8</td>
<td>A 2-, 4-, or 8- byte signed integer. (CL does not support 1 byte integer type)</td>
</tr>
<tr>
<td></td>
<td>*UINT LEN(&amp;N)</td>
<td>2, 4, 8</td>
<td>A 2-, 4-, or 8- byte unsigned integer. (CL does not support 1 byte integer type)</td>
</tr>
<tr>
<td>float constants</td>
<td>CL constants only.</td>
<td>4</td>
<td>A 4- or 8- byte floating point.</td>
</tr>
<tr>
<td>decimal(n,p)</td>
<td>*DEC</td>
<td>n/2+1</td>
<td>A packed decimal. The limit of n is 15 and p is 9.</td>
</tr>
<tr>
<td>union.element</td>
<td>Not supported.</td>
<td>element length</td>
<td>An element of a union.</td>
</tr>
<tr>
<td>struct</td>
<td>Not supported.</td>
<td>n</td>
<td>A structure. Use the _Packed qualifier on the struct.</td>
</tr>
<tr>
<td>pointer to function</td>
<td>Not supported.</td>
<td>16</td>
<td>A 16-byte pointer.</td>
</tr>
</tbody>
</table>

The following table shows the ILE C data type compatibility with OPM RPG/400®.
<table>
<thead>
<tr>
<th>ILE C declaration in prototype</th>
<th>OPM RPG/400 I spec, DS subfield columns spec</th>
<th>Length</th>
<th>Comments</th>
</tr>
</thead>
</table>
| char
char *                      | 1 10                                        | n      | An array of characters where n=1 to 32766. |
<p>| char                          | *INxxxx                                     | 1      | An Indicator that is a variable starting with *IN. |
| char[n]                      | 1 nd (d&gt;=0)                                 | n      | A zoned decimal. The limit of n is 30. |
| _Packed struct {unsigned short i; char[n]} | Not supported.                              | n+2    | A variable length field where i is the intended length and n is the maximum length. |
| _Packed struct {unsigned int i; char[n]} | Not supported.                              | n+4    | A variable length field where i is the intended length and n is the maximum length. |
| wchar_t[n]                   | Not supported.                              | 2n     | An array of UCS-2 characters. |
| _Packed struct {unsigned short i; wchar_t[n]} | Not supported.                              | 2n+2   | A variable length UCS-2 field where i is the intended length and n is the maximum length. |
| _Packed struct {unsigned int i; wchar_t[n]} | Not supported.                              | 2n+4   | A variable length UCS-2 field where i is the intended length and n is the maximum length. |
| wchar_t[n]                   | Not supported.                              | 2n     | An array of graphic characters. |
| _Packed struct {unsigned short i; wchar_t[n]} | Not supported.                              | 2n+2   | A variable length graphic field where i is the intended length and n is the maximum length. |
| _Packed struct {unsigned int i; wchar_t[n]} | Not supported.                              | 2n+4   | A variable length graphic field where i is the intended length and n is the maximum length. |
| char[n]                      | Not supported.                              | 6, 8, 10 | A date field. |
| char[n]                      | Not supported.                              | 8      | A time field. |
| char[n]                      | Not supported.                              | 26     | A time stamp field. |
| short int                    | B 1 20                                      | 2      | A 2-byte signed integer with a range of -9999 to +9999. |
| int                          | B 1 40                                      | 4      | A 4-byte signed integer with a range of -999999999 to +999999999. |
| long int                     | B 1 40                                      | 4      | A 4-byte signed integer with a range of -9999999999 to +9999999999. |
| long long int                | Not supported.                              | 8      | An 8-byte integer field. |
| struct {unsigned int : n};   | Not supported.                              | 4      | A 4-byte unsigned integer, a bitfield. |
| float                        | Not supported.                              | 4      | A 4-byte floating point. |</p>
<table>
<thead>
<tr>
<th>ILE C declaration in prototype</th>
<th>OPM RPG/400 I spec, DS subfield columns spec</th>
<th>Length</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>double</td>
<td>Not supported.</td>
<td>8</td>
<td>An 8-byte double.</td>
</tr>
<tr>
<td>long double</td>
<td>Not supported.</td>
<td>8</td>
<td>An 8-byte long double.</td>
</tr>
<tr>
<td>_Decimal32</td>
<td>Not supported.</td>
<td>4</td>
<td>A 4-byte decimal floating point.</td>
</tr>
<tr>
<td>_Decimal64</td>
<td>Not supported.</td>
<td>8</td>
<td>An 8-byte decimal floating point.</td>
</tr>
<tr>
<td>_Decimal128</td>
<td>Not supported.</td>
<td>16</td>
<td>A 16-byte decimal floating point.</td>
</tr>
<tr>
<td>enum</td>
<td>Not supported.</td>
<td>1, 2, 4</td>
<td>Enumeration.</td>
</tr>
<tr>
<td>*</td>
<td>Not supported.</td>
<td>16</td>
<td>A pointer.</td>
</tr>
<tr>
<td>decimal(n,p)</td>
<td>P 1 n/2+1d</td>
<td>n/2+1</td>
<td>A packed decimal. n must be less than or equal to 30.</td>
</tr>
<tr>
<td>union.element</td>
<td>data structure subfield starting at position 1</td>
<td>element length</td>
<td>An element of a union.</td>
</tr>
<tr>
<td>data_type[n]</td>
<td>E-SPEC array</td>
<td>16</td>
<td>An array to which C passes a pointer.</td>
</tr>
<tr>
<td>struct</td>
<td>data structure</td>
<td>n</td>
<td>A structure. Use the _Packed qualifier on the struct.</td>
</tr>
<tr>
<td>pointer to function</td>
<td>Not supported.</td>
<td>16</td>
<td>A 16 byte pointer.</td>
</tr>
</tbody>
</table>

The following table shows the ILE C data type compatibility with OPM COBOL/400.

<table>
<thead>
<tr>
<th>ILE C declaration in prototype</th>
<th>OPM COBOL LINKAGE SECTION</th>
<th>Length</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>char[n]</td>
<td>PIC X(n).</td>
<td>n</td>
<td>An array of characters where n=1 to 3,000,000</td>
</tr>
<tr>
<td>char *</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>char</td>
<td>PIC 1 INDIC ..</td>
<td>1</td>
<td>An indicator.</td>
</tr>
<tr>
<td>char[n]</td>
<td>PIC S9(n) USAGE IS DISPLAY</td>
<td>n</td>
<td>A zoned decimal. The limit of n is 18.</td>
</tr>
<tr>
<td>_Packed struct {short i; char[n]}</td>
<td>05 VL-FIELD. 10 i PIC S9(4) COMP-4. 10 data PIC X(n).</td>
<td>n+2</td>
<td>A variable length field where i is the intended length and n is the maximum length.</td>
</tr>
<tr>
<td>char[n]</td>
<td>PIC X(n).</td>
<td>6, 8, 10</td>
<td>A date field.</td>
</tr>
<tr>
<td>char[n]</td>
<td>PIC X(n).</td>
<td>8</td>
<td>A time field.</td>
</tr>
<tr>
<td>char[n]</td>
<td>PIC X(n).</td>
<td>26</td>
<td>A time stamp field.</td>
</tr>
<tr>
<td>short int</td>
<td>PIC S9(4) COMP-4.</td>
<td>2</td>
<td>A 2 byte signed integer with a range of -9999 to +9999.</td>
</tr>
<tr>
<td>int</td>
<td>PIC S9(9) COMP-4.</td>
<td>4</td>
<td>A 4-byte signed integer with a range of -9999999999 to +9999999999.</td>
</tr>
</tbody>
</table>
### Table 35. ILE C Data Type Compatibility with OPM COBOL/400 (continued)

<table>
<thead>
<tr>
<th>ILE C declaration in prototype</th>
<th>OPM COBOL LINKAGE SECTION</th>
<th>Length</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>long int</td>
<td>PIC S9(9) COMP-4.</td>
<td>4</td>
<td>A 4-byte signed integer with a range of -999999999 to +999999999.</td>
</tr>
<tr>
<td>struct {unsigned int : n}x;</td>
<td>PIC 9(9) COMP-4. PIC X(4).</td>
<td>4</td>
<td>Bitfields can be manipulated using hex literals.</td>
</tr>
<tr>
<td>float</td>
<td>Not supported.</td>
<td>4</td>
<td>A 4-byte floating point.</td>
</tr>
<tr>
<td>double</td>
<td>Not supported.</td>
<td>8</td>
<td>An 8-byte double.</td>
</tr>
<tr>
<td>long double</td>
<td>Not supported.</td>
<td>8</td>
<td>An 8-byte long double.</td>
</tr>
<tr>
<td>enum</td>
<td>Not supported.</td>
<td>1, 2, 4</td>
<td>Enumeration.</td>
</tr>
<tr>
<td>*</td>
<td>USAGE IS POINTER</td>
<td>16</td>
<td>A pointer.</td>
</tr>
<tr>
<td>decimal(n,p)</td>
<td>PIC S9(n-p)V9(p) COMP-3</td>
<td>n/2+1</td>
<td>A packed decimal. The limits of n and p are 18.</td>
</tr>
<tr>
<td>union.element</td>
<td>REDEFINES element length</td>
<td>element length</td>
<td>An element of a union.</td>
</tr>
<tr>
<td>data_type[n]</td>
<td>OCCURS</td>
<td>16</td>
<td>An array to which C passes a pointer.</td>
</tr>
<tr>
<td>struct</td>
<td>01 record</td>
<td>n</td>
<td>A structure. Use the _Packed qualifier on the struct. Structures passed should be passed as a pointer to the structure if you want to change the contents of the structure.</td>
</tr>
<tr>
<td>pointer to function</td>
<td>Not supported.</td>
<td>16</td>
<td>A 16-byte pointer.</td>
</tr>
<tr>
<td>Not supported.</td>
<td>PIC S9(18) COMP-4.</td>
<td>8</td>
<td>An 8 byte integer.</td>
</tr>
</tbody>
</table>

The following table shows the ILE C data type compatibility with CL.

### Table 36. ILE C Data Type Compatibility with CL

<table>
<thead>
<tr>
<th>ILE C declaration in prototype</th>
<th>CL</th>
<th>Length</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>char[n] char *</td>
<td>*CHAR LEN(&amp;N)</td>
<td>n</td>
<td>An array of characters where n=1 to 32766. A null terminated string. For example, CHGVAR &amp;V1 VALUE (&amp;V *TCAT X'00') where &amp;V1 is one byte bigger than &amp;V. The limit of n is 9999.</td>
</tr>
<tr>
<td>char</td>
<td>*LGL</td>
<td>1</td>
<td>Holds '1' or '0'.</td>
</tr>
<tr>
<td>_Packed struct {short i; char[n]}</td>
<td>Not supported.</td>
<td>n+2</td>
<td>A variable length field where i is the intended length and n is the maximum length.</td>
</tr>
</tbody>
</table>
Table 36. ILE C Data Type Compatibility with CL (continued)

<table>
<thead>
<tr>
<th>ILE C declaration in prototype</th>
<th>CL</th>
<th>Length</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>integer types</td>
<td>*INT LEN(&amp;N)</td>
<td>2, 4</td>
<td>A 2- or 4- byte signed integer. (CL does not support 1 byte integer type)</td>
</tr>
<tr>
<td></td>
<td>*UINT LEN(&amp;N)</td>
<td>2, 4</td>
<td>A 2- or 4- byte unsigned integer. (CL does not support 1 byte integer type)</td>
</tr>
<tr>
<td>float constants</td>
<td>CL constants only.</td>
<td>4</td>
<td>A 4- or 8- byte floating point.</td>
</tr>
<tr>
<td>decimal(n,p)</td>
<td>*DEC</td>
<td>n/2+1</td>
<td>A packed decimal. The limit of n is 15 and p is 9.</td>
</tr>
<tr>
<td>union.element</td>
<td>Not supported.</td>
<td>element length</td>
<td>An element of a union.</td>
</tr>
<tr>
<td>struct</td>
<td>Not supported.</td>
<td>n</td>
<td>A structure. Use the _Packed qualifier on the struct.</td>
</tr>
<tr>
<td>pointer to function</td>
<td>Not supported.</td>
<td>16</td>
<td>A 16-byte pointer.</td>
</tr>
</tbody>
</table>

The following table shows how arguments are passed from a command line CL call to an ILE C program.

Table 37. Arguments Passed From a Command Line CL Call to an ILE C Program

<table>
<thead>
<tr>
<th>Command Line Argument</th>
<th>Argv Array</th>
<th>ILE C Arguments</th>
</tr>
</thead>
<tbody>
<tr>
<td>argv[0]</td>
<td>&quot;LIB/PGMNAME&quot;</td>
<td></td>
</tr>
<tr>
<td>argv[1..255]</td>
<td>normal parameters</td>
<td></td>
</tr>
<tr>
<td>'123.4'</td>
<td>argv[1]</td>
<td>&quot;123.4&quot;</td>
</tr>
<tr>
<td>123.4</td>
<td>argv[2]</td>
<td>0000000123.40000D</td>
</tr>
<tr>
<td>'Hi'</td>
<td>argv[3]</td>
<td>&quot;Hi&quot;</td>
</tr>
<tr>
<td>Lo</td>
<td>argv[4]</td>
<td>&quot;LO&quot;</td>
</tr>
</tbody>
</table>

A CL character array (string) will not be NULL-ended when passed to an ILE C program. A C program that will receive such arguments from a CL program should not expect the strings to be NULL-ended. You can use the QCMDEXC to ensure that all the arguments will be NULL-ended.

The following table shows how CL constants are passed from a compiled CL program to an ILE C program.

Table 38. CL Constants Passed from a Compiled CL Program to an ILE C Program

<table>
<thead>
<tr>
<th>Compile CL Program Argument</th>
<th>Argv Array</th>
<th>ILE C Arguments</th>
</tr>
</thead>
<tbody>
<tr>
<td>argv[0]</td>
<td>&quot;LIB/PGMNAME&quot;</td>
<td></td>
</tr>
<tr>
<td>argv[1..255]</td>
<td>normal parameters</td>
<td></td>
</tr>
<tr>
<td>'123.4'</td>
<td>argv[1]</td>
<td>&quot;123.4&quot;</td>
</tr>
<tr>
<td>123.4</td>
<td>argv[2]</td>
<td>0000000123.40000D</td>
</tr>
<tr>
<td>'Hi'</td>
<td>argv[3]</td>
<td>&quot;Hi&quot;</td>
</tr>
<tr>
<td>Lo</td>
<td>argv[4]</td>
<td>&quot;LO&quot;</td>
</tr>
</tbody>
</table>

A command processing program (CPP) passes CL constants as defined in Table 38 on page 563. You define an ILE C program as a command processing program when you create your own CL command with the Create Command (CRTCMD) command to call the ILE C program.
The following table shows how CL variables are passed from a compiled CL program to an ILE C program. All arguments are passed by reference from CL to C.

**Table 39. CL Variables Passed from a Compiled CL Program to an ILE C Program**

<table>
<thead>
<tr>
<th>CL Variables</th>
<th>ILE C Arguments</th>
</tr>
</thead>
<tbody>
<tr>
<td>DCL VAR(&amp;v) TYPE(*CHAR) LEN(10) VALUE('123.4')</td>
<td>123.4</td>
</tr>
<tr>
<td>DCL VAR(&amp;d) TYPE(*DEC) LEN(10) VALUE(123.4)</td>
<td>0000000123.40000D</td>
</tr>
<tr>
<td>DCL VAR(&amp;h) TYPE(*CHAR) LEN(10) VALUE('Hi')</td>
<td>Hi</td>
</tr>
<tr>
<td>DCL VAR(&amp;i) TYPE(*CHAR) LEN(10) VALUE(Lo)</td>
<td>Lo</td>
</tr>
<tr>
<td>DCL VAR(&amp;j) TYPE(*LGL) LEN(1) VALUE('1')</td>
<td>1</td>
</tr>
</tbody>
</table>

CL variables and numeric constants are not passed to an ILE C program with null-ended strings. Character constants and logical literals are passed as null-ended strings, but are not padded with blanks. Numeric constraints such as packed decimals are passed as 15,5 (8 bytes).

**Runtime Character Set**

Each EBCDIC CCSID consists of two character types: invariant characters and variant characters.

The following table identifies the hexadecimal representation of the invariant characters in the C character set.

**Table 40. Invariant Characters**

<table>
<thead>
<tr>
<th>.</th>
<th>&lt;</th>
<th>(</th>
<th>+</th>
<th>&amp;</th>
<th>*</th>
<th>)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x4b</td>
<td>0x4c</td>
<td>0x4d</td>
<td>0x4e</td>
<td>0x50</td>
<td>0x5c</td>
<td>0x5d</td>
</tr>
<tr>
<td>0x60</td>
<td>0x6a</td>
<td>0x6b</td>
<td>0x6c</td>
<td>0x6d</td>
<td>0x6e</td>
<td>0x6f</td>
</tr>
<tr>
<td>0x7c</td>
<td>0x7d</td>
<td>0x7e</td>
<td>0x7f</td>
<td>0x81 - 0x89</td>
<td>0x91 - 0x99</td>
<td>0xa2 - 0xa9</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>J-R</th>
<th>S-Z</th>
<th>0-9</th>
<th>‘\a’</th>
<th>‘\b’</th>
<th>‘\t’</th>
<th>‘\v’</th>
<th>‘\f’</th>
</tr>
</thead>
<tbody>
<tr>
<td>0xd1 - 0xd9</td>
<td>0xe2 - 0xe9</td>
<td>0xf0 - 0xf9</td>
<td>0x2f</td>
<td>0x16</td>
<td>0x05</td>
<td>0x0b</td>
<td>0xc0</td>
</tr>
</tbody>
</table>

**Note:** Not all EBCDIC character sets have all invariant characters at the invariant code points. Here are the exceptions:

- Code page 290, used in Japanese CCSIDs 290, 930, and 5026, has the lowercase Latin characters a-z in a nonstandard position.
- Code page 420, used in some Arabic CCSIDs, does not have the back quotation mark (‘) whose hexadecimal value is 0x7a.
- Code page 423, used in some older Greek CCSIDs, does not have the ampersand (&) whose hexadecimal value is 0x50.
• Code pages 905 and 1026, both used in some Turkish CCSIDs, have a hexadecimal value of 0xfc for the double quotation mark instead of the invariant hexadecimal value of 0x7f.

The following table identifies the hexadecimal representation of the variant characters in the C character set for the most commonly used CCSIDs.

<table>
<thead>
<tr>
<th>CC-SID</th>
<th>!</th>
<th>!</th>
<th>~</th>
<th>\</th>
<th>`</th>
<th>#</th>
<th>~</th>
<th>[</th>
<th>]</th>
<th>^</th>
<th>{</th>
<th>}</th>
<th>/</th>
<th>£</th>
<th>$</th>
</tr>
</thead>
<tbody>
<tr>
<td>037</td>
<td>0x4f</td>
<td>0x5a</td>
<td>0x5f</td>
<td>0xe0</td>
<td>0x79</td>
<td>0x7b</td>
<td>0xa1</td>
<td>0xba</td>
<td>0xbb</td>
<td>0xb0</td>
<td>0xc0</td>
<td>0xd0</td>
<td>0x61</td>
<td>0x4a</td>
<td>0x5b</td>
</tr>
<tr>
<td>256</td>
<td>0xbb</td>
<td>0x4f</td>
<td>0xba</td>
<td>0xe0</td>
<td>0x79</td>
<td>0x7b</td>
<td>0xa1</td>
<td>0x4a</td>
<td>0x5a</td>
<td>0x5f</td>
<td>0xc0</td>
<td>0xd0</td>
<td>0x61</td>
<td>0xb0</td>
<td>0x5b</td>
</tr>
<tr>
<td>273</td>
<td>0xbb</td>
<td>0x4f</td>
<td>0xba</td>
<td>0xec</td>
<td>0x79</td>
<td>0x7b</td>
<td>0x59</td>
<td>0x63</td>
<td>0xfc</td>
<td>0x5f</td>
<td>0xc0</td>
<td>0xd0</td>
<td>0x61</td>
<td>0xb0</td>
<td>0x5b</td>
</tr>
<tr>
<td>277</td>
<td>0xbb</td>
<td>0x4f</td>
<td>0xba</td>
<td>0xe0</td>
<td>0x79</td>
<td>0x4a</td>
<td>0xdc</td>
<td>0x9e</td>
<td>0x9f</td>
<td>0x5f</td>
<td>0xc0</td>
<td>0x47</td>
<td>0x61</td>
<td>0xb0</td>
<td>0x67</td>
</tr>
<tr>
<td>278</td>
<td>0xbb</td>
<td>0x4f</td>
<td>0xba</td>
<td>0x71</td>
<td>0x51</td>
<td>0x63</td>
<td>0xdc</td>
<td>0xb5</td>
<td>0x9f</td>
<td>0x5f</td>
<td>0xc0</td>
<td>0x47</td>
<td>0x61</td>
<td>0xb2</td>
<td>0x67</td>
</tr>
<tr>
<td>280</td>
<td>0xbb</td>
<td>0x4f</td>
<td>0xba</td>
<td>0x48</td>
<td>0xdd</td>
<td>0xb1</td>
<td>0x58</td>
<td>0x90</td>
<td>0x51</td>
<td>0x5f</td>
<td>0x44</td>
<td>0x45</td>
<td>0x61</td>
<td>0xb0</td>
<td>0x5b</td>
</tr>
<tr>
<td>284</td>
<td>0x4f</td>
<td>0xbb</td>
<td>0x5f</td>
<td>0xe0</td>
<td>0x79</td>
<td>0x69</td>
<td>0xbd</td>
<td>0x4a</td>
<td>0x5a</td>
<td>0xba</td>
<td>0xc0</td>
<td>0x4d</td>
<td>0x61</td>
<td>0xb0</td>
<td>0x5b</td>
</tr>
<tr>
<td>285</td>
<td>0x4f</td>
<td>0x5a</td>
<td>0x5f</td>
<td>0xe0</td>
<td>0x79</td>
<td>0x7b</td>
<td>0xbc</td>
<td>0xb1</td>
<td>0xbb</td>
<td>0xba</td>
<td>0xc0</td>
<td>0xd0</td>
<td>0x61</td>
<td>0xb0</td>
<td>0x4a</td>
</tr>
<tr>
<td>297</td>
<td>0xbb</td>
<td>0x4f</td>
<td>0xba</td>
<td>0x48</td>
<td>0xa0</td>
<td>0xb1</td>
<td>0xbd</td>
<td>0x90</td>
<td>0x65</td>
<td>0x5f</td>
<td>0x51</td>
<td>0x54</td>
<td>0x61</td>
<td>0xb0</td>
<td>0x5b</td>
</tr>
<tr>
<td>500</td>
<td>0xbb</td>
<td>0x4f</td>
<td>0xba</td>
<td>0xe0</td>
<td>0x79</td>
<td>0x7b</td>
<td>0xa1</td>
<td>0x4a</td>
<td>0x5a</td>
<td>0x5f</td>
<td>0xc0</td>
<td>0xd0</td>
<td>0x61</td>
<td>0xb0</td>
<td>0x5b</td>
</tr>
</tbody>
</table>

See the i5/OS globalization topic for more information about coding variant characters in the other IBM CCSIDs.

Understanding CCSIDs and Locales

CCSIDs of Characters and Character Strings

Every character or character string has a CCSID associated with it. The CCSID of the character or character string depends on the origin of the data. You need to pay attention to the CCSID of a character or character string. It is also important that values are converted to the appropriate CCSID when required.

If LOCALETYPE(*LOCALEUTF) is not specified on the compilation command, the following assumptions are made:

• The CCSID of the job is the same as the CCSID of the LC_CTYPE category of the current locale.
• The CCSID of character literal values matches the CCSID of the LC_CTYPE category of the current locale.
• The CCSID of the LC_CTYPE category of the current locale is an EBCDIC CCSID.
• The CCSID that is used has all of the invariant characters in the proper positions, and some functions assume that certain variant characters have the same hexadecimal value as they would in CCSID 37.

When LOCALETYPE(*LOCALEUTF) is specified, most functions (unless otherwise specified) expect character data input in the CCSID of the LC_CTYPE category of the current locale, regardless of the source of the character data. See “Unicode Support” on page 572 for more information.

For more information about variant and invariant characters, see “Runtime Character Set” on page 564. For more information about CCSIDs, code pages, and other globalization concepts, see the i5/OS globalization topic.
Character Literal CCSID

Character literal CCSID is the CCSID of the character and character string literals in compiled source code. If a programmer does not take special action, the CCSID of these literals is set to the CCSID of the source file. The CCSID of all the literals in a compilation unit can be changed by using the TGTCCSID option on the compilation command. The #pragma convert directive can be used to change the CCSID of character and character string literals within C or C++ source code. See the ILE C/C++ Compiler Reference for more information.

If LOCALETYPE(*CLD) or LOCALETYPE(*LOCALE) is specified on the compilation command, all wide character literals will be wide EBCDIC literals in the CCSID of the source file. If LOCALETYPE(*LOCALEUCS2) is specified on the compilation command, all wide character literals will be UCS-2 literals. If LOCALETYPE(*LOCALEUTF) is specified on the compilation command, all wide characters will be UTF-32 literals.

The programmer must be aware of the CCSID of character literal values. The character literal CCSID cannot be retrieved at runtime.

Job CCSID

The CCSID of the job is always an EBCDIC CCSID. ASCII and Unicode job CCSIDs are not supported. Data read from files is sometimes in the job CCSID. Some functions (for example, getenv()) produce job CCSID output; some functions (for example, putenv()) expect job CCSID input. The CCSID used most often by the C runtime is the CCSID of the LC_CTYPE category of the current locale. If the job CCSID does not match the locale CCSID, conversion might be necessary.

Using the JOBI0400 receiver variable format, the job CCSID value can be retrieved at runtime using the QUSRJOBI API. The Default Coded Character Set ID field contains the job CCSID value.

File CCSID

When a file is opened, a CCSID is associated with it. Read operations of character and string values return data in the CCSID of the file. Write operations to the file expect the data in the CCSID of the file. The CCSID associated with a file when it is opened is dependent on the function that is used to open the file:

• catopen() function

The CCSID associated with a catalog file that is opened using catopen depends on the content of the oflag parameter. Two of the flags that can be specified for the oflag parameter are NL_CAT_JOB_MODE and NL_CAT_CTYPE_MODE. These flags are mutually exclusive.

– If NL_CAT_JOB_MODE is specified, the job CCSID is associated with the file.
– If NL_CAT_CTYPE_MODE is specified, the CCSID of the LC_CTYPE category of the current locale is associated with the file.
– If neither flag is specified, no conversion takes place and the CCSID of the returned messages is the same CCSID as that of the message file.

• fopen() function

– If LOCALETYPE(*LOCALEUTF) is not specified, then the default CCSID for a file is the job CCSID. The keyword ccsid=value, o_ccsid=value, or codepage=value can be used in the mode string on the file open command to change the CCSID associated with the file. o_ccsid=value is the recommended keyword. The standard files are always associated with the default file CCSID, so they are associated with the job CCSID.

– If LOCALETYPE(*LOCALEUTF) is specified, then the default CCSID for a file is the CCSID of the LC_CTYPE category of the current locale when the fopen() function is called. The keywords described in the previous paragraph can still be used to override the CCSID associated with the file. The standard files are always associated with the default file CCSID, so they are associated with the CCSID of the LC_CTYPE category of the current locale when they are opened.

• fclose() and freopen() functions

– If LOCALETYPE(*LOCALEUTF) is not specified, the default CCSID for a file is the job CCSID.
If SYSIFCOPT(*NOIFSIO) is specified on the compilation command, the keyword ccsid=value can be used in the mode string on the file open command to change the CCSID of data read from or written to the file. If SYSIFCOPT(*NOIFSIO) is not specified on the compilation command, the keyword ccsid=value, o_ccsid=value, or codepage=value can be used in the mode string on the file open command to change the CCSID associated with the file. o_ccsid=value is the recommended keyword.

The standard files are always associated with the default file CCSID, so they are associated with the job CCSID.

- If LOCALETYPE(*LOCALEUTF) is specified, then the default CCSID for a file is the CCSID of the LC_CTYPE category of the current locale when the fopen() or freopen() function is called. The keyword ccsid=value, o_ccsid=value, or codepage=value can still be used to override the CCSID associated with the file. The standard files are always associated with the default file CCSID, so they are associated with the CCSID of the LC_CTYPE category of the current locale when they are opened.

Locale CCSID

A CCSID is associated with each category of the locale (see “setlocale() — Set Locale” on page 366 for a list of locale categories). The most commonly used CCSID from the locale is the CCSID associated with the LC_CTYPE category of the locale. Confusion might arise if different locale categories have different CCSID values, so it is recommended that all locale categories have the same CCSID value. You can retrieve the CCSID of the LC_CTYPE category of the current locale by using the nl_langinfo() function and specifying CODESET as the nl_item. Here are some additional locale CCSID details, broken down by LOCALETYPE option specified on the compilation command:

- LOCALETYPE(*CLD)
  LOCALETYPE(*CLD) is only supported by the ILE C compiler. Many POSIX functions are not supported when LOCALETYPE(*CLD) is specified. One benefit of the LOCALETYPE(*CLD) option is that all *CLD locales are CCSID 37. A limited number of locale objects are shipped with the system that can be used with LOCALETYPE(*CLD). These objects all have the object type *CLD. To get a list of *CLD locale objects, use the following command:

  WRKOBJ OBJ(QSYS/*ALL) OBJTYPE(*CLD)

  For more information about *CLD locales, see the ILE C/C++ Compiler Reference.

- LOCALETYPE(*LOCALE)
  This is the default LOCALETYPE setting for the ILE C compiler and ILE C++ compiler. The default locale value usually has a CCSID that is equal to the job CCSID. A wide variety of locale objects exists for this setting. These locale objects have the *LOCALE object type. The LOCALETYPE(*LOCALE) option supports a larger number of CCSIDs and a larger number of functions than the LOCALETYPE(*CLD) option.

- LOCALETYPE(*LOCALEUCS2)
  This setting introduces a new set of locale categories for UCS-2 characters. These locale category names begin with the LC_UNI_ substring. The original locale categories are still present, and all the preceding notes for LOCALETYPE(*LOCALE) apply to LOCALETYPE(*LOCALEUCS2). This setting causes
wide characters to be interpreted as UCS-2 characters instead of wide EBCDIC characters. For more information, see “Unicode Support” on page 572.

• **LOCALETYPE(*LOCALEUTF)**

The CCSID of the non-wide locale categories is UTF-8 (CCSID 1208) by default, but it can be changed to have any single-byte or multibyte CCSID. The CCSID of the wide character (LC_UNI_*) locale categories is UTF-32. This setting includes limited CCSID neutrality. LOCALETYPE(*LOCALEUTF) uses locale objects of type *LOCALE. For more information, see “Unicode Support” on page 572.

### Wide Characters

The ILE C/C++ compilers support the following:

- If LOCALETYPE(*CLD) or LOCALETYPE(*LOCALE) is specified on the compilation command, wide characters are treated as 2-byte wide EBCDIC characters.
- If LOCALETYPE(*LOCALEUCS2) is specified on the compilation command, wide characters are treated as 2-byte UCS-2 characters.
- If LOCALETYPE(*LOCALEUTF) is specified on the compilation command, wide characters are treated as 4-byte UTF-32 characters.

When EBCDIC wide characters are used, the CCSID of the EBCDIC characters depends on the CCSID of the LC_CTYPE category of the current locale. See “Unicode Support” on page 572 for more information about Unicode characters.

### Wide Character Conversions to and from Single-Byte or Multibyte Characters

The character conversion functions examine the CCSID setting for the LC_CTYPE category of the current locale to determine whether single-byte or multibyte characters are expected for the conversion from or to wide characters.

The handling of wide character conversions (to and from single-byte or multibyte character strings) is dependent on the LOCALETYPE parameter value specified on the compilation command. The handling depends on the shift state of the single-byte or multibyte character string. The mbtowc, mbstowcs, wctomb, and wcstombs functions maintain an internal shift state variable. The mbrtowc, mbsrtowcs, wcrtomb, and wcsrtombs functions allow the shift state variable to be passed as a parameter. The second set of functions is recommended because they are more versatile and are also threadsafe.

**LOCALETYPE(*CLD) and LOCALETYPE(*LOCALE) behavior**

When converting from a single-byte CCSID to wide EBCDIC, the wide EBCDIC character is constructed by adding a zero byte to the single-byte character. For example, the single-byte CCSID 37 character A (hexadecimal value 0xC1) would have the hexadecimal value 0x00C1 when it is converted to a wide EBCDIC character.

When converting from a multibyte CCSID to wide EBCDIC, the conversion method depends on the shift state of the input string. In the initial shift state, characters are read exactly as if they were single-byte characters until a shift-out character (hexadecimal value 0x0E) is read. This character indicates a shift to double-byte shift state. In the double-byte shift state, 2 bytes are read at a time: the first byte makes up the first byte of the EBCDIC wide character and the second byte will be the second byte of the EBCDIC wide character. If the shift-in character (hexadecimal value 0x0F) is encountered, the function returns to the initial shift state parsing. For example, the multibyte string represented by the hexadecimal value C10E43DA0FC2 is translated to the EBCDIC wide character string with the hexadecimal value 00C143DA00C2.

When converting from wide EBCDIC to a single-byte CCSID, if the character has a hexadecimal value greater than 0x00FF, EOF is returned; otherwise, the top byte is truncated and the lower byte is returned. For example, the wide EBCDIC character with the hexadecimal value 0x00C1 is converted to the single-byte character whose hexadecimal value is 0xC1.

When converting from wide EBCDIC to a multibyte CCSID, the conversion method is determined by the shift state of the output string:
• If the output string is in the initial shift state, any EBCDIC wide character with a hexadecimal value that is less than or equal to 0x00FF is truncated to 1 byte and placed in the output string.

• If the output string is in the initial shift state, any EBCDIC wide character with a value that is greater than 0x00FF causes a shift-out character (hexadecimal value 0x0E) to be generated in the output string. The shift state of the output string is updated to double-byte, and both bytes of the EBCDIC wide character are copied to the output string.

• If the output string is in the double-byte shift state and an EBCDIC wide character whose hexadecimal value is less than or equal to 0x00FF is encountered, a shift-in character (hexadecimal value 0x0F) is placed in the output string. The shift-in character is followed by the value of the EBCDIC wide character that is truncated to 1 byte. The shift state of the output string is changed to single-byte.

• If the output string is in the double-byte shift state and an EBCDIC wide character whose value is greater than 0x00FF is encountered, the 2 bytes of the EBCDIC wide character are copied to the output string.

For example, the EBCDIC wide character string with the hexadecimal value 00C143DA00C2 is translated to a multibyte string with the hexadecimal value C10E43DA0FC2.

**LOCALETYPEn(LOCALEUCS2) and LOCALETYPEn(LOCALEUTF) behavior**

If LOCALETYPEn(LOCALEUCS2) is specified on the compilation command, wide character values are 2-byte UCS-2 values. All conversions between UCS-2 strings and single-byte or multibyte strings are conducted as if the iconv() function were used. CCSID 13488 is used for the UCS-2 string, and the CCSID of the LC_CTYPE category of the current locale is used for the single-byte or multibyte string.

If LOCALETYPEn(LOCALEUTF) is specified on the compilation command, wide character values are 4-byte UTF-32 values. All conversions between UTF-32 strings and single-byte or multibyte strings are conducted as if the iconv() function were used. UTF-32 is not supported by the iconv() function.

**Wide Characters and File I/O**

**Wide character write functions**

Several functions, including fprintf, vfprintf, vwprintf, wprintf, fputwc, fputws, putwc, putwchar, and ungetwc can be used to write wide characters to a file. These functions are not available when either LOCALETYPEn(LOCLD) or SYSIFCOPT( NOIFSIO) is specified on the compilation command.

If LOCALETYPEn(LOCALE) is specified on the compilation command, the wide characters that are written are assumed to be wide character equivalents of the code points in the file CCSID. The CCSID of the file is assumed to be a single or multibyte EBCDIC CCSID.

If LOCALETYPEn(LOCALEUCS2) or LOCALETYPEn(LOCALEUTF) is specified on the compilation command, the wide characters that are being written are assumed to be Unicode characters. For LOCALETYPEn(LOCALEUCS2), they are assumed to be 2-byte UCS-2 characters. For LOCALETYPEn(LOCALEUTF), they are assumed to be 4-byte UTF-32 characters. If the file that is being written to is not one of the standard files, the Unicode characters are then written directly to the file as if the file had been opened for writing in binary mode. The CCSID of the file is assumed to be a Unicode CCSID that matches the locale setting. If the file that is being written to is a standard file, the Unicode input is converted to the CCSID of the job before being written to the file.

**Non-wide character write functions**

The non-wide character write functions (fprintf, vfprintf, vprintf, and printf) can take a wide character as input.

In all cases, the wide characters are converted to multibyte character strings in the CCSID of the LC_CTYPE category of the current locale as if the wctomb function or the wcstombs function were used. The file CCSID is assumed to match the CCSID of the LC_CTYPE category of the current locale.
If LOCALETYPE(*LOCALEUTF) is specified on the compilation command and the file that is being written to is a standard file, the output will automatically be converted from the CCSID of the LC_CTYPE category of the current locale to the CCSID of the file (which usually matches the job CCSID).

**Wide character read functions**

The functions that can read wide characters from a file include `fgetwc`, `fgetws`, `fwscanf`, `getwc`, `getwchar`, `vfscanf`, `vwscanf`, and `wscanf`. These functions are not available when either LOCALETYPE(*CLD) or SYSIFCOPT(*NOIFSIO) is specified on the compilation command.

If LOCALETYPE(*LOCALE) is specified on the compilation command, the wide characters read from the file are assumed to be EBCDIC wide character equivalents of the code points in the file CCSID.

If LOCALETYPE(*LOCALEUCS2) or LOCALETYPE(*LOCALEUTF) is specified on the compilation command, the input wide characters and the characters in the file are assumed to be Unicode characters. For LOCALETYPE(*LOCALEUCS2), they are assumed to be 2-byte UCS-2 characters. For LOCALETYPE(*LOCALEUTF), they are assumed to be 4-byte UTF-32 characters. If the file that is being read is not one of the standard files, the Unicode characters are read directly from the file as if the file had been opened in binary mode. The CCSID of the file is assumed to be a Unicode CCSID that matches the locale setting. If the file that is being read is a standard file, then the job CCSID input that is read from the file is converted to the appropriate Unicode CCSID.

**Non-wide character read functions**

The non-wide character read functions (`fscanf`, `scanf`, `vfscanf`, and `vscanf`) can produce a wide character as output.

In all cases, the wide characters are converted from multibyte character strings in the CCSID of the LC_CTYPE category of the current locale to the appropriate wide character type for the locale setting as if the `mbtowc` function or the `mbstowcs` function were used.

**Other ILE Languages**

The standard C/C++ library functions are easily accessible in the C and C++ languages if you include the appropriate header files and use the appropriate C or C++ compilation command. The functions are also accessible from other ILE languages, such as RPG, COBOL, and CL, although no header files are provided for these languages. An additional consideration exists for those functions which are locale sensitive (that is, dependent upon the current locale). When you use the C or C++ compiler, the default locale is loaded automatically at program startup time. When you use any of the C/C++ library functions from a different language, a call to `setlocale()` should be added when the application starts to ensure that the proper locale is loaded. Here is a table which describes the correct call to `setlocale()` based on the desired LOCALETYPE.

<table>
<thead>
<tr>
<th>C/C++ compiler option</th>
<th>Function call</th>
</tr>
</thead>
<tbody>
<tr>
<td>LOCALETYPE(*CLD)</td>
<td><code>setlocale(LC_ALL, &quot;&quot;)</code></td>
</tr>
<tr>
<td>LOCALETYPE(*LOCALE)</td>
<td><code>_C_PSX_setlocale(LC_ALL, &quot;)&quot;)</code></td>
</tr>
<tr>
<td>LOCALETYPE(*LOCALEUCS2)</td>
<td><code>_UCS2_setlocale(LC_ALL, &quot;)&quot;)</code></td>
</tr>
<tr>
<td>LOCALETYPE(*LOCALEUTF)</td>
<td><code>_C_UTF_setlocale(LC_ALL, &quot;)&quot;)</code></td>
</tr>
</tbody>
</table>

**Asynchronous Signal Model**

The Asynchronous Signal Model (ASM) is used when the SYSIFCOPT(*ASYNCSIGNAL) option is specified on the Create C Module (CRTCMOD) or Create Bound C Program (CRTBNDC) compilation command. The ASM is also used when the RTBND(*LLP64) option is specified on the Create C++ Module (CRTCPPMOD) or Create Bound C++ Program (CRTBNDCPP) compilation command. It is intended for compatibility with applications ported from the UNIX operating system. For modules that use the ASM, the `signal()` and...
raise() functions are implemented using the Signal APIs described in the Application programming interfaces topic under the Programming heading in the Information Center.

Operating system exceptions sent to an ASM module or program are converted to asynchronous signals. The exceptions are processed by an asynchronous signal handler.

Modules compiled to use the ASM can be intermixed with modules using the Original Signal Model (OSM) in the same processes, programs, and service programs. There are several differences between the two signal models:

- The OSM is based on exceptions, while the ASM is based on asynchronous signals.
- Under the OSM, the signal vector and signal mask are scoped to the activation group. Under the ASM, there is one signal vector per process and one signal mask per thread. Both types of signal vectors and signal masks are maintained at runtime.
- The same asynchronous signal vector and signal masks are operated on by all ASM modules in a thread, regardless of the activation group the signal vector and signal masks belong to. You must save and restore the state of the signal vector and signal masks to ensure that they are not changed by any ASM modules. The OSM does not use the asynchronous signal vector and signal masks.
- Signals that are raised by OSM modules are sent as exceptions. Under the OSM, the exceptions are received and handled by the _C_exception_router function, which directly calls the OSM signal handler of the user.

Asynchronous signals are not mapped to exceptions, and are not handled by signal handlers that are registered under the OSM. Under the ASM, the exceptions are received and handled by the _C_async_exception_router function, which maps the exception to an asynchronous signal. An ASM signal handler receives control from the operating system asynchronous signal component.

When an OSM module raises a signal, the generated exception percolates up the call stack until it finds an exception monitor. If the previous call is an OSM function, the _C_exception_router catches the exception and performs the OSM signal action. The ASM signal handler does not receive the signal.

If the previous call is an ASM function, the _C_async_exception_router handles the exception and maps it to an asynchronous signal. The handling of the asynchronous signal then depends on the asynchronous signal vector and mask state of the thread, as defined in the Signal management topic.

If the previous call is an ASM function within a different program or service program, one of two actions occurs. If the OSM program that raises the signal is running in the same activation group with the ASM program, the exception is mapped to an asynchronous signal using the mapping described previously. The signal ID is preserved when the exception is mapped to a signal. So, signal handlers that are registered with the asynchronous signal model are able to receive signals generated under the original signal model. This approach can be used to integrate two programs with different signal models.

If the OSM program that raises the signal is running in a different activation group than the ASM program, any signal that is unmonitored in that activation group causes the termination of that program and activation group. The unmonitored signal is then percolated to the calling program as a CEE9901 exception. The CEE9901 exception is mapped to a SIGSEGV asynchronous signal.

- Under the ASM, the C functions raise() and signal() are integrated with the system signal functions, such as kill() and sigaction(). These two sets of APIs can be used interchangeably. This cannot be done under the OSM.
- A user-specified exception monitor established with #pragma exception_handler has precedence over the compiler-generated monitor, which calls _C_async_exception_router. In some situations, this precedence enables you to bypass the compiler-generated monitor, which invokes _C_async_exception_router.
- The _GetExcData() function is not available under the ASM to retrieve the exception ID associated with the signal. However, if an extended signal handler is established using the sigaction() function, it can access the exception information from the signal-specific data structure. For more information, see “_GetExcData() — Get Exception Data” on page 176.
**Unicode Support**

The Unicode Standard is a standardized character code designed to encode international texts for display and storage. It uses a unique 16- or 32-bit value to represent each individual character, regardless of platform, language, or program. Using Unicode, you can develop a software product that will work with various platforms, languages, and countries or regions. Unicode also allows data to be transported through many different systems.

There are two different forms of Unicode support available from the compiler and runtime. This section describes the two forms of Unicode support as well as some of the features of and considerations for using that support. To obtain additional information about Unicode, visit the Unicode Home Page at www.unicode.org.

The first type of Unicode support is UCS-2 support. When the LOCALETYPE(*LOCALEUCS2) option is specified on the compilation command, the compiler and runtime use wide characters (that is, characters of the wchar_t type) and wide character strings (that is, strings of the wchar_t * type) that represent 2-byte Unicode characters. Narrow (non-wide) characters and narrow character strings represent EBCDIC characters, just as they do when the UCS-2 support is not enabled. The Unicode characters represent codepoints in CCSID 13488.

The second type of Unicode support is UTF-8 or UTF-32 support (also known as UTF support). When the LOCALETYPE(*LOCALEUTF) option is specified on the compilation command, the compiler and runtime use wide characters and wide character strings that represent 4-byte Unicode characters. Each 4-byte character represents a single UTF-32 character. Narrow characters and narrow character strings represent UTF-8 characters. Each UTF-8 character is from 1 to 4 bytes in size. Most normal characters are a single byte in size, and, in fact, all 7-bit ASCII characters map directly to UTF-8 and are 1 byte in size. The UTF-8 characters represent codepoints in CCSID 1208.

When the UTF support is enabled, not only do the wide characters become UTF-32 Unicode, but the narrow characters become UTF-8 Unicode as well. As an example, consider the following HelloWorld program.

```c
#include <stdio.h>
int main() {
  printf("Hello World\n");
  return 0;
}
```

When this program is compiled with UTF support, the character string is stored within the program as UTF-8 characters and not EBCDIC characters. The printf() function knows this and is able to parse the UTF-8 characters and generate the output as expected. However, if this program called some other user-supplied function that did not know how to handle UTF-8 characters, the other function might yield incorrect results or behavior.

**Reasons to Use Unicode Support**

You might want to use Unicode support for your application in two situations. The first situation is if your application is an international application and requires support for several different languages. The Unicode character set provides an easy way to allow a single application to handle any language or character set. The application can perform all input, processing, and output using Unicode characters. Another situation for using Unicode support is for porting a 7-bit ASCII application. Because the UTF-8 character set is a superset of 7-bit ASCII, an ASCII application can be ported more easily to a UTF-8 environment than to an EBCDIC environment.

**Pseudo-CCSID Neutrality**

When a program is compiled with UTF support, the runtime allows more than just UTF-8 characters, and it essentially becomes CCSID neutral. The runtime handles whatever CCSID is contained within the current locale. By default, when a program is compiled with UTF support, the locale that is loaded is a UTF-8 (CCSID 1208) locale. This allows the runtime to handle CCSID 1208. If the setlocale() function is
called to set the locale to an EBCDIC locale (for example, a CCSID 37 locale), the runtime handles CCSID 37. This, along with the #pragma convert support within the compiler, can be used to provide international application support. Here is an example:

```c
#include <stdio.h>
#include <locale.h>

int main() {
    /* This string is in CCSID 1208 */
    printf("Hello World\n");
    /* Change locale to a CCSID 37 locale */
    setlocale(LC_ALL, "/QSYS.LIB/EN_US.LOCALE");
    #pragma convert(37)
    /* This string is in CCSID 37 */
    printf("Hello World\n");
    return 0;
}
```

### Unicode from Other ILE Languages

The Unicode functions are easily accessible in the C and C++ languages if you include the appropriate header files and use the appropriate LOCALETYPE option on the C or C++ compilation command. The Unicode functions are accessible from other ILE languages, such as RPG, COBOL, and CL, although no header files are provided for these languages.

The following table shows the functions added for UCS-2 support. The support functions have a prefix of _UCS2_ or _C_UCS2_ added to the standard function name. The Unicode function has the same parameters as the standard (non-Unicode) function.

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>_C_UCS2_bttowc</td>
<td>_C_UCS2_iswpunct</td>
</tr>
<tr>
<td>_C_UCS2_fgetwc</td>
<td>_C_UCS2_iswspace</td>
</tr>
<tr>
<td>_C_UCS2_fgetws</td>
<td>_C_UCS2_iswuppper</td>
</tr>
<tr>
<td>_C_UCS2_fprintf</td>
<td>_C_UCS2_iswdigit</td>
</tr>
<tr>
<td>_C_UCS2_fputwc</td>
<td>_C_UCS2_mblen</td>
</tr>
<tr>
<td>_C_UCS2_fputws</td>
<td>_C_UCS2_mbrlen</td>
</tr>
<tr>
<td>_C_UCS2_fscanf</td>
<td>_C_UCS2_mbinit</td>
</tr>
<tr>
<td>_C_UCS2_fwprintf</td>
<td>_C_UCS2_vfprintf</td>
</tr>
<tr>
<td>_C_UCS2_fwscanf</td>
<td>_C_UCS2_vfscanf</td>
</tr>
<tr>
<td>_C_UCS2_getwc</td>
<td>_C_UCS2_vprintf</td>
</tr>
<tr>
<td>_C_UCS2_getwchar</td>
<td>_C_UCS2_vswprintf</td>
</tr>
<tr>
<td>_C_UCS2_isalnum</td>
<td>_C_UCS2_vswscanf</td>
</tr>
<tr>
<td>_C_UCS2_isalpha</td>
<td>_C_UCS2_vswscnaf</td>
</tr>
<tr>
<td>_C_UCS2_isblank</td>
<td>_C_UCS2_vswscanf</td>
</tr>
<tr>
<td>_C_UCS2_iscntrl</td>
<td>_C_UCS2_vswscnaf</td>
</tr>
<tr>
<td>_C_UCS2_isdigit</td>
<td>_C_UCS2_vswscnaf</td>
</tr>
<tr>
<td>_C_UCS2_isgraph</td>
<td>_C_UCS2_vswscnaf</td>
</tr>
<tr>
<td>_C_UCS2_islower</td>
<td>_C_UCS2_vswscnaf</td>
</tr>
<tr>
<td>_C_UCS2_isprint</td>
<td>_C_UCS2_vswscnaf</td>
</tr>
<tr>
<td>_C_UCS2_isupper</td>
<td>_C_UCS2_vswscnaf</td>
</tr>
<tr>
<td>_C_UCS2_iswgraph</td>
<td>_C_UCS2_vswscnaf</td>
</tr>
<tr>
<td>_C_UCS2_iswlower</td>
<td>_C_UCS2_vswscnaf</td>
</tr>
<tr>
<td>_C_UCS2_iswprint</td>
<td>_C_UCS2_vswscnaf</td>
</tr>
<tr>
<td>_C_UCS2_iswupper</td>
<td>_C_UCS2_vswscnaf</td>
</tr>
</tbody>
</table>

When you use the LOCALETYPE(*LOCALEUCS2) option with either the C or C++ compiler, the default UCS-2 locale is loaded when the program starts. When you use any of the Unicode functions in the preceding table from a different language, a call to _UCS2_setlocale(LC_ALL, ") should be added when the application starts to ensure that the default UCS2 locale is loaded.

The following table shows the functions added for CCSID neutral and UTF-8 support. The functions have a prefix of _C_NEU_DM_ (for data management I/O functions), _C_NEU_IFS_ or _C_UTF_IFS_ (for IFS I/O functions), or _C_NEU_ or _C_UTF_ added to the standard function name. The Unicode function has the same parameters as the standard (non-Unicode) function.
Functions that operate on wide characters have UTF in the prefix. Functions that do not operate on wide characters have NEU in the prefix.
When you use the LOCALETYPE(*LOCALEUTF) option with either the C or C++ compiler, the default UTF locale is loaded at program startup time. If you use any of the Unicode functions in the preceding table from a different language, a call to `_C_UTF_setlocale(LC_ALL, "")` should be added when the application starts to ensure that the default UTF locale is loaded.

**Standard Files**

When using the UTF support, the default standard input and output files stdin, stdout, and stderr have some special processing done for them by the runtime. Since a program using UTF support contains data in UTF-8 and the standard files interact with the screen and spool files, there is a potential mismatch in data. The screen and spool file functions are provided by the operating system and thus expect EBCDIC. For stdout and stderr, the runtime will automatically convert UTF-8 data to EBCDIC. For stdin, the runtime will automatically convert the incoming EBCDIC to UTF-8 data.

**Considerations**

Because the default environment for IBM i is primarily an EBCDIC environment, you must be aware of the situations described in this topic when you use UTF support in an application.

If a program or service program has some modules compiled with the UTF support and some modules compiled without the UTF support, care must be taken to ensure that unexpected mismatches do not occur. The wide characters and wide character strings are two bytes in size for a non-UTF module and
four bytes in size for a UTF module, so sharing wide characters between the modules may not work correctly. The narrow (non-wide) characters and character strings are in job CCSID for a non-UTF module and in CCSID 1208 for a UTF module, so sharing narrow characters between the modules may not work correctly either.

Whenever a setlocale() is performed to set the locale to a different CCSID, the standard output files should be flushed to avoid buffering problems with character data containing multiple CCSIDs. Since stdout is line buffered by default, if each output line ends in a newline character, the problem will not occur. However, if this is not done, the output may not be shown as intended. The following example illustrates the problem.

```c
#include <stdio>
#include <locale.h>

int main() {
  /* This string is in CCSID 1208 */
  printf("Hello World");

  /* Change locale to a CCSID 37 locale */
  setlocale(LC_ALL, "/QSYS.LIB/EN_US.LOCALE");
  #pragma convert(37)

  /* This string is in CCSID 37 */
  printf("Hello World\n");

  return 0;
}
```

In this case, the first printf() causes the CCSID 1208 string "Hello World" to be copied to the stdout buffer. Before the setlocale() is done, stdout should be flushed to copy that string to the screen. The second printf() causes the CCSID 37 string "Hello World\n" to be copied to the stdout buffer. Because of the trailing newline character, the buffer is flushed at that point and the whole buffer is copied to the screen. Because the CCSID of the current locale is 37 and the screen can handle CCSID 37 without problems, the whole buffer is copied without conversion. The CCSID 1208 characters are displayed as unreadable characters. If a flush had been done, the CCSID 1208 characters would have been converted to CCSID 37 and would have been displayed correctly.

Nearly all of the runtime functions have been modified to support UTF, but there are a handful of them that have not. Functions and structures that deal with exception handling, such as the _GetExcData() function, the _EXCP_MSGID variable, and the exception handler structure _INTRPT_Hndlr_Parms_T are provided by the operating system, not the runtime. They are strictly EBCDIC. The getenv() and putenv() functions handle only EBCDIC. The QXXCHGDA() and QXXRTVDA() functions handle only EBCDIC. The argv and envp parameters are also EBCDIC only.

Some of the record I/O functions (that is, functions beginning with _R) do not completely support UTF. The functions that do not support UTF are _Rformat(), _Rcommit(), _Racquire(), _Rrelease(), _Rpgmdev(), _Rindara(), and _Rdevatr(). They are available when compiling with the UTF option, but they accept and generate only EBCDIC. In addition, any character data within the structures returned by the _R functions will be in EBCDIC rather than UTF.

Other operating system functions have not been modified to support UTF. For example, the integrated file system functions, such as open(), still accept the job CCSID. Other operating system APIs still accept the job CCSID. For UTF applications, the characters and character strings provided to these functions need to be converted to the job CCSID using QTQCVRVT, iconv(), #pragma convert, or some other method.

**Default File CCSID**

When the fopen() function is used to open files, the default CCSID of the file is different depending on whether or not UTF support is used. If UTF support is not used (that is, if LOCALETYPE(*CLD), LOCALETYPE(*LOCAL), or LOCALETYPE(*LOCALEUCS2) are specified on the compilation command), the file CCSID defaults to the current job CCSID. Usually this works well because the job CCSID is set correctly and the current locale is set to match the job CCSID.
With UTF support, the job CCSID cannot be set to UTF-8 because of system limitations. When
LOCALETYPE(*LOCALEUTF) is specified, the file CCSID defaults to the CCSID of the current locale. If the
default locale is being used, the CCSID defaults to UTF-8 (CCSID 1208). If this default is not desired, the
ccsid or o_ccsid keyword can be specified in the second parameter of the fopen() call. However,
database files are an exception, because DB2® for IBM i does not completely support UTF-8. When
SYSIFCOPT(*NOIFSIO) is specified, and the CCSID of the current locale is 1208, the CCSID of the file
defaults to CCSID 65535 (no conversion) rather than CCSID 1208. This allows CCSID 1208 to be used
with database files. For more information about file CCSIDs, see “fopen() — Open Files” on page 134.

Newline Character

When the UTF support is not used, the hexadecimal value generated by the compiler for the character \n
and used by the run time has two different values. The hexadecimal value 0x15 is used if
SYSIFCOPT(*NOIFSIO) is specified on the compilation command. The hexadecimal value 0x25 is used if
SYSIFCOPT(*IFSIO) or SYSIFCOPT(*IFS64IO) is specified on the compilation command. When the UTF
support is used, the newline character in UTF-8 will be hexadecimal 0x0a regardless of what SYSIFCOPT
value is used.

Conversion Errors

Some runtime functions perform a CCSID conversion from UTF-8 to an EBCDIC CCSID when required to
interface with an operating system function that does not support UTF-8. When a conversion error occurs
in these cases, a C2M1217 message is generated to the job log with the conversion information.

Heap Memory

Heap Memory Overview

Heap memory is a common pool of free memory used for dynamic memory allocations within an
application.

Heap Memory Manager

A heap memory manager is responsible for the management of heap memory.
The heap memory manager performs the following fundamental memory operations:
• Allocation - performed by malloc and calloc
• Deallocation - performed by free
• Reallocation - performed by realloc

The ILE runtime provides three different heap memory managers:
• Default memory manager - a general-purpose memory manager
• Quick Pool memory manager - a pool memory manager
• Debug memory manager - a memory manager for debugging application heap problems

In addition, each of the memory managers has two different versions - a single-level store version and a
teraspace version. In most cases, the two versions behave similarly except that the single-level store
version returns pointers into single-level store storage and the teraspace version returns pointers into
teraspace storage. The single-level store versions are limited to slightly less than 16 MB for a single
allocation. The single-level store versions are also limited to slightly less than 4 GB for the maximum
amount of allocated heap storage. The teraspace versions are not subject to these limitations. For
additional information about single-level store and teraspace storage, please refer to the ILE Concepts
manual.

The default memory manager is the preferred choice for most applications and is the memory manager
enabled by default. The other memory managers have unique characteristics that can be beneficial in
specific circumstances. Environment variables can be used to indicate which heap manager to use as well as to provide heap manager options. In some cases, functions are also available to indicate which heap manager to use.

**Note:** The heap manager environment variables are checked only once per activation group, at the first heap function which is called within the activation group. To ensure that the environment variables are used, set up the environment variables before the creation of the activation group.

**Default Memory Manager**

The default memory manager is a general-purpose memory manager which attempts to balance performance and memory requirements. It provides adequate performance for most applications while attempting to minimize the amount of additional memory needed.

The memory manager maintains the free space in the heap as nodes in a Cartesian binary search tree. This data structure imposes no limitation on the number of block sizes supported by the tree, allowing a wide range of potential block sizes.

**Allocation**

A small amount of additional memory is required for each allocation request. This additional memory is due to the need for a header on each allocation and the need for alignment of each block of memory. The size of the header on each allocation is 16 bytes. Each block must be aligned on a 16 byte boundary, thus the total amount of memory required for an allocation of size \(n\) is:

\[
\text{size} = \text{ROUND} (n+16, 16)
\]

For example, an allocation of size 37 would require a size of \(\text{ROUND}(37+16, 16)\), which is equal to 64 bytes.

A node of the tree that is greater than or equal to the size required is removed from the tree. If the block found is larger than the needed size, the block is divided into two blocks: one of the needed size, and the second a remainder. The second block is returned to the free tree for future allocation. The first block is returned to the caller.

If a block of sufficient size is not found in the free tree, the following processing occurs:

- The heap is expanded.
- A block the size of the acquired extension is added to the free tree.
- Allocation continues as previously described.

**Deallocation**

Memory blocks deallocated with the `free` operation are returned to the tree, at the root. Each node along the path to the insertion point for the new node is examined to see if it adjoins the node being inserted. If it does, the two nodes are merged and the newly merged node is relocated in the tree. If no adjoining block is found, the node is inserted at the appropriate place in the tree. Merging adjacent blocks is done to reduce heap fragmentation.

**Reallocation**

If the size of the reallocated block is larger than the original block, and the original block already has enough space to accommodate the new size (e.g. due to alignment requirements), the original block is returned without any data movement. If the size of the reallocated block is larger than the original block, the following processing occurs:

- A new block of the requested size is allocated.
- The data is moved from the original block to the new block.
- The original block is returned to the free tree with the `free` operation.
- The new block is returned to the caller.
If the size of the reallocated block is smaller than the original block, and the difference in size is small, the original block is returned. Otherwise, if the size of the reallocated block is smaller than the original block, the block is split and the remaining portion is returned to the free tree.

**Enabling the default memory manager**

The default memory manager is enabled by default and can be configured by setting the following environment variables:

```
QIBM_MALLOC_TYPE=DEFAULT
QIBM_MALLOC_DEFAULT_OPTIONS=options
```

To specify user-specified configuration options for the default memory manager, set `QIBM_MALLOC_DEFAULT_OPTIONS=options`, where `options` is a blank delimited list of one or more configuration options.

If the `QIBM_MALLOC_TYPE=DEFAULT` environment variable is specified and the `_C_Quickpool_Init()` function is called, the environment variable settings take precedence over the `_C_Quickpool_Init()` function and the `_C_Quickpool_Init()` function returns a -1 value indicating that an alternate heap manager has already been enabled.

**Configuration Options**

The following configuration options are available:

**MALLOC_INIT:N**

This option can be used to specify that each byte of allocated memory is initialized to the given value. The value `N` represents an integer in the range of 0 to 255.

This option is not enabled by default.

**FREE_INIT:N**

This option can be used to specify that each byte of freed memory is initialized to the given value. The value `N` represents an integer in the range of 0 to 255.

This option is not enabled by default.

Any number of options can be specified and they can be specified in any order. Blanks are the only valid delimiter for separating configuration options. Each configuration option can only be specified once. If a configuration option is specified more than once, only the final instance applies. If a configuration option is specified with an invalid value, the configuration option is ignored.

**Examples**

```
ADDENVVAR ENVVAR(QIBM_MALLOC_DEFAULT_OPTIONS) LEVEL(*JOB) REPLACE(*YES) VALUE('')
ADDENVVAR ENVVAR(QIBM_MALLOC_DEFAULT_OPTIONS) LEVEL(*JOB) REPLACE(*YES) VALUE('MALLOC_INIT:255 FREE_INIT:0')
```

The first example represents the default configuration values. The second example illustrates all options being specified.

**Related functions**

There are no functions available to enable or specify configuration options for the default memory manager. The environment variable support must be used.

**Related Information**

- “calloc() — Reserve and Initialize Storage” on page 80
- “free() — Release Storage Blocks” on page 151
- “malloc() — Reserve Storage Block” on page 217
- “realloc() — Change Reserved Storage Block Size” on page 291
- “_C_TS_malloc_debug() — Determine amount of teraspace memory used (with optional dumps and verification)” on page 103
Quick Pool Memory Manager

The Quick Pool memory manager breaks memory up into a series of pools. It is intended to improve heap performance for applications that issue large numbers of small allocation requests. When the Quick Pool memory manager is enabled, allocation requests that fall within a given range of block sizes are assigned a cell within a pool. These requests can be handled more quickly than requests outside of this range. Allocation requests outside this range are handled in the same manner as the default memory manager.

A pool consists of a block of memory (called an extent) that is subdivided into a predetermined number of smaller blocks (called cells) of uniform size. Each cell can be allocated as a block of memory. Each pool is identified using a pool number. The first pool is pool 1, the second pool is pool 2, the third pool is pool 3, and so on. The first pool is the smallest and each succeeding pool is equal to or larger in size than the preceding pool.

The number of pools and cell sizes for each of the pools is determined at the time the Quick Pool memory manager is initialized.

Allocation

A cell is allocated from one of the pools when an allocation request falls within the range of cell sizes defined by the pools. Each allocation request is serviced from the smallest possible pool to conserve space.

When the first request comes in for a pool, an extent is allocated for the pool and the request is satisfied from that extent. Later requests for that pool are also satisfied by the extent until the extent is exhausted. When an extent is exhausted, a new extent is allocated for the pool.

Deallocation

Memory blocks (cells) deallocated with the free operation are added to a free queue associated with the pool that contains the cell. Each pool has a free queue that contains cells that have been freed and have not yet been reallocated. Additional allocation requests from that pool use cells from the free queue.

Reallocation

If the size of the reallocated block falls within the same pool as the original block, the original block is returned without any data movement. Otherwise, a new block of the requested size is allocated, the data is moved from the original block to the new block, the original block is returned to the free queue with the free operation, and the new block is returned to the caller.

Enabling the Quick Pool Memory Manager

The Quick Pool memory manager is not enabled by default. It is enabled and configured either by calling the _C_Quickpool_Init() and _C_Quickpool_Debug() functions or by setting the following environment variables:

```
QIBM_MALLOC_TYPE=QUICKPOOL
QIBM_MALLOC_QUICKPOOL_OPTIONS=options
```

To enable the Quick Pool memory manager with the default settings, the QIBM_MALLOC_QUICKPOOL_OPTIONS environment variable does not need to be specified, only QIBM_MALLOC_TYPE=QUICKPOOL needs to be specified. To enable the Quick Pool memory manager with user-specified configuration options, set QIBM_MALLOC_QUICKPOOL_OPTIONS=options, where options is a blank delimited list of one or more configuration options.

If the QIBM_MALLOC_TYPE=QUICKPOOL environment variable is specified and the _C_Quickpool_Init() function is called, the environment variable settings take precedence over the _C_Quickpool_Init() function and the _C_Quickpool_Init() function returns a -1 value indicating that an alternate heap manager has already been enabled.

If the QIBM_MALLOC_TYPE=QUICKPOOL environment variable is specified and the _C_Quickpool_Debug() function is called to change the Quick Pool memory manager characteristics,
the settings specified on the parameter to the _C_Quickpool_Debug() function override the environment variable settings.

**Configuration Options**

The following configuration options are available:

**POOLS:**($C_1$ $E_1$)($C_2$ $E_2$)...($C_n$ $E_n$)

This option can be used to specify the number of pools to be used, along with the cell size and extent cell count for each pool. The subscript value $n$ indicates the number of pools. The minimum valid value of $n$ is 1. The maximum valid value of $n$ is 64.

The value $C_1$ indicates the cell size for pool 1, $C_2$ indicates the cell size for pool 2, $C_n$ indicates the cell size for pool $n$, and so on. This value must be a multiple of 16 bytes. If a value is specified that is not a multiple of 16 bytes, the cell size is rounded up to the next larger multiple of 16 bytes. The minimum valid value is 16 and the maximum valid value is 4096.

The value $E_1$ indicates the extent cell count for pool 1, $E_2$ indicates the extent cell count for pool 2, $E_n$ indicates the extent cell count for pool $n$, and so on. The value specifies the number of cells in a single extent. The value can be any non-negative number, but the total size of the extent might be limited due to architecture constraints. A value of zero indicates that the implementation can choose a large value.

The default value for this option is "POOLS:(16 4096) (32 4096) (64 1024) (128 1024) (256 512) (512 512) (1024 256) (2048 256) (4096 256)". The defaults represent 9 pools with cells of sizes 16, 32, 64, 128, 256, 512, 1024, 2048, and 4096 bytes. The number of cells in each extent is 4096, 4096, 1024, 1024, 512, 512, 256, 256, and 256.

**MALLOC_INIT:**$N$

This option can be used to specify that each byte of allocated memory is initialized to the given value. The value $N$ represents an integer in the range of 0 to 255.

This option is not enabled by default.

**FREE_INIT:**$N$

This option can be used to specify that each byte of freed memory is initialized to the given value. The value $N$ represents an integer in the range of 0 to 255.

This option is not enabled by default.

**COLLECT_STATS**

This option can be used to specify that the Quick Pool memory manager collect statistics and report those statistics upon termination of the application. The Quick Pool memory manager collects statistics by calling _atexit(_C_Quickpool_Report) when this option is specified. Details about the information contained within that report are documented in the description for _C_Quickpool_Report().

This option is not enabled by default.

Any number of options can be specified and they can be specified in any order. Blanks are the only valid delimiters for separating configuration options. Each configuration option should only be specified once. If a configuration option is specified more than once, only the final instance applies. If a configuration option is specified with an invalid value, the configuration option is ignored.

**Examples**

```
ADDENVVAR ENVVAR(QIBM_MALLOC_QUICKPOOL_OPTIONS) LEVEL(*JOB) REPLACE(*YES)
VALUE('POOLS:(16 4096) (32 4096) (64 1024) (128 1024) (256 512) (512 512) (1024 256) (2048 256) (4096 256)')

ADDENVVAR ENVVAR(QIBM_MALLOC_QUICKPOOL_OPTIONS) LEVEL(*JOB) REPLACE(*YES)
VALUE('POOLS:(16 1000) MALLOC_INIT:255 FREE_INIT:0 COLLECT_STATS')
```

The first example represents the default configuration values. The second example illustrates all options being specified.
Related Functions

The _C_Quickpool_Init() function allows enablement of the Quick Pool memory manager. The _C_Quickpool_Init() function also specifies the number of pools to be used, the cell size, and the extent cell count for each pool.

The _C_Quickpool_Debug() function allows enablement of the other configuration options.

The _C_Quickpool_Report() function is used to report memory statistics.

Note:

1. The default configuration for the Quick Pool memory manager provides a performance improvement for many applications that issue large numbers of small allocation requests. However, it might be possible to achieve additional gains by modifying the default configuration. Before modifying the default configuration, become familiar with the memory requirements and usage of the application. The Quick Pool memory manager can be enabled with the COLLECT_STATS option to fine-tune the Quick Pool memory manager configuration.

2. Because of variations in memory requirements and usage, some applications might not benefit from the memory allocation scheme used by the Quick Pool memory manager. Therefore, it is not advisable to enable the Quick Pool memory manager for system-wide use. For optimal performance, enable and configure the Quick Pool memory manager on a per-application basis.

3. It is allowable to create more than one pool with the same size cells. This can be useful for multi-threaded applications which perform many similar sized allocations. When there is no contention, the first pool of the requested size is used. When contention occurs on the first pool, the Quick Pool memory manager allocates cells from any other equal sized pools to minimize contention.

Related Information

- “_C_Quickpool_Init() — Initialize Quick Pool Memory Manager” on page 93
- “_C_Quickpool_Debug() — Modify Quick Pool Memory Manager Characteristics” on page 91
- “_C_Quickpool_Report() — Generate Quick Pool Memory Manager Report” on page 95

Debug Memory Manager

The debug memory manager is used primarily to find incorrect heap usage by an application. It is not optimized for performance and might negatively affect the performance of the application. However, it is valuable in determination of incorrect heap usage.

Memory management errors are sometimes caused by writing past the end of an allocated buffer. Symptoms do not arise until much later when the memory that was overwritten (typically belonging to another allocation) is referenced and no longer contains the expected data.

The debug memory manager allows detection of memory overwrites, memory over reads, duplicate frees, and reuse of freed memory. Memory problems detected by the debug memory manager result in one of two behaviors:

- If the problem is detected at the point that the incorrect usage occurs, an MCH exception message (typically an MCH0601, MCH3402, or MCH6801) is generated. In this case, the error message typically stops the application.
- If the problem is not detected until later, after the incorrect usage has already occurred, a C2M1212 message is generated. In this case, the message does not typically stop the application.

The debug memory manager detects memory overwrites and memory over reads in two ways:

- First, it uses restricted access memory pages. A memory page with restricted access is placed before and after each allocation. Each memory block is aligned on a 16 byte boundary and placed as close to the end of a page as possible. Since memory protection is only allowed on a page boundary, this alignment allows the best detection of memory overwrites and memory over reads. Any read or write from one of the restricted access memory pages immediately results in an MCH exception.
- Second, it uses padding bytes before and after each allocation. A few bytes immediately before each allocation are initialized at allocation time to a preset byte pattern. Any padding bytes following the
allocation required to round the allocation size to a multiple of 16 bytes are initialized at allocation time
to a preset byte pattern. When the allocation is freed, all the padding bytes are verified to ensure that
they still contain the expected preset byte pattern. If any of the padding bytes have been modified, the
debug memory manager generates a C2M1212 message with reason code X’80000000’, indicating this
fact.

Allocation

A large amount of extra memory is required for each allocation request. The extra memory is due to the
following:

- A memory page before the allocation (single-level store version only)
- A memory page after the allocation
- A header on each allocation
- Alignment of each block of memory on a 16 byte boundary

The size of the header on each allocation is 16 bytes. Each block must be aligned on a 16 byte boundary.
The total amount of memory required for an allocation of size $n$ in the single-level store version is:

$$\text{size} = \text{ROUND}((\text{PAGESIZE} * 2) + n + 16, \text{PAGESIZE})$$

For example, an allocation of size 37 with a page size of 4096 bytes requires a size of \text{ROUND}(8192 + 37
+ 16, 4096), which is equal to 12,288 bytes.

The total amount of memory required for an allocation of size $n$ in the teraspace version is:

$$\text{size} = \text{ROUND}(\text{PAGESIZE} + n + 16, \text{PAGESIZE})$$

For example, an allocation of size 37 with a page size of 4096 bytes requires a size of \text{ROUND}(4096 + 37
+ 16, 4096), which is equal to 8,192 bytes.

Deallocation

Memory blocks deallocated with the free operation are returned to the system. The page protection
attributes are set so that any further read or write access to that memory block generates an MCH
exception.

Reallocation

In all cases, the following processing occurs:

- A new block of the requested size is allocated.
- The data is moved from the original block to the new block.
- The original block is returned with the free operation.
- The new block is returned to the caller.

Enabling the debug memory manager

The debug memory manager is not enabled by default, but is enabled and configured by setting the
following environment variables:

```
QIBM_MALLOC_TYPE=DEBUG
QIBM_MALLOC_DEBUG_OPTIONS=options
```

To enable the debug memory manager with the default settings, QIBM_MALLOC_TYPE=DEBUG needs to
be specified. To enable the debug memory manager with user-specified configuration options, set
QIBM_MALLOC_DEBUG_OPTIONS=options where options is a blank delimited list of one or more
configuration options.

If the QIBM_MALLOC_TYPE=DEBUG environment variable is specified and the _C_Quickpool_Init() function is called, the environment variable settings take precedence over the _C_Quickpool_Init() function and the _C_Quickpool_Init() function returns a -1 value indicating that an alternate heap manager has been enabled.
**Configuration Options**

The following configuration options are available:

**MALLOC_INIT:N**

This option can be used to specify that each byte of allocated memory is initialized to the given value. The value \( N \) represents an integer in the range of 0 to 255.

This option is not enabled by default.

**FREE_INIT:N**

This option can be used to specify that each byte of freed memory is initialized to the given value. The value \( N \) represents an integer in the range of 0 to 255.

This option is not enabled by default.

Any number of options can be specified and they can be specified in any order. Blanks are the only valid delimiters for separating configuration options. Each configuration option should only be specified once. If a configuration option is specified more than once, only the final instance applies. If a configuration option is specified with an invalid value, the configuration option is ignored.

**Examples**

```
ADDENVVAR ENVVAR(QIBM_MALLOC_DEBUG_OPTIONS) LEVEL(*JOB) REPLACE(*YES) VALUE('')
ADDENVVAR ENVVAR(QIBM_MALLOC_DEBUG_OPTIONS) LEVEL(*JOB) REPLACE(*YES) VALUE('MALLOC_INIT:255 FREE_INIT:0')
```

The first example represents the default configuration values. The second example illustrates all options being specified.

**Related Functions**

There are no functions available to enable or specify configuration options for the debug memory manager. Use the environment variable support to enable or specify configuration options.

**Note:**

1. Use the debug memory manager to debug single applications or small groups of applications at the same time.

   The debug memory manager is not appropriate for full-time, constant, or system-wide use. Although it is designed for minimal performance impact upon the application being debugged, significant negative impact on overall system throughput can result if it is used on a system-wide basis. It might cause significant system problems, such as excessive use of the system auxiliary storage pool (ASP).

2. The debug memory manager consumes significantly more memory than the default memory manager. As a result, the debug memory manager might not be appropriate for use in some debugging situations.

   Because the allocations require two memory pages or more of extra memory per allocation, applications that issue many small allocation requests see their memory usage increase dramatically. These programs might encounter new failures as memory allocation requests are denied due to a lack of memory. These failures are not necessarily errors in the application being debugged and they are not errors in the debug memory manager.

   Single-level store versions are limited to slightly less than 4 GB for the maximum amount of allocated heap storage. The debug memory manager allocates a minimum of three pages per allocation, which allows for less than 350,000 outstanding heap allocations (with a page size of 4096 bytes).

3. The single-level store version of the debug memory manager does each allocation in a separate 16 MB segment which can cause the system to use temporary addresses more rapidly.

**Related Information**

- “calloc() — Reserve and Initialize Storage” on page 80
- “free() — Release Storage Blocks” on page 151
Environment Variables

The following tables describe the environment variables which can be used to enable and configure heap memory managers.

The following environment variable can be used to indicate which memory manager should be used:

<table>
<thead>
<tr>
<th>Environment Variable</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>QIBM_MALLOC_TYPE</td>
<td>DEFAULT</td>
<td>Indicates that the default memory manager is to be used.</td>
</tr>
<tr>
<td>QUICKPOOL</td>
<td></td>
<td>Indicates that the Quick Pool memory manager is to be used.</td>
</tr>
<tr>
<td>DEBUG</td>
<td></td>
<td>Indicates that the debug memory manager is to be used.</td>
</tr>
</tbody>
</table>

If the QIBM_MALLOC_TYPE environment variable is not set, or if it has a value different than one of the above values, the default memory manager is used and all of the following environment variables are ignored.

If QIBM_MALLOC_TYPE is set to DEFAULT, the following environment variable can be used to indicate default memory manager options. Otherwise, the environment variable is ignored.

<table>
<thead>
<tr>
<th>Environment Variable</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>QIBM_MALLOC_DEFAULT_OPTIONS</td>
<td>MALLOC_INIT:N</td>
<td>Each byte of allocated memory is initialized to this value.</td>
</tr>
<tr>
<td></td>
<td>FREE_INIT:N</td>
<td>Each byte of freed memory is initialized to this value.</td>
</tr>
</tbody>
</table>

By default, neither allocated memory nor freed memory is initialized.

If QIBM_MALLOC_TYPE is set to QUICKPOOL, the following environment variable can be used to indicate Quick Pool memory manager options. Otherwise, the environment variable is ignored.
Table 44. Quick Pool Memory Manager Options

<table>
<thead>
<tr>
<th>Environment Variable</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>QIBM_MALLOC_QUICKPOOL_OPTIONS</td>
<td>POOLS:(C₁ E₁) (C₂ E₂) ... (Cₙ Eₙ)</td>
<td>Defines the cell sizes and extent cell counts for each pool. The number of (Cₙ Eₙ) pairs indicate the number of the pools.</td>
</tr>
<tr>
<td>MALLOC_INIT:N</td>
<td>Each byte of allocated memory is initialized to this value.</td>
<td></td>
</tr>
<tr>
<td>FREE_INIT:N</td>
<td>Each byte of freed memory is initialized to this value.</td>
<td></td>
</tr>
<tr>
<td>COLLECT_STATS</td>
<td>Indicates to collect statistics and generate a report when the application ends.</td>
<td></td>
</tr>
</tbody>
</table>

By default, neither allocated memory nor freed memory is initialized. The default behavior is not to collect statistics. If the cell sizes and extent cell counts are not specified or are specified incorrectly, the default configuration values are used, as described earlier in this section.

If QIBM_MALLOC_TYPE is set to DEBUG, the following environment variable can be used to indicate debug memory manager options. Otherwise, the environment variable is ignored.

Table 45. Debug Memory Manager Options

<table>
<thead>
<tr>
<th>Environment Variable</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>QIBM_MALLOC_DEBUG_OPTIONS</td>
<td>MALLOC_INIT: N</td>
<td>Each byte of allocated memory is initialized to this value.</td>
</tr>
<tr>
<td></td>
<td>FREE_INIT:N</td>
<td>Each byte of freed memory is initialized to this value.</td>
</tr>
</tbody>
</table>

By default, neither allocated memory nor freed memory is initialized.

Diagnosing C2M1211/C2M1212 Message Problems

This section provides information that might help to diagnose problems which are indicated with a C2M1211 message or C2M1212 message in the job log.

C2M1211 Message

A C2M1211 message indicates that a teraspace version of the heap memory manager has detected that the heap control structure has been corrupted.

The C2M1211 message can be caused by many things. The most common causes include:

- Freeing a space twice.
- Writing outside the bounds of allocated storage.
- Writing to storage that has been freed.
The CM1211 message often indicates an application heap problem. Unfortunately, these problems are often difficult to track down. The best approach to debug this type of problem is to enable the debug memory manager.

**C2M1212 Message**

A C2M1212 message indicates some type of memory problem which can lead to memory corruption and other issues. The memory corruption could occur within application code or operating system code. The message is only a diagnostic message, but can be an indicator of a real problem. The C2M1212 message might or might not be the source of other problems. Clean up the memory problem if possible.

When a C2M1212 message is generated, the hexadecimal value of the pointer passed to the `free()` function is included as part of the message description. This hexadecimal value can provide clues as to the origin of the problem. The `malloc()` function returns only pointers that end in hexadecimal 0. Any pointer that does not end in hexadecimal 0 was either never set to point to storage reserved by the `malloc()` function or was modified since it was set to point to storage reserved by the `malloc()` function. If the pointer ends in hexadecimal 0, then the cause of the C2M1212 message is uncertain, and the program code that calls `free()` should be examined.

In most cases, a C2M1212 message from a single-level store heap memory manager is preceded by an MCH6902 message. The MCH6902 message has an error code indicating what the problem is. The most common error code is 2, which indicates that memory is being freed which is not currently allocated. This error code could mean one of the following:

- Memory is being freed which has not been allocated.
- Memory is being freed for a second time.

In some cases, a memory leak can cause the single-level store heap to become fragmented to the point that the heap control segment is full and deallocates fail. This problem is indicated by an MCH6906 message. In this case, the only solution is to debug the application and fix the memory leak.

Stack tracebacks (See “Stack Tracebacks” on page 588) can be used to find the code which is causing the problem. Once the code has been found, the difficult part is to determine what the problem is with the pointer to the heap storage. There are several potential causes:

1. The pointer was never initialized and contains an unexpected value. The C2M1212 message dumps the hex value of the pointer.
2. The pointer was not obtained from `malloc()`. Perhaps the pointer is a pointer to an automatic (local) variable or a static (global) variable and not a pointer to heap storage from `malloc()`.
3. The pointer was modified after it was returned from `malloc()`. For example, if the pointer returned from `malloc()` was incremented by some amount and then passed to `free()`, it would be invalid and a C2M1212 message is issued.
4. The pointer is being passed a second time to `free()`. Once `free()` has been called with the pointer, the space pointed to by that pointer is deallocated and if `free()` is called again, a C2M1212 message is issued.
5. The heap structure maintained by the heap manager to track heap allocations has been corrupted. In this case, the pointer is a valid pointer but the heap manager cannot determine that and a C2M1212 message results. When the heap structure is corrupted, there is typically at least one C2M1211 message in the job log to indicate that heap corruption has occurred.
6. If the debug memory manager is in use and the reason code on the C2M1212 message is X'8000000000', padding bytes were overwritten for the given allocation. Refer to “Debug Memory Manager” on page 583 for more information.

**Stack Tracebacks**

**Enablement for single-level store heap memory managers**

When a C2M1211 or C2M1212 message is generated from a single-level store heap function, the code checks for a *DTAARA named QGPL/QC2M1211 or QGPL/QC2M1212. If the data area exists, the program stack is dumped. If the data area does not exist, no dump is performed.
Enablement for teraspace heap memory managers

When a C2M1211 message or C2M1212 message is generated from a teraspace heap function, the code checks for a *DTAARA named QGPL/QC2M1211 or QGPL/QC2M1212. If the data area exists and contains at least 50 characters of data, a 50 character string is retrieved from the data area. If the string within the data area matches one of the following strings, special behavior is triggered.

_C_TS_dump_stack
_C_TS_dump_stack_vfy_heap
_C_TS_dump_stack_vfy_heap_wabort
_C_TS_dump_stack_vfy_heap_wsleep

If the data area does not exist, no dump or heap verification is performed.

The behavior defaults to the _C_TS_dump_stack behavior in the following cases:

- The data area exists but does not contain character data.
- The data area is less than 50 characters in length.
- The data area does not contain any of the listed strings.

The strings in the data area have the following meaning:

_C_TS_dump_stack
The default behavior of dumping the stack is to be performed. No heap verification is done.

_C_TS_dump_stack_vfy_heap
After the stack is dumped, the _C_TS_malloc_debug() function is called to verify the heap control structures. If any corruption is detected within the heap control structures, the heap errors and all heap control information are dumped. Any heap information which is dumped is contained within the same file as the stack dump. If no heap corruption is detected, no heap information is dumped.

After the verification is performed, control returns to the original program generating the C2M1211 or C2M1212 message and execution continues.

_C_TS_dump_stack_vfy_heap_wabort
_C_TS_dump_stack_vfy_heap_wabort has the same verification behavior as _C_TS_dump_stack_vfy_heap.

Instead of returning control to the original program if heap corruption is detected, the abort() function is called to halt execution.

_C_TS_dump_stack_vfy_heap_wsleep
_C_TS_dump_stack_vfy_heap_wsleep has the same verification behavior as _C_TS_dump_stack_vfy_heap.

Instead of returning control to the original program if heap corruption is detected, the sleep() function is called to sleep indefinitely, and pause execution to allow debug of the application. The application needs to be ended manually.

Here is an example of how to create a data area to indicate to call _C_TS_malloc_debug to verify the heap whenever a C2M1212 message is generated:

CRDTAARA DTAARA(QGPL/QC2M1212) TYPE(*CHAR) LEN(50)
VALUE('_C_TS_dump_stack_vfy_heap')

Analysis

Once the data area is in place, a spool file named QPRINT is created with dump information for every C2M1211 message or C2M1212 message. The spool file is created for the user running the job which gets the message. For example, if the job getting the C2M1211 message or C2M1212 message is a server job or batch job running under userid ABC123 then the spool file is created in the output queue for userid ABC123. Once the spool files containing stack tracebacks are obtained, the data area can be removed, and the tracebacks analyzed.
The stack tracebacks can be used to find the code which is causing the problem. Here is an example stack traceback:

<table>
<thead>
<tr>
<th>PROGRAM NAME</th>
<th>PROGRAM LIB</th>
<th>MODULE NAME</th>
<th>MODULE LIB</th>
<th>INST#</th>
<th>PROCEDURE</th>
<th>STATEMENT#</th>
</tr>
</thead>
<tbody>
<tr>
<td>QC2UTIL1</td>
<td>QSYS</td>
<td>QC2ALLOC</td>
<td>QBUILDSS1</td>
<td>000000</td>
<td>dump_stack__Fv</td>
<td>0000001019</td>
</tr>
<tr>
<td>QC2UTIL1</td>
<td>QSYS</td>
<td>QC2ALLOC</td>
<td>QBUILDSS1</td>
<td>000000</td>
<td>free</td>
<td>0000001128</td>
</tr>
<tr>
<td>QYPPRT370</td>
<td>QSYS</td>
<td>DLSCNTDF37</td>
<td>QBUILDSS1</td>
<td>000000</td>
<td>__dl__FPv</td>
<td>0000000007</td>
</tr>
<tr>
<td>FSOSA</td>
<td>ABCSYS</td>
<td>OSAACTS</td>
<td>FSTESTOSA</td>
<td>000000</td>
<td>FS_FinalizeDoc</td>
<td>0000000110</td>
</tr>
<tr>
<td>ABCKRNK</td>
<td>ABCSYS</td>
<td>A2PDFUTILS</td>
<td>ABMOD_8</td>
<td>000000</td>
<td>PRT_EndDoc_Adb</td>
<td>0000000625</td>
</tr>
<tr>
<td>ABCKRNK</td>
<td>ABCSYS</td>
<td>A2PDFUTILS</td>
<td>ABMOD_8</td>
<td>000000</td>
<td>PRT_EndDoc</td>
<td>0000000003</td>
</tr>
<tr>
<td>ABCKRNK</td>
<td>ABCSYS</td>
<td>A2ENGINE</td>
<td>ABMOD_8</td>
<td>000000</td>
<td>ABCReport_Start</td>
<td>0000000087</td>
</tr>
<tr>
<td>ABCKRNK</td>
<td>ABCSYS</td>
<td>A2ENTRYPNT</td>
<td>ABMOD_8</td>
<td>000000</td>
<td>ABCReport_Run</td>
<td>0000000056</td>
</tr>
<tr>
<td>PRINTABC</td>
<td>ABCSYS</td>
<td>RUNBATCH</td>
<td>ABMOD_6</td>
<td>000000</td>
<td>main</td>
<td>0000000040</td>
</tr>
<tr>
<td>PRIMTBABC</td>
<td>ABCSYS</td>
<td>RUNBATCH</td>
<td>ABMOD_6</td>
<td>000000</td>
<td>_C_pep</td>
<td>0000422</td>
</tr>
<tr>
<td>QCMD</td>
<td>QSYS</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

The first line is the header line, which shows the program name, program library, module name, module library, instruction number, procedure name, and statement number.

The first line under the header is always a dump_stack procedure - this procedure is generating the C2M1211 message or C2M1212 message. The next line is the procedure which is calling the dump_stack procedure - that is almost always the free procedure, but it could be realloc or something else. The next line is the __dl__FPv procedure, which is the procedure which handles the C++ delete operator. For C++ code, this procedure is often in the stack - for C code, it is not.

The free and delete functions are library functions which are freeing memory on behalf of the caller. They are not important in determining the source of the memory problem.

The line after the __dl__FPv procedure is the one where things get interesting. In this example, the procedure is called FS_FinalizeDoc and this code contains the incorrect call to delete (it is deleting an object which has been previously deleted/freed). The owner of that application needs to look at the source code for that procedure at the given statement number to determine what is being deleted/freed. In some cases, this object is a local object of some type and it is easy to determine the problem. In other cases, the object can be passed to the procedure as a parameter and the caller of that procedure needs to be examined. In this case, the PRT_EndDoc_Adb procedure is the caller of FS_FinalizeDoc.

For this example, the problem is in code within the ABCSYS library.
Library Functions and Extensions

This topic summarizes all the standard C library functions and the ILE C library extensions.

Standard C Library Functions Table, By Name

This table briefly describes the C library functions, listed in alphabetical order. This table provides the include file name and the function prototype for each function.

<table>
<thead>
<tr>
<th>Function</th>
<th>System Include File</th>
<th>Function Prototype</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>abort</td>
<td>stdlib.h</td>
<td>void abort(void);</td>
<td>Stops a program abnormally.</td>
</tr>
<tr>
<td>abs</td>
<td>stdlib.h</td>
<td>int abs(int n);</td>
<td>Calculates the absolute value of an integer argument n.</td>
</tr>
<tr>
<td>acos</td>
<td>math.h</td>
<td>double acos(double x);</td>
<td>Calculates the arc cosine of x.</td>
</tr>
<tr>
<td>asctime</td>
<td>time.h</td>
<td>char *asctime(const struct tm *time);</td>
<td>Converts the time that is stored as a structure to a character string.</td>
</tr>
<tr>
<td>asctime_r</td>
<td>time.h</td>
<td>char *asctime_r (const struct tm *tm, char *buf);</td>
<td>Converts tm that is stored as a structure to a character string. (Restartable version of asctime.)</td>
</tr>
<tr>
<td>asin</td>
<td>math.h</td>
<td>double asin(double x);</td>
<td>Calculates the arc sine of x.</td>
</tr>
<tr>
<td>assert</td>
<td>assert.h</td>
<td>void assert(int expression);</td>
<td>Prints a diagnostic message and ends the program if the expression is false.</td>
</tr>
<tr>
<td>atan</td>
<td>math.h</td>
<td>double atan(double x);</td>
<td>Calculates the arc tangent of x.</td>
</tr>
<tr>
<td>atan2</td>
<td>math.h</td>
<td>double atan2(double y, double x);</td>
<td>Calculates the arc tangent of y/x.</td>
</tr>
<tr>
<td>atexit</td>
<td>stdlib.h</td>
<td>int atexit(void (*func)(void));</td>
<td>Registers a function to be called at normal termination.</td>
</tr>
<tr>
<td>atof</td>
<td>stdlib.h</td>
<td>double atof(const char *string);</td>
<td>Converts string to a double-precision floating-point value.</td>
</tr>
<tr>
<td>atoi</td>
<td>stdlib.h</td>
<td>int atoi(const char *string);</td>
<td>Converts string to an integer.</td>
</tr>
<tr>
<td>atol</td>
<td>stdlib.h</td>
<td>long int atol(const char *string);</td>
<td>Converts string to a long integer.</td>
</tr>
<tr>
<td>bsearch</td>
<td>stdlib.h</td>
<td>void *bsearch(const void *key, const void *base, size_t num, size_t size, int (*compare) (const void *element1, const void *element2));</td>
<td>Performs a binary search on an array of num elements, each of size bytes. The array must be sorted in ascending order by the function pointed to by compare.</td>
</tr>
<tr>
<td>btowc</td>
<td>stdio.h wchar.h</td>
<td>wint_t btowc(int c);</td>
<td>Determines whether c constitutes a valid multibyte character in the initial shift state.</td>
</tr>
<tr>
<td>Function</td>
<td>System Include File</td>
<td>Function Prototype</td>
<td>Description</td>
</tr>
<tr>
<td>----------</td>
<td>---------------------</td>
<td>--------------------</td>
<td>-------------</td>
</tr>
<tr>
<td>calloc</td>
<td>stdlib.h</td>
<td>void *calloc(size_t num, size_t size);</td>
<td>Reserves storage space for an array of num elements, each of size size, and initializes the values of all elements to 0.</td>
</tr>
<tr>
<td>catclose</td>
<td>nl_types.h</td>
<td>int catclose (nl_catd catd);</td>
<td>Closes a previously opened message catalog.</td>
</tr>
<tr>
<td>catgets</td>
<td>nl_types.h</td>
<td>char *catgets(nl_catd catd, int set_id, int msg_id, const char *s);</td>
<td>Retrieves a message from an open message catalog.</td>
</tr>
<tr>
<td>catopen</td>
<td>nl_types.h</td>
<td>nl_catd catopen (const char *name, int oflag);</td>
<td>Opens a message catalog, which must be done before a message can be retrieved.</td>
</tr>
<tr>
<td>ceil</td>
<td>math.h</td>
<td>double ceil(double x);</td>
<td>Calculates the double value representing the smallest integer that is greater than or equal to x.</td>
</tr>
<tr>
<td>clearerr</td>
<td>stdio.h</td>
<td>void clearerr(FILE *stream);</td>
<td>Resets the error indicators and the end-of-file indicator for stream.</td>
</tr>
<tr>
<td>clock</td>
<td>time.h</td>
<td>clock_t clock(void);</td>
<td>Returns the processor time that has elapsed since the job was started.</td>
</tr>
<tr>
<td>cos</td>
<td>math.h</td>
<td>double cos(double x);</td>
<td>Calculates the cosine of x.</td>
</tr>
<tr>
<td>cosh</td>
<td>math.h</td>
<td>double cosh(double x);</td>
<td>Calculates the hyperbolic cosine of x.</td>
</tr>
<tr>
<td>ctime</td>
<td>time.h</td>
<td>char *ctime(const time_t *time);</td>
<td>Converts time to a character string.</td>
</tr>
<tr>
<td>ctime64</td>
<td>time.h</td>
<td>char *ctime64(const time64_t *time);</td>
<td>Converts time to a character string.</td>
</tr>
<tr>
<td>ctime_r</td>
<td>time.h</td>
<td>char *ctime_r(const time_t *time, char *buf);</td>
<td>Converts time to a character string. (Restartable version of ctime.)</td>
</tr>
<tr>
<td>ctime64_r</td>
<td>time.h</td>
<td>char *ctime64_r(const time64_t *time, char *buf);</td>
<td>Converts time to a character string. (Restartable version of ctime64.)</td>
</tr>
<tr>
<td>difftime</td>
<td>time.h</td>
<td>double difftime(time_t time2, time_t time1);</td>
<td>Computes the difference between time2 and time1.</td>
</tr>
<tr>
<td>difftime64</td>
<td>time.h</td>
<td>double difftime64(time64_t time2, time64_t time1);</td>
<td>Computes the difference between time2 and time1.</td>
</tr>
<tr>
<td>div</td>
<td>stdlib.h</td>
<td>div_t div(int numerator, int denominator);</td>
<td>Calculates the quotient and remainder of the division of numerator by denominator.</td>
</tr>
<tr>
<td>erf</td>
<td>math.h</td>
<td>double erf(double x);</td>
<td>Calculates the error function of x.</td>
</tr>
<tr>
<td>erfc</td>
<td>math.h</td>
<td>double erfc(double x);</td>
<td>Calculates the error function for large values of x.</td>
</tr>
<tr>
<td>exit</td>
<td>stdlib.h</td>
<td>void exit(int status);</td>
<td>Ends a program normally.</td>
</tr>
<tr>
<td>Function</td>
<td>System Include File</td>
<td>Function Prototype</td>
<td>Description</td>
</tr>
<tr>
<td>----------</td>
<td>---------------------</td>
<td>--------------------</td>
<td>-------------</td>
</tr>
<tr>
<td>exp</td>
<td>math.h</td>
<td>double exp(double x);</td>
<td>Calculates the exponential function of a floating-point argument x.</td>
</tr>
<tr>
<td>fabs</td>
<td>math.h</td>
<td>double fabs(double x);</td>
<td>Calculates the absolute value of a floating-point argument x.</td>
</tr>
<tr>
<td>fclose</td>
<td>stdio.h</td>
<td>int fclose(FILE *stream);</td>
<td>Closes the specified stream.</td>
</tr>
<tr>
<td>fdopen⁵</td>
<td>stdio.h</td>
<td>FILE *fdopen(int handle, const char *type);</td>
<td>Associates an input or output stream with the file identified by handle.</td>
</tr>
<tr>
<td>feof</td>
<td>stdio.h</td>
<td>int feof(FILE *stream);</td>
<td>Tests whether the end-of-file flag is set for a given stream.</td>
</tr>
<tr>
<td>ferror</td>
<td>stdio.h</td>
<td>int ferror(FILE *stream);</td>
<td>Tests for an error indicator in reading from or writing to stream.</td>
</tr>
<tr>
<td>fflush¹</td>
<td>stdio.h</td>
<td>int fflush(FILE *stream);</td>
<td>Writes the contents of the buffer associated with the output stream.</td>
</tr>
<tr>
<td>fgetc¹</td>
<td>stdio.h</td>
<td>int fgetc(FILE *stream);</td>
<td>Reads a single unsigned character from the input stream.</td>
</tr>
<tr>
<td>fgetpos¹</td>
<td>stdio.h</td>
<td>int fgetpos(FILE *stream, fpos_t *pos);</td>
<td>Stores the current position of the file pointer associated with stream into the object pointed to by pos.</td>
</tr>
<tr>
<td>fgets¹</td>
<td>stdio.h</td>
<td>char *fgets(char *string, int n, FILE *stream);</td>
<td>Reads a string from the input stream.</td>
</tr>
<tr>
<td>fgetwc⁶</td>
<td>stdio.h wchar.h</td>
<td>wint_t fgetwc(FILE *stream);</td>
<td>Reads the next multibyte character from the input stream pointed to by stream.</td>
</tr>
<tr>
<td>fgetws⁶</td>
<td>stdio.h wchar.h</td>
<td>wchar_t *fgetws(wchar_t *wcs, int n, FILE *stream);</td>
<td>Reads wide characters from the stream into the array pointed to by wcs.</td>
</tr>
<tr>
<td>fileno⁵</td>
<td>stdio.h</td>
<td>int fileno(FILE *stream);</td>
<td>Determines the file handle currently associated with stream.</td>
</tr>
<tr>
<td>floor</td>
<td>math.h</td>
<td>double floor(double x);</td>
<td>Calculates the floating-point value representing the largest integer less than or equal to x.</td>
</tr>
<tr>
<td>fmod</td>
<td>math.h</td>
<td>double fmod(double x, double y);</td>
<td>Calculates the floating-point remainder of x/y.</td>
</tr>
<tr>
<td>fopen</td>
<td>stdio.h</td>
<td>FILE *fopen(const char *filename, const char *mode);</td>
<td>Opens the specified file.</td>
</tr>
<tr>
<td>fprintf</td>
<td>stdio.h</td>
<td>int fprintf(FILE *stream, const char *format-string, arg-list);</td>
<td>Formats and prints characters and values to the output stream.</td>
</tr>
<tr>
<td>fputc¹</td>
<td>stdio.h</td>
<td>int fputc(int c, FILE *stream);</td>
<td>Prints a character to the output stream.</td>
</tr>
<tr>
<td>Function</td>
<td>System Include File</td>
<td>Function Prototype</td>
<td>Description</td>
</tr>
<tr>
<td>--------------</td>
<td>---------------------</td>
<td>--------------------</td>
<td>-------------------------------------------------------</td>
</tr>
<tr>
<td>fputs¹</td>
<td>stdio.h</td>
<td>int fputs(const char *string, FILE *stream);</td>
<td>Copies a string to the output stream.</td>
</tr>
<tr>
<td>fputwc⁶</td>
<td>stdio.h wchar.h</td>
<td>wint_t fputwc(wchar_t wc, FILE *stream);</td>
<td>Converts the wide character wc to a multibyte character and writes it to the output stream pointed to by stream at the current position.</td>
</tr>
<tr>
<td>fputws⁶</td>
<td>stdio.h wchar.h</td>
<td>int fputws(const wchar_t *wcs, FILE *stream);</td>
<td>Converts the wide-character string wcs to a multibyte-character string and writes it to stream as a multibyte character string.</td>
</tr>
<tr>
<td>fread</td>
<td>stdio.h</td>
<td>size_t fread(void *buffer, size_t size, size_t count, FILE *stream);</td>
<td>Reads up to count items of size length from the input stream, and stores them in buffer.</td>
</tr>
<tr>
<td>free</td>
<td>stdlib.h</td>
<td>void free(void *ptr);</td>
<td>Frees a block of storage.</td>
</tr>
<tr>
<td>freopen</td>
<td>stdio.h</td>
<td>FILE *freopen(const char *filename, const char *mode, FILE *stream);</td>
<td>Closes stream, and reassigns it to the file specified.</td>
</tr>
<tr>
<td>frexp</td>
<td>math.h</td>
<td>double frexp(double x, int *expptr);</td>
<td>Separates a floating-point number into its mantissa and exponent.</td>
</tr>
<tr>
<td>fscanf</td>
<td>stdio.h</td>
<td>int fscanf(FILE *stream, const char *format-string, arg-list);</td>
<td>Reads data from stream into locations given by arg-list.</td>
</tr>
<tr>
<td>fseek¹</td>
<td>stdio.h</td>
<td>int fseek(FILE *stream, long int offset, int origin);</td>
<td>Changes the current file position associated with stream to a new location.</td>
</tr>
<tr>
<td>fsetpos¹</td>
<td>stdio.h</td>
<td>int fsetpos(FILE *stream, const fpos_t *pos);</td>
<td>Moves the current file position to a new location determined by pos.</td>
</tr>
<tr>
<td>ftell¹</td>
<td>stdio.h</td>
<td>long int ftell(FILE *stream);</td>
<td>Gets the current position of the file pointer.</td>
</tr>
<tr>
<td>fwide⁶</td>
<td>stdio.h wchar.h</td>
<td>int fwide(FILE *stream, int mode);</td>
<td>Determines the orientation of the stream pointed to by stream.</td>
</tr>
<tr>
<td>fwprintf⁶</td>
<td>stdio.h wchar.h</td>
<td>int fwprintf(FILE *stream, const wchar_t *format, arg-list);</td>
<td>Writes output to the stream pointed to by stream.</td>
</tr>
<tr>
<td>fwrite</td>
<td>stdio.h</td>
<td>size_t fwrite(const void *buffer, size_t size, size_t count, FILE *stream);</td>
<td>Writes up to count items of size length from buffer to stream.</td>
</tr>
<tr>
<td>fwscanf⁶</td>
<td>stdio.h wchar.h</td>
<td>int fwscanf(FILE *stream, const wchar_t *format, arg-list);</td>
<td>Reads input from the stream pointed to by stream.</td>
</tr>
<tr>
<td>gamma</td>
<td>math.h</td>
<td>double gamma(double x);</td>
<td>Computes the Gamma Function</td>
</tr>
<tr>
<td>getc¹</td>
<td>stdio.h</td>
<td>int getc(FILE *stream);</td>
<td>Reads a single character from the input stream.</td>
</tr>
<tr>
<td>Function</td>
<td>System Include File</td>
<td>Function Prototype</td>
<td>Description</td>
</tr>
<tr>
<td>--------------</td>
<td>---------------------</td>
<td>---------------------------------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>getchar¹</td>
<td>stdio.h</td>
<td>int getchar(void);</td>
<td>Reads a single character from stdin.</td>
</tr>
<tr>
<td>getenv</td>
<td>stdlib.h</td>
<td>char *getenv(const char *varname);</td>
<td>Searches environment variables for varname.</td>
</tr>
<tr>
<td>gets</td>
<td>stdio.h</td>
<td>char *gets(char *buffer);</td>
<td>Reads a string from stdin, and stores it in buffer.</td>
</tr>
<tr>
<td>getwc⁶</td>
<td>stdio.h wchar.h</td>
<td>wint_t getwc(FILE *stream);</td>
<td>Reads the next multibyte character from stream, converts it to a wide character and advances the associated file position indicator for stream.</td>
</tr>
<tr>
<td>getwchar⁶</td>
<td>wchar.h</td>
<td>wint_t getwchar(void);</td>
<td>Reads the next multibyte character from stdin, converts it to a wide character, and advances the associated file position indicator for stdin.</td>
</tr>
<tr>
<td>gmtime</td>
<td>time.h</td>
<td>struct tm *gmtime(const time_t *time);</td>
<td>Converts a time value to a structure of type tm.</td>
</tr>
<tr>
<td>gmtime64</td>
<td>time.h</td>
<td>struct tm *gmtime64(const time64_t *time);</td>
<td>Converts a time value to a structure of type tm.</td>
</tr>
<tr>
<td>gmtime_r</td>
<td>time.h</td>
<td>struct tm *gmtime_r (const time_t *time, struct tm *result);</td>
<td>Converts a time value to a structure of type tm. (Restartable version of gmtime.)</td>
</tr>
<tr>
<td>gmtime64_r</td>
<td>time.h</td>
<td>struct tm *gmtime64_r (const time64_t *time, struct tm *result);</td>
<td>Converts a time value to a structure of type tm. (Restartable version of gmtime64.)</td>
</tr>
<tr>
<td>hypot</td>
<td>math.h</td>
<td>double hypot(double side1, double side2);</td>
<td>Calculates the hypotenuse of a right-angled triangle with sides of length side1 and side2.</td>
</tr>
<tr>
<td>isalnum</td>
<td>ctype.h</td>
<td>int isalnum(int c);</td>
<td>Tests if c is alphanumeric.</td>
</tr>
<tr>
<td>isalpha</td>
<td>ctype.h</td>
<td>int isalpha(int c);</td>
<td>Tests if c is alphabetic.</td>
</tr>
<tr>
<td>isascii⁴</td>
<td>ctype.h</td>
<td>int isascii(int c);</td>
<td>Tests if c is within the 7-bit US-ASCII range.</td>
</tr>
<tr>
<td>isblank</td>
<td>ctype.h</td>
<td>int isblank(int c);</td>
<td>Tests if c is a blank or tab character.</td>
</tr>
<tr>
<td>iscntrl</td>
<td>ctype.h</td>
<td>int iscntrl(int c);</td>
<td>Tests if c is a control character.</td>
</tr>
<tr>
<td>isdigit</td>
<td>ctype.h</td>
<td>int isdigit(int c);</td>
<td>Tests if c is a decimal digit.</td>
</tr>
<tr>
<td>isgraph</td>
<td>ctype.h</td>
<td>int isgraph(int c);</td>
<td>Tests if c is a printable character excluding the space.</td>
</tr>
<tr>
<td>islower</td>
<td>ctype.h</td>
<td>int islower(int c);</td>
<td>Tests if c is a lowercase letter.</td>
</tr>
<tr>
<td>isprint</td>
<td>ctype.h</td>
<td>int isprint(int c);</td>
<td>Tests if c is a printable character including the space.</td>
</tr>
<tr>
<td>Function</td>
<td>System Include File</td>
<td>Function Prototype</td>
<td>Description</td>
</tr>
<tr>
<td>--------------</td>
<td>---------------------</td>
<td>----------------------------------</td>
<td>------------------------------------------------------------</td>
</tr>
<tr>
<td>ispunct</td>
<td>ctype.h</td>
<td>int ispunct(int c);</td>
<td>Tests if c is a punctuation character.</td>
</tr>
<tr>
<td>isspace</td>
<td>ctype.h</td>
<td>int isspace(int c);</td>
<td>Tests if c is a whitespace character.</td>
</tr>
<tr>
<td>isupper</td>
<td>ctype.h</td>
<td>int isupper(int c);</td>
<td>Tests if c is an uppercase letter.</td>
</tr>
<tr>
<td>iswalnum</td>
<td>wchar.h</td>
<td>int iswalnum (wint_t wc);</td>
<td>Checks for any alphanumeric wide character.</td>
</tr>
<tr>
<td>iswalpha</td>
<td>wchar.h</td>
<td>int iswalpha (wint_t wc);</td>
<td>Checks for any alphabetic wide character.</td>
</tr>
<tr>
<td>iswblank</td>
<td>wchar.h</td>
<td>int iswblank (wint_t wc);</td>
<td>Checks for any blank or tab wide character.</td>
</tr>
<tr>
<td>iswcntrl</td>
<td>wchar.h</td>
<td>int iswcntrl (wint_t wc);</td>
<td>Tests for any control wide character.</td>
</tr>
<tr>
<td>iswctype</td>
<td>wchar.h</td>
<td>int iswctype(wint_t wc,</td>
<td>Determines whether or not the wide character wc has the</td>
</tr>
<tr>
<td></td>
<td></td>
<td>wchar_t_t wc_prop);</td>
<td>property wc_prop.</td>
</tr>
<tr>
<td>iswdigit</td>
<td>wchar.h</td>
<td>int iswdigit (wint_t wc);</td>
<td>Checks for any decimal-digit wide character.</td>
</tr>
<tr>
<td>iswgraph</td>
<td>wchar.h</td>
<td>int iswgraph (wint_t wc);</td>
<td>Checks for any printing wide character except for the</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>wide-character space.</td>
</tr>
<tr>
<td>iswlower</td>
<td>wchar.h</td>
<td>int iswlower (wint_t wc);</td>
<td>Checks for any lowercase wide character.</td>
</tr>
<tr>
<td>iswprint</td>
<td>wchar.h</td>
<td>int iswprint (wint_t wc);</td>
<td>Checks for any printing wide character.</td>
</tr>
<tr>
<td>iswpunct</td>
<td>wchar.h</td>
<td>int iswpunct (wint_t wc);</td>
<td>Test for a wide non-alphanumeric, non-space character.</td>
</tr>
<tr>
<td>iswspace</td>
<td>wchar.h</td>
<td>int iswspace (wint_t wc);</td>
<td>Checks for any wide character that corresponds to an</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>implementation-defined set of wide characters for which</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>iswalnum is false.</td>
</tr>
<tr>
<td>iswupper</td>
<td>wchar.h</td>
<td>int iswupper (wint_t wc);</td>
<td>Checks for any uppercase wide character.</td>
</tr>
<tr>
<td>iswxdigit</td>
<td>wchar.h</td>
<td>int iswxdigit (wint_t wc);</td>
<td>Checks for any hexadecimal digit character.</td>
</tr>
<tr>
<td>isxdigit</td>
<td>wchar.h</td>
<td>int isxdigit(int c);</td>
<td>Tests if c is a hexadecimal digit.</td>
</tr>
<tr>
<td>j0</td>
<td>math.h</td>
<td>double j0(double x);</td>
<td>Calculates the Bessel function value of the first kind of</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>order 0.</td>
</tr>
<tr>
<td>j1</td>
<td>math.h</td>
<td>double j1(double x);</td>
<td>Calculates the Bessel function value of the first kind of</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>order 1.</td>
</tr>
<tr>
<td>jn</td>
<td>math.h</td>
<td>double jn(int n, double x);</td>
<td>Calculates the Bessel function value of the first kind of</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>order n.</td>
</tr>
<tr>
<td>Function</td>
<td>System Include File</td>
<td>Function Prototype</td>
<td>Description</td>
</tr>
<tr>
<td>-----------</td>
<td>---------------------</td>
<td>--------------------</td>
<td>-------------</td>
</tr>
<tr>
<td>labs</td>
<td>stdlib.h</td>
<td>long int labs(long int n);</td>
<td>Calculates the absolute value of n.</td>
</tr>
<tr>
<td>ldexp</td>
<td>math.h</td>
<td>double ldexp(double x, int exp);</td>
<td>Returns the value of x multiplied by (2 to the power of exp).</td>
</tr>
<tr>
<td>ldiv</td>
<td>stdlib.h</td>
<td>ldiv_t ldiv(long int numerator, long int denominator);</td>
<td>Calculates the quotient and remainder of numerator/denominator.</td>
</tr>
<tr>
<td>localeconv</td>
<td>locale.h</td>
<td>struct lconv *localeconv(void);</td>
<td>Formats numeric quantities in struct lconv according to the current locale.</td>
</tr>
<tr>
<td>localtime</td>
<td>time.h</td>
<td>struct tm *localtime(const time_t *timeval);</td>
<td>Converts timeval to a structure of type tm.</td>
</tr>
<tr>
<td>localtime64</td>
<td>time.h</td>
<td>struct tm *localtime64(const time64_t *timeval);</td>
<td>Converts timeval to a structure of type tm.</td>
</tr>
<tr>
<td>localtime_r</td>
<td>time.h</td>
<td>struct tm *localtime_r (const time_t *timeval, struct tm *result);</td>
<td>Converts a time value to a structure of type tm. (Restartable version of localtime.)</td>
</tr>
<tr>
<td>localtime64_r</td>
<td>time.h</td>
<td>struct tm *localtime64_r (const time64_t *timeval, struct tm *result);</td>
<td>Converts a time value to a structure of type tm. (Restartable version of localtime64.)</td>
</tr>
<tr>
<td>log</td>
<td>math.h</td>
<td>double log(double x);</td>
<td>Calculates the natural logarithm of x.</td>
</tr>
<tr>
<td>log10</td>
<td>math.h</td>
<td>double log10(double x);</td>
<td>Calculates the base 10 logarithm of x.</td>
</tr>
<tr>
<td>longjmp</td>
<td>setjmp.h</td>
<td>void longjmp(jmp_buf env, int value);</td>
<td>Restores a stack environment previously set in env by the setjmp function.</td>
</tr>
<tr>
<td>malloc</td>
<td>stdlib.h</td>
<td>void *malloc(size_t size);</td>
<td>Reserves a block of storage.</td>
</tr>
<tr>
<td>mblen</td>
<td>stdlib.h</td>
<td>int mblen(const char *string, size_t n);</td>
<td>Determines the length of a multibyte character string.</td>
</tr>
<tr>
<td>mbrlen4</td>
<td>wchar.h</td>
<td>int mbrlen (const char *s, size_t n, mbstate_t *ps);</td>
<td>Determines the length of a multibyte character. (Restartable version of mblen.)</td>
</tr>
<tr>
<td>mbtowc4</td>
<td>wchar.h</td>
<td>int mbtowc (wchar_t *pwc, const char *s, size_t n, mbstate_t *ps);</td>
<td>Convert a multibyte character to a wide character (Restartable version of mbtowc.)</td>
</tr>
<tr>
<td>mbsinit4</td>
<td>wchar.h</td>
<td>int mbsinit (const mbstate_t *ps);</td>
<td>Test state object *ps for initial state.</td>
</tr>
<tr>
<td>mbsrtowcs4</td>
<td>wchar.h</td>
<td>size_t mbsrtowcs (wchar_t *dst, const char **src, size_t len, mbstate_t *ps);</td>
<td>Convert multibyte string to a wide character string. (Restartable version of mbsrtowcs.)</td>
</tr>
<tr>
<td>Function</td>
<td>System Include File</td>
<td>Function Prototype</td>
<td>Description</td>
</tr>
<tr>
<td>------------</td>
<td>---------------------</td>
<td>--------------------</td>
<td>-------------</td>
</tr>
<tr>
<td>mbstowcs</td>
<td>stdlib.h</td>
<td>size_t mbstowcs(wchar_t *pwc, const char *string, size_t n);</td>
<td>Converts the multibyte characters in string to their corresponding wchar_t codes, and stores not more than n codes in pwc.</td>
</tr>
<tr>
<td>mbtowc</td>
<td>stdlib.h</td>
<td>int mbtowc(wchar_t *pwc, const char *string, size_t n);</td>
<td>Stores the wchar_t code corresponding to the first n bytes of multibyte character string into the wchar_t character pwc.</td>
</tr>
<tr>
<td>memchr</td>
<td>string.h</td>
<td>void *memchr(const void *buf, int c, size_t count);</td>
<td>Searches the first count bytes of buf for the first occurrence of c converted to an unsigned character.</td>
</tr>
<tr>
<td>memcmp</td>
<td>string.h</td>
<td>int memcmp(const void *buf1, const void *buf2, size_t count);</td>
<td>Compares up to count bytes of buf1 and buf2.</td>
</tr>
<tr>
<td>memcpy</td>
<td>string.h</td>
<td>void *memcpy(void *dest, const void *src, size_t count);</td>
<td>Copies count bytes of src to dest.</td>
</tr>
<tr>
<td>memmove</td>
<td>string.h</td>
<td>void *memmove(void *dest, const void *src, size_t count);</td>
<td>Copies count bytes of src to dest. Allows copying between objects that overlap.</td>
</tr>
<tr>
<td>memset</td>
<td>string.h</td>
<td>void *memset(void *dest, int c, size_t count);</td>
<td>Sets count bytes of dest to a value c.</td>
</tr>
<tr>
<td>mktime</td>
<td>time.h</td>
<td>time_t mktime(struct tm *time);</td>
<td>Converts local time into calendar time.</td>
</tr>
<tr>
<td>mktime64</td>
<td>time.h</td>
<td>time64_t mktime64(struct tm *time);</td>
<td>Converts local time into calendar time.</td>
</tr>
<tr>
<td>modf</td>
<td>math.h</td>
<td>double modf(double x, double *intptr);</td>
<td>Breaks down the floating-point value x into fractional and integral parts.</td>
</tr>
<tr>
<td>nextafter</td>
<td>math.h</td>
<td>double nextafter(double x, double y);</td>
<td>Calculates the next representable value after x in the direction of y.</td>
</tr>
<tr>
<td>nextafterl</td>
<td>math.h</td>
<td>long double nextafterl(long double x, long double y);</td>
<td>Calculates the next representable value after x in the direction of y.</td>
</tr>
<tr>
<td>nexttoward</td>
<td>math.h</td>
<td>double nexttoward(double x, long double y);</td>
<td>Calculates the next representable value after x in the direction of y.</td>
</tr>
<tr>
<td>nexttowardl</td>
<td>math.h</td>
<td>long double nexttowardl(long double x, long double y);</td>
<td>Calculates the next representable value after x in the direction of y.</td>
</tr>
<tr>
<td>nl_langinfo</td>
<td>langinfo.h</td>
<td>char *nl_langinfo(nl_item item);</td>
<td>Retrieve from the current locale the string that describes the requested information specified by item.</td>
</tr>
<tr>
<td>Function</td>
<td>System Include File</td>
<td>Function Prototype</td>
<td>Description</td>
</tr>
<tr>
<td>----------</td>
<td>---------------------</td>
<td>-------------------</td>
<td>-------------</td>
</tr>
<tr>
<td>perror</td>
<td>stdio.h</td>
<td>void perror(const char *string);</td>
<td>Prints an error message to stderr.</td>
</tr>
<tr>
<td>pow</td>
<td>math.h</td>
<td>double pow(double x, double y);</td>
<td>Calculates the value x to the power y.</td>
</tr>
<tr>
<td>printf</td>
<td>stdio.h</td>
<td>int printf(const char *format-string, arg-list);</td>
<td>Formats and prints characters and values to stdout.</td>
</tr>
<tr>
<td>putc$^1$</td>
<td>stdio.h</td>
<td>int putc(int c, FILE *stream);</td>
<td>Prints c to the output stream.</td>
</tr>
<tr>
<td>putchar$^1$</td>
<td>stdio.h</td>
<td>int putchar(int c);</td>
<td>Prints c to stdout.</td>
</tr>
<tr>
<td>putenv</td>
<td>stdlib.h</td>
<td>int *putenv(const char *varname);</td>
<td>Sets the value of an environment variable by altering an existing variable or creating a new one.</td>
</tr>
<tr>
<td>puts</td>
<td>stdio.h</td>
<td>int puts(const char *string);</td>
<td>Prints a string to stdout.</td>
</tr>
<tr>
<td>putwc$^6$</td>
<td>stdio.h,wchar.h</td>
<td>wint_t putwchar(wchar_t wc, FILE *stream);</td>
<td>Converts the wide character wc to a multibyte character, and writes it to the stream at the current position.</td>
</tr>
<tr>
<td>putwchar$^6$</td>
<td>wchar.h</td>
<td>wint_t putwchar(wchar_t wc);</td>
<td>Converts the wide character wc to a multibyte character and writes it to stdout.</td>
</tr>
<tr>
<td>qsort</td>
<td>stdlib.h</td>
<td>void qsort(void *base, size_t num, size_t width, int(*compare)(const void *element1, const void *element2));</td>
<td>Performs a quick sort of an array of num elements, each of width bytes in size.</td>
</tr>
<tr>
<td>quantexpd32</td>
<td>math.h</td>
<td>_Decimal32 quantized32(_Decimal32 x, _Decimal32 y);</td>
<td>Compute the quantum exponent of a single-precision decimal floating-point value.</td>
</tr>
<tr>
<td>quantexpd64</td>
<td>math.h</td>
<td>_Decimal64 quantized64(_Decimal64 x, _Decimal64 y);</td>
<td>Compute the quantum exponent of a double-precision decimal floating-point value.</td>
</tr>
<tr>
<td>quantexpd128</td>
<td>math.h</td>
<td>_Decimal128 quantized128(_Decimal128 x, _Decimal128 y);</td>
<td>Compute the quantum exponent of a quad-precision decimal floating-point value.</td>
</tr>
<tr>
<td>quantized32</td>
<td>math.h</td>
<td>int quantexpd32(_Decimal32 x);</td>
<td>Set the quantum exponent of a single-precision decimal floating-point value to the quantum exponent of another single-precision decimal floating-point value.</td>
</tr>
<tr>
<td>quantized64</td>
<td>math.h</td>
<td>int quantexpd64(_Decimal64 x);</td>
<td>Set the quantum exponent of a double-precision decimal floating-point value to the quantum exponent of another double-precision decimal floating-point value.</td>
</tr>
<tr>
<td>Function</td>
<td>System Include File</td>
<td>Function Prototype</td>
<td>Description</td>
</tr>
<tr>
<td>-------------------</td>
<td>---------------------</td>
<td>-------------------------------------------------------------------------------------</td>
<td>----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>quantized128</td>
<td>math.h</td>
<td>int quantexpd128(_Decimal128 x);</td>
<td>Set the quantum exponent of a quad-precision decimal floating-point value to the quantum exponent of another quad-precision decimal floating-point value.</td>
</tr>
<tr>
<td>samequantumd32</td>
<td>math.h</td>
<td><strong>bool</strong> samequantumd32(_Decimal32 x, _Decimal32 y);</td>
<td>Determine if the quantum exponents of two single-precision decimal floating-point values are the same.</td>
</tr>
<tr>
<td>samequantumd64</td>
<td>math.h</td>
<td><strong>bool</strong> samequantumd64(_Decimal64 x, _Decimal64 y);</td>
<td>Determine if the quantum exponents of two double-precision decimal floating-point values are the same.</td>
</tr>
<tr>
<td>samequantumd128</td>
<td>math.h</td>
<td><strong>bool</strong> samequantumd128(_Decimal128 x, _Decimal128 y);</td>
<td>Determine if the quantum exponents of two quad-precision decimal floating-point values are the same.</td>
</tr>
<tr>
<td>raise</td>
<td>signal.h</td>
<td>int raise(int sig);</td>
<td>Sends the signal sig to the running program.</td>
</tr>
<tr>
<td>rand</td>
<td>stdlib.h</td>
<td>int rand(void);</td>
<td>Returns a pseudo-random integer.</td>
</tr>
<tr>
<td>rand_r</td>
<td>stdlib.h</td>
<td>int rand_r(void);</td>
<td>Returns a pseudo-random integer. (Restartable version)</td>
</tr>
<tr>
<td>realloc</td>
<td>stdlib.h</td>
<td>void *realloc(void *ptr, size_t size);</td>
<td>Changes the size of a previously reserved storage block.</td>
</tr>
<tr>
<td>regcomp</td>
<td>regex.h</td>
<td>int regcomp(regex_t *preg, const char *pattern, int cflags);</td>
<td>Compiles the source regular expression pointed to by pattern into an executable version and stores it in the location pointed to by preg.</td>
</tr>
<tr>
<td>regerror</td>
<td>regex.h</td>
<td>size_t regerror(int errcode, const regex_t *preg, char *errbuf, size_t errbuf_size);</td>
<td>Finds the description for the error code errcode for the regular expression preg.</td>
</tr>
<tr>
<td>regexec</td>
<td>regex.h</td>
<td>int regexec(const regex_t *preg, const char *string, size_t nmatch, regmatch_t *pmatch, int cflags);</td>
<td>Compares the null-ended string string against the compiled regular expression preg to find a match between the two.</td>
</tr>
<tr>
<td>regfree</td>
<td>regex.h</td>
<td>void regfree(regex_t *preg);</td>
<td>Frees any memory that was allocated by regcomp to implement the regular expression preg.</td>
</tr>
<tr>
<td>remove</td>
<td>stdio.h</td>
<td>int remove(const char *filename);</td>
<td>Deletes the file specified by filename.</td>
</tr>
<tr>
<td>rename</td>
<td>stdio.h</td>
<td>int rename(const char *oldname, const char *newname);</td>
<td>Renames the specified file.</td>
</tr>
<tr>
<td>Function</td>
<td>System Include File</td>
<td>Function Prototype</td>
<td>Description</td>
</tr>
<tr>
<td>----------</td>
<td>---------------------</td>
<td>------------------------------------------------------------------------------------</td>
<td>------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>rewind¹</td>
<td>stdio.h</td>
<td>void rewind(FILE *stream);</td>
<td>Repositions the file pointer associated with stream to the beginning of the file.</td>
</tr>
<tr>
<td>scanf</td>
<td>stdio.h</td>
<td>int scanf(const char *format-string, arg-list);</td>
<td>Reads data from stdin into locations given by arg-list.</td>
</tr>
<tr>
<td>setbuf</td>
<td>stdio.h</td>
<td>void setbuf(FILE *stream, char *buffer);</td>
<td>Controls buffering for stream.</td>
</tr>
<tr>
<td>setjmp</td>
<td>setjmp.h</td>
<td>int setjmp(jmp_buf env);</td>
<td>Saves a stack environment that can be subsequently restored by longjmp.</td>
</tr>
<tr>
<td>setlocale</td>
<td>locale.h</td>
<td>char *setlocale(int category, const char *locale);</td>
<td>Changes or queries variables defined in the locale.</td>
</tr>
<tr>
<td>setvbuf</td>
<td>stdio.h</td>
<td>int setvbuf(FILE *stream, char *buf, int type, size_t size);</td>
<td>Controls buffering and buffer size for stream.</td>
</tr>
<tr>
<td>signal</td>
<td>signal.h</td>
<td>void(*signal (int sig, void(*func)(int))) (int);</td>
<td>Registers func as a signal handler for the signal sig.</td>
</tr>
<tr>
<td>sin</td>
<td>math.h</td>
<td>double sin(double x);</td>
<td>Calculates the sine of x.</td>
</tr>
<tr>
<td>sinh</td>
<td>math.h</td>
<td>double sinh(double x);</td>
<td>Calculates the hyperbolic sine of x.</td>
</tr>
<tr>
<td>snprintf</td>
<td>stdio.h</td>
<td>int snprintf(char <em>outbuf, size_t n, const char</em>, ...)</td>
<td>Same as sprintf except that the function will stop after n characters have been written to outbuf.</td>
</tr>
<tr>
<td>sprintf</td>
<td>stdio.h</td>
<td>int sprintf(char *buffer, const char *format-string, arg-list);</td>
<td>Formats and stores characters and values in buffer.</td>
</tr>
<tr>
<td>sqrt</td>
<td>math.h</td>
<td>double sqrt(double x);</td>
<td>Calculates the square root of x.</td>
</tr>
<tr>
<td>srand</td>
<td>stdlib.h</td>
<td>void srand(unsigned int seed);</td>
<td>Sets the seed for the pseudo-random number generator.</td>
</tr>
<tr>
<td>sscanf</td>
<td>stdio.h</td>
<td>int sscanf(const char *buffer, const char *format);</td>
<td>Reads data from buffer into the locations given by arg-list.</td>
</tr>
<tr>
<td>strcasecmp</td>
<td>strings.h</td>
<td>int strcasecmp(const char *string1, const char *string2);</td>
<td>Compares strings without case sensitivity.</td>
</tr>
<tr>
<td>strcat</td>
<td>string.h</td>
<td>char *strcat(char *string1, const char *string2);</td>
<td>Concatenates string2 to string1.</td>
</tr>
<tr>
<td>strchr</td>
<td>string.h</td>
<td>char *strchr(const char *string, int c);</td>
<td>Locates the first occurrence of c in string.</td>
</tr>
<tr>
<td>strcmp</td>
<td>string.h</td>
<td>int strcmp(const char *string1, const char *string2);</td>
<td>Compares the value of string1 to string2.</td>
</tr>
<tr>
<td>Function</td>
<td>System Include File</td>
<td>Function Prototype</td>
<td>Description</td>
</tr>
<tr>
<td>-------------</td>
<td>---------------------</td>
<td>--------------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>strcoll</td>
<td>string.h</td>
<td>int strcoll(const char *string1, const char *string2);</td>
<td>Compares two strings using the collating sequence in the current locale.</td>
</tr>
<tr>
<td>strcpy</td>
<td>string.h</td>
<td>char *strcpy(char *string1, const char *string2);</td>
<td>Copies string2 into string1.</td>
</tr>
<tr>
<td>strcsnpn</td>
<td>string.h</td>
<td>size_t strcsnpn(const char *string1, const char *string2);</td>
<td>Returns the length of the initial substring of string1 consisting of characters not contained in string2.</td>
</tr>
<tr>
<td>strerror</td>
<td>string.h</td>
<td>char *strerror(int errnum);</td>
<td>Maps the error number in errnum to an error message string.</td>
</tr>
<tr>
<td>strfmon⁴</td>
<td>wchar.h</td>
<td>int strfmon (char *s, size_t maxsize, const char *format, ...);</td>
<td>Converts monetary value to string.</td>
</tr>
<tr>
<td>strftime</td>
<td>time.h</td>
<td>size_t strftime (char *dest, size_t maxsize, const char *format, const struct tm *timeptr);</td>
<td>Stores characters in an array pointed to by dest, according to the string determined by format.</td>
</tr>
<tr>
<td>strlen</td>
<td>string.h</td>
<td>size_t strlen(const char *string);</td>
<td>Calculates the length of string.</td>
</tr>
<tr>
<td>strncasecmp</td>
<td>strings.h</td>
<td>int strncasecmp(const char *string1, const char *string2, size_t count);</td>
<td>Compares strings without case sensitivity.</td>
</tr>
<tr>
<td>strncat</td>
<td>string.h</td>
<td>char *strncat(char *string1, const char *string2, size_t count);</td>
<td>Concatenates up to count characters of string2 to string1.</td>
</tr>
<tr>
<td>strncmp</td>
<td>string.h</td>
<td>int strncmp(const char *string1, const char *string2, size_t count);</td>
<td>Compares up to count characters of string1 and string2.</td>
</tr>
<tr>
<td>strncpy</td>
<td>string.h</td>
<td>char *strncpy(char *string1, const char *string2, size_t count);</td>
<td>Copies up to count characters of string2 to string1.</td>
</tr>
<tr>
<td>strpbrk</td>
<td>string.h</td>
<td>char *strpbrk(const char *string1, const char *string2);</td>
<td>Locates the first occurrence in string1 of any character in string2.</td>
</tr>
<tr>
<td>strptime⁴</td>
<td>time.h</td>
<td>char *strptime (const char *buf, const char *format, struct tm *tm);</td>
<td>Date and time conversion</td>
</tr>
<tr>
<td>strrchr</td>
<td>string.h</td>
<td>char *strrchr(const char *string, int c);</td>
<td>Locates the last occurrence of c in string.</td>
</tr>
<tr>
<td>strspn</td>
<td>string.h</td>
<td>size_t strspn(const char *string1, const char *string2);</td>
<td>Returns the length of the initial substring of string1 consisting of characters contained in string2.</td>
</tr>
<tr>
<td>Function</td>
<td>System Include File</td>
<td>Function Prototype</td>
<td>Description</td>
</tr>
<tr>
<td>-----------</td>
<td>---------------------</td>
<td>--------------------</td>
<td>-------------</td>
</tr>
<tr>
<td>strstr</td>
<td>string.h</td>
<td><code>char *strstr(const char *string1, const char *string2);</code></td>
<td>Returns a pointer to the first occurrence of <code>string2</code> in <code>string1</code>.</td>
</tr>
<tr>
<td>strtod</td>
<td>stdlib.h</td>
<td><code>double strtod(const char *nptr, char **endptr);</code></td>
<td>Converts <code>nptr</code> to a double precision value.</td>
</tr>
<tr>
<td>strtod32</td>
<td>stdlib.h</td>
<td><code>_Decimal32 strtod32(const char *nptr, char **endptr);</code></td>
<td>Converts <code>nptr</code> to a single-precision decimal floating-point value.</td>
</tr>
<tr>
<td>strtod64</td>
<td>stdlib.h</td>
<td><code>_Decimal64 strtod64(const char *nptr, char **endptr);</code></td>
<td>Converts <code>nptr</code> to a double-precision decimal floating-point value.</td>
</tr>
<tr>
<td>strtod128</td>
<td>stdlib.h</td>
<td><code>_Decimal128 strtod128(const char *nptr, char **endptr);</code></td>
<td>Converts <code>nptr</code> to a quad-precision decimal floating-point value.</td>
</tr>
<tr>
<td>strtof</td>
<td>stdlib.h</td>
<td><code>float strtof(const char *nptr, char **endptr);</code></td>
<td>Converts <code>nptr</code> to a float value.</td>
</tr>
<tr>
<td>strtok</td>
<td>string.h</td>
<td><code>char *strtok(char *string1, const char *string2);</code></td>
<td>Locates the next token in <code>string1</code> delimited by the next character in <code>string2</code>.</td>
</tr>
<tr>
<td>strtok_r</td>
<td>string.h</td>
<td><code>char *strtok_r(char *string, const char *seps, char **lasts);</code></td>
<td>Locates the next token in <code>string</code> delimited by the next character in <code>seps</code>. (Restartable version of <code>strtok</code>.)</td>
</tr>
<tr>
<td>strtol</td>
<td>stdlib.h</td>
<td><code>long int strtol(const char *nptr, char **endptr, int base);</code></td>
<td>Converts <code>nptr</code> to a signed long integer.</td>
</tr>
<tr>
<td>strtold</td>
<td>stdlib.h</td>
<td><code>long double strtold(const char *nptr, char **endptr);</code></td>
<td>Converts <code>nptr</code> to a long double value.</td>
</tr>
<tr>
<td>strtoul</td>
<td>stdlib.h</td>
<td><code>unsigned long int strtoul(const char *string1, char **string2, int base);</code></td>
<td>Converts <code>string1</code> to an unsigned long integer.</td>
</tr>
<tr>
<td>strxfrm</td>
<td>string.h</td>
<td><code>size_t strxfrm(char *string1, const char *string2, size_t count);</code></td>
<td>Converts <code>string2</code> and places the result in <code>string1</code>. The conversion is determined by the program’s current locale.</td>
</tr>
<tr>
<td>swprintf</td>
<td>wchar.h</td>
<td><code>int swprintf(wchar_t *wcsbuffer, size_t n, const wchar_t *format, arg-list);</code></td>
<td>Formats and stores a series of wide characters and values into the wide-character buffer <code>wcsbuffer</code>.</td>
</tr>
<tr>
<td>swscanf</td>
<td>wchar.h</td>
<td><code>int swscanf (const wchar_t *buffer, const wchar_t *format, arg-list)</code></td>
<td>Reads data from <code>buffer</code> into the locations given by <code>arg-list</code>.</td>
</tr>
<tr>
<td>system</td>
<td>stdlib.h</td>
<td><code>int system(const char *string);</code></td>
<td>Passes <code>string</code> to the system command analyzer.</td>
</tr>
<tr>
<td>tan</td>
<td>math.h</td>
<td><code>double tan(double x);</code></td>
<td>Calculates the tangent of <code>x</code>.</td>
</tr>
<tr>
<td>Function</td>
<td>System Include File</td>
<td>Function Prototype</td>
<td>Description</td>
</tr>
<tr>
<td>----------</td>
<td>---------------------</td>
<td>--------------------</td>
<td>-------------</td>
</tr>
<tr>
<td>tanh</td>
<td>math.h</td>
<td>double tanh(double x);</td>
<td>Calculates the hyperbolic tangent of x.</td>
</tr>
<tr>
<td>time</td>
<td>time.h</td>
<td>time_t time(time_t *timeptr);</td>
<td>Returns the current calendar time.</td>
</tr>
<tr>
<td>time64</td>
<td>time.h</td>
<td>time64_t time64(time64_t *timeptr);</td>
<td>Returns the current calendar time.</td>
</tr>
<tr>
<td>tmpfile</td>
<td>stdio.h</td>
<td>FILE *tmpfile(void);</td>
<td>Creates a temporary binary file and opens it.</td>
</tr>
<tr>
<td>tmpnam</td>
<td>stdio.h</td>
<td>char *tmpnam(char *string);</td>
<td>Generates a temporary file name.</td>
</tr>
<tr>
<td>toascii</td>
<td>ctype.h</td>
<td>int toascii(int c);</td>
<td>Converts c to a character in the 7-bit US-ASCII character set.</td>
</tr>
<tr>
<td>tolower</td>
<td>ctype.h</td>
<td>int tolower(int c);</td>
<td>Converts c to lowercase.</td>
</tr>
<tr>
<td>toupper</td>
<td>ctype.h</td>
<td>int toupper(int c);</td>
<td>Converts c to uppercase.</td>
</tr>
<tr>
<td>towctrans</td>
<td>wctype.h</td>
<td>wint_t towctrans(wint_t wc, wctrans_t desc);</td>
<td>Translates the wide character wc based on the mapping described by desc.</td>
</tr>
<tr>
<td>towlower</td>
<td>wctype.h</td>
<td>wint_t towlower (wint_t wc);</td>
<td>Converts uppercase letter to lowercase letter.</td>
</tr>
<tr>
<td>towupper</td>
<td>wctype.h</td>
<td>wint_t towupper (wint_t wc);</td>
<td>Converts lowercase letter to uppercase letter.</td>
</tr>
<tr>
<td>ungetc</td>
<td>stdio.h</td>
<td>int ungetc(int c, FILE *stream);</td>
<td>Pushes c back onto the input stream.</td>
</tr>
<tr>
<td>ungetwc</td>
<td>stdio.h, wchar.h</td>
<td>wint_t ungetwc(wint_t wc, FILE *stream);</td>
<td>Pushes the wide character wc back onto the input stream.</td>
</tr>
<tr>
<td>va_arg</td>
<td>stdarg.h</td>
<td>var_type va_arg(va_list arg_ptr, var_type);</td>
<td>Returns the value of one argument and modifies arg_ptr to point to the next argument.</td>
</tr>
<tr>
<td>va_copy</td>
<td>stdarg.h</td>
<td>void va_copy(va_list dest, va_list src);</td>
<td>Initializes dest as a copy of src.</td>
</tr>
<tr>
<td>va_end</td>
<td>stdarg.h</td>
<td>void va_end(va_list arg_ptr);</td>
<td>Facilitates normal return from variable argument list processing.</td>
</tr>
<tr>
<td>va_start</td>
<td>stdarg.h</td>
<td>void va_start(va_list arg_ptr, variable_name);</td>
<td>Initializes arg_ptr for subsequent use by va_arg and va_end.</td>
</tr>
<tr>
<td>vfprintf</td>
<td>stdio.h, stdarg.h</td>
<td>int vfprintf(FILE *stream, const char *format, va_list arg_ptr);</td>
<td>Formats and prints characters to the output stream using a variable number of arguments.</td>
</tr>
<tr>
<td>vfscanf</td>
<td>stdio.h, stdarg.h</td>
<td>int vfscanf(FILE *stream, const char *format, va_list arg_ptr);</td>
<td>Reads data from a specified stream into locations given by a variable number of arguments.</td>
</tr>
<tr>
<td>Function</td>
<td>System Include File</td>
<td>Function Prototype</td>
<td>Description</td>
</tr>
<tr>
<td>--------------</td>
<td>---------------------</td>
<td>--------------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>vfwprintf⁶</td>
<td>stdarg.h</td>
<td>int vfwprintf(FILE *stream, const wchar_t *format, va_list arg);</td>
<td>Equivalent to fprintf, except that the variable argument list is replaced by arg.</td>
</tr>
<tr>
<td>vfwscanf</td>
<td>stdio.h</td>
<td>int vfwscanf(FILE *stream, const wchar_t *format, va_list arg_ptr);</td>
<td>Reads wide data from a specified stream into locations given by a variable number of arguments.</td>
</tr>
<tr>
<td>vprintf</td>
<td>stdio.h</td>
<td>int vprintf(const char *format, va_list arg_ptr);</td>
<td>Formats and prints characters to stdout using a variable number of arguments.</td>
</tr>
<tr>
<td>vscanf</td>
<td>stdio.h</td>
<td>int vscanf(const char *format, va_list arg_ptr);</td>
<td>Reads data from stdin into locations given by a variable number of arguments.</td>
</tr>
<tr>
<td>vsprintf</td>
<td>stdio.h</td>
<td>int vsprintf(char *target-string, const char *format, va_list arg_ptr);</td>
<td>Formats and stores characters in a buffer using a variable number of arguments.</td>
</tr>
<tr>
<td>vsnprintf</td>
<td>stdio.h</td>
<td>int vsnprintf(char <em>target-outbuf, size_t n, const char</em>, va_list);</td>
<td>Same as vsprintf except that the function will stop after n characters have been written to outbuf.</td>
</tr>
<tr>
<td>vsscanf</td>
<td>stdio.h</td>
<td>int vsscanf(const char *buffer, const char *format, va_list arg_ptr);</td>
<td>Reads data from a buffer into locations given by a variable number of arguments.</td>
</tr>
<tr>
<td>vswprintf</td>
<td>stdio.h</td>
<td>int vswprintf(const wchar_t *wcsbuffer, size_t n, const wchar_t *format, va_list arg);</td>
<td>Formats and stores a series of wide characters and values in the buffer wcsbuffer.</td>
</tr>
<tr>
<td>vswscanf</td>
<td>stdio.h</td>
<td>int vswscanf(const wchar_t *buffer, const wchar_t *format, va_list arg_ptr);</td>
<td>Reads wide data from a buffer into locations given by a variable number of arguments.</td>
</tr>
<tr>
<td>vwprintf⁶</td>
<td>stdio.h</td>
<td>int vwprintf(const wchar_t *format, va_list arg);</td>
<td>Equivalent to vprintf, except that the variable argument list is replaced by arg.</td>
</tr>
<tr>
<td>vwscanf</td>
<td>stdio.h</td>
<td>int vwscanf(const wchar_t *format, va_list arg_ptr);</td>
<td>Reads wide data from stdin into locations given by a variable number of arguments.</td>
</tr>
<tr>
<td>wcrtomb⁴</td>
<td>wchar.h</td>
<td>int wcrtomb (char *s, wchar_t *wchar, mbstate_t *pss);</td>
<td>Converts a wide character to a multibyte character. (Restartable version of wcrtob.)</td>
</tr>
<tr>
<td>wcscat</td>
<td>wchar.h</td>
<td>wchar_t *wcscat(wchar_t *string1, const wchar_t *string2);</td>
<td>Appends a copy of the string pointed to by string2 to the end of the string pointed to by string1.</td>
</tr>
</tbody>
</table>
| wcschr        | wchar.h             | wchar_t *wcschr(const wchar_t *string, wchar_t *character); | Searches the wide-character string pointed to by string for the occurrence of character.
<table>
<thead>
<tr>
<th>Function</th>
<th>System Include File</th>
<th>Function Prototype</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>wcscmp</td>
<td>wchar.h</td>
<td>int wcscmp(const wchar_t *string1, const wchar_t *string2);</td>
<td>Compares two wide-character strings, *string1 and *string2.</td>
</tr>
<tr>
<td>wcscoll⁴</td>
<td>wchar.h</td>
<td>int wcscoll (const wchar_t *wcs1, const wchar_t *wcs2);</td>
<td>Compares two wide-character strings using the collating sequence in the current locale.</td>
</tr>
<tr>
<td>wcscpy</td>
<td>wchar.h</td>
<td>wchar_t *wcsncpy(wchar_t *string1, const wchar_t *string2);</td>
<td>Copies the contents of *string2 (including the ending wchar_t null character) into *string1.</td>
</tr>
<tr>
<td>wcscspn</td>
<td>wchar.h</td>
<td>size_t wcscspn(const wchar_t *string1, const wchar_t *string2);</td>
<td>Determines the number of wchar_t characters in the initial segment of the string pointed to by *string1 that do not appear in the string pointed to by *string2.</td>
</tr>
<tr>
<td>wcsftime</td>
<td>wchar.h</td>
<td>size_t wcsftime(wchar_t *wdest, size_t maxsize, const wchar_t *format, const struct tm *timeptr);</td>
<td>Converts the time and date specification in the timeptr structure into a wide-character string.</td>
</tr>
<tr>
<td>wcslen</td>
<td>wchar.h</td>
<td>size_t wcslen(const wchar_t *string);</td>
<td>Computes the number of wide-characters in the string pointed to by string.</td>
</tr>
<tr>
<td>wcslocaleconv</td>
<td>locale.h</td>
<td>struct wcslocaleconv *wcslocaleconv(void);</td>
<td>Formats numeric quantities in struct wcslocaleconv according to the current locale.</td>
</tr>
<tr>
<td>wcscncat</td>
<td>wchar.h</td>
<td>wchar_t *wcscncat(wchar_t *string1, const wchar_t *string2, size_t count);</td>
<td>Appends up to count wide characters from string2 to the end of string1, and appends a wchar_t null character to the result.</td>
</tr>
<tr>
<td>wcscncmp</td>
<td>wchar.h</td>
<td>int wcscncmp(const wchar_t *string1, const wchar_t *string2, size_t count);</td>
<td>Compares up to count wide characters in string1 to string2.</td>
</tr>
<tr>
<td>wcscncpy</td>
<td>wchar.h</td>
<td>wchar_t *wcscncpy(wchar_t *string1, const wchar_t *string2, size_t count);</td>
<td>Copies up to count wide characters from string2 to string1.</td>
</tr>
<tr>
<td>wcspbrk</td>
<td>wchar.h</td>
<td>wchar_t *wcspbrk(const wchar_t *string1, const wchar_t *string2);</td>
<td>Locates the first occurrence in the string pointed to by string1 of any wide characters from the string pointed to by string2.</td>
</tr>
<tr>
<td>wcsptime</td>
<td>wchar.h</td>
<td>wchar_t *wcsptime (const wchar_t *buf, const wchar_t *format, struct tm *tm);</td>
<td>Date and time conversion. Equivalent to strptime(), except that it uses wide characters.</td>
</tr>
<tr>
<td>wcsrcrchr</td>
<td>wchar.h</td>
<td>wchar_t *wcsrcrchr(const wchar_t *string, wchar_t character);</td>
<td>Locates the last occurrence of character in the string pointed to by string.</td>
</tr>
<tr>
<td>Function</td>
<td>System Include File</td>
<td>Function Prototype</td>
<td>Description</td>
</tr>
<tr>
<td>------------</td>
<td>---------------------</td>
<td>-------------------------------------------------------------------------------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>wcsrtombs4</td>
<td>wchar.h</td>
<td>size_t wcsrtombs(const wchar_t **dest, const wchar_t **src, size_t len, mbstate_t *ps);</td>
<td>Converts wide character string to multibyte string. (Restartable version of wcstombs.)</td>
</tr>
<tr>
<td>wcsspn</td>
<td>wchar.h</td>
<td>size_t wcsspn(const wchar_t *string1, const wchar_t *string2);</td>
<td>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.</td>
</tr>
<tr>
<td>wcsstr</td>
<td>wchar.h</td>
<td>wchar_t *wcsstr(const wchar_t *wcs1, const wchar_t *wcs2);</td>
<td>Locates the first occurrence of wcs2 in wcs1.</td>
</tr>
<tr>
<td>wcstod</td>
<td>wchar.h</td>
<td>double wcstod(const wchar_t *nptr, wchar_t **endptr);</td>
<td>Converts the initial portion of the wide-character string pointed to by nptr to a double value.</td>
</tr>
<tr>
<td>wcstod32</td>
<td>wchar.h</td>
<td>_Decimal32 wcstod32(const wchar_t *nptr, wchar_t **endptr);</td>
<td>Converts the initial portion of the wide-character string pointed to by nptr to a single-precision decimal floating-point value.</td>
</tr>
<tr>
<td>wcstod64</td>
<td>wchar.h</td>
<td>_Decimal64 wcstod64(const wchar_t *nptr, wchar_t **endptr);</td>
<td>Converts the initial portion of the wide-character string pointed to by nptr to a double-precision decimal floating-point value.</td>
</tr>
<tr>
<td>wcstod128</td>
<td>wchar.h</td>
<td>_Decimal128 wcstod128(const wchar_t *nptr, wchar_t **endptr);</td>
<td>Converts the initial portion of the wide-character string pointed to by nptr to a quad-precision decimal floating-point value.</td>
</tr>
<tr>
<td>wcstof</td>
<td>wchar.h</td>
<td>float wcstof(const wchar_t *nptr, wchar_t **endptr);</td>
<td>Converts the initial portion of the wide-character string pointed to by nptr to a float value.</td>
</tr>
<tr>
<td>wcstok</td>
<td>wchar.h</td>
<td>wchar_t *wcstok(const wchar_t *wcs1, const wchar_t *wcs2, wchar_t **ptr)</td>
<td>Breaks wcs1 into a sequence of tokens, each of which is delimited by a wide character from the wide string pointed to by wcs2.</td>
</tr>
<tr>
<td>wcstol</td>
<td>wchar.h</td>
<td>long int wcstol(const wchar_t *nptr, wchar_t **endptr, int base);</td>
<td>Converts the initial portion of the wide-character string pointed to by nptr to a long integer value.</td>
</tr>
<tr>
<td>wcstold</td>
<td>wchar.h</td>
<td>long double wcstold(const wchar_t *nptr, wchar_t **endptr);</td>
<td>Converts the initial portion of the wide-character string pointed to by nptr to a long double value.</td>
</tr>
<tr>
<td>wcstombs</td>
<td>stdlib.h</td>
<td>size_t wcstombs(const wchar_t *dest, const wchar_t *string, size_t count);</td>
<td>Converts the wchar_t string into a multibyte string dest.</td>
</tr>
<tr>
<td>Function</td>
<td>System Include File</td>
<td>Function Prototype</td>
<td>Description</td>
</tr>
<tr>
<td>-----------</td>
<td>---------------------</td>
<td>--------------------</td>
<td>-------------</td>
</tr>
<tr>
<td>wcstoul</td>
<td>wchar.h</td>
<td>unsigned long int</td>
<td>Converts the initial portion of the wide-character string pointed to by <code>nptr</code> to an unsigned long integer value.</td>
</tr>
<tr>
<td>wcsxfrm</td>
<td>wchar.h</td>
<td>size_t wcsxfrm(wchar_t *wcs1, const wchar_t *wcs2, size_t n);</td>
<td>Transforms a wide-character string to values which represent character collating weights and places the resulting wide-character string into an array.</td>
</tr>
<tr>
<td>wctob</td>
<td>stdarg.h wchar.h</td>
<td>int wctob(wint_t wc);</td>
<td>Determines whether <code>wc</code> corresponds to a member of the extended character set whose multibyte character representation is a single byte when in the initial shift state.</td>
</tr>
<tr>
<td>wcxfrm</td>
<td>wchar.h</td>
<td></td>
<td></td>
</tr>
<tr>
<td>wctomb</td>
<td>stdlib.h</td>
<td>int wctomb(char *string, wchar_t character);</td>
<td>Converts the wchar_t value of <code>character</code> into a multibyte string.</td>
</tr>
<tr>
<td>wctrans</td>
<td>wctype.h</td>
<td>wchar_t wctrans(const char *property);</td>
<td>Constructs a value with type <code>wctrans_t</code> that describes a mapping between wide characters identified by the string argument <code>property</code>.</td>
</tr>
<tr>
<td>wctype</td>
<td>wchar.h</td>
<td>wchar_t wctype(const char *property);</td>
<td>Obtains handle for character property classification.</td>
</tr>
<tr>
<td>wcwidth</td>
<td>wchar.h</td>
<td>int wcswidth(const wchar_t *pwcs, size_t n);</td>
<td>Determine the display width of a wide character string.</td>
</tr>
<tr>
<td>wmemchr</td>
<td>wchar.h</td>
<td>wchar_t *wmemchr(const wchar_t *s, wchar_t c, size_t n);</td>
<td>Locates the first occurrence of <code>c</code> in the initial <code>n</code> wide characters of the object pointed to by <code>s</code>.</td>
</tr>
<tr>
<td>wmemcmp</td>
<td>wchar.h</td>
<td>int wmemcmp(const wchar_t *s1, const wchar_t *s2, size_t n);</td>
<td>Compares the first <code>n</code> wide characters of the object pointed to by <code>s1</code> to the first <code>n</code> characters of the object pointed to by <code>s2</code>.</td>
</tr>
<tr>
<td>wmemcpy</td>
<td>wchar.h</td>
<td>wchar_t *wmemcpy(wchar_t *s1, const wchar_t *s2, size_t n);</td>
<td>Copies <code>n</code> wide characters from the object pointed to by <code>s2</code> to the object pointed to by <code>s1</code>.</td>
</tr>
<tr>
<td>wmemmove</td>
<td>wchar.h</td>
<td>wchar_t *wmemmove(wchar_t *s1, const wchar_t *s2, size_t n);</td>
<td>Copies <code>n</code> wide characters from the object pointed to by <code>s2</code> to the object pointed to by <code>s1</code>.</td>
</tr>
<tr>
<td>wmemset</td>
<td>wchar.h</td>
<td>wchar_t *wmemset(wchar_t *s, wchar_t c, size_t n);</td>
<td>Copies the value of <code>c</code> into each of the first <code>n</code> wide characters of the object pointed to by <code>s</code>.</td>
</tr>
<tr>
<td>wprintf</td>
<td>wchar.h</td>
<td>int wprintf(const wchar_t *format, arg-list);</td>
<td>Equivalent to fwprintf with the argument stdout interposed before the arguments to wprintf.</td>
</tr>
</tbody>
</table>
Table 46. Standard C Library Functions (continued)

<table>
<thead>
<tr>
<th>Function</th>
<th>System Include File</th>
<th>Function Prototype</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>wscanf</td>
<td>wchar.h</td>
<td>int wscanf(const wchar_t *format, arg-list);</td>
<td>Equivalent to fscanf with the argument stdin interposed before the arguments of wscanf.</td>
</tr>
<tr>
<td>y0</td>
<td>math.h</td>
<td>double y0(double x);</td>
<td>Calculates the Bessel function value of the second kind of order 0.</td>
</tr>
<tr>
<td>y1</td>
<td>math.h</td>
<td>double y1(double x);</td>
<td>Calculates the Bessel function value of the second kind of order 1.</td>
</tr>
<tr>
<td>yn</td>
<td>math.h</td>
<td>double yn(int n, double x);</td>
<td>Calculates the Bessel function value of the second kind of order n.</td>
</tr>
</tbody>
</table>

**Note:**
1. This function is not supported for files opened with type=record.
2. This function is not supported for files opened with type=record and mode=ab+, rb+, or wb+.
3. The ILE C compiler only supports fully buffered and line-buffered streams. Since a block and a line are equal to the record length of the opened file, fully buffered and line-buffered streams are supported in the same way. The setbuf() and setvbuf() functions have no effect.
4. This function is not available when LOCALETYPE(*CLD) is specified on the compilation command.
5. This function is available only when SYSIFCOPT(*IFSIO) is specified on the CRTMOD or CRTBNDC command.
6. This function is not available when either LOCALETYPE(*CLD) or SYSIFCOPT(*NOIFSIO) is specified on the compilation command.

ILE C Library Extensions to C Library Functions Table

This table briefly describes all the ILE C library extensions, listed in alphabetical order. This table provides the include file name, and the function prototype for each function.

Table 47. ILE C Library Extensions

<table>
<thead>
<tr>
<th>Function</th>
<th>System Include file</th>
<th>Function prototype</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>_C_Get_Ssn_Handle</td>
<td>stdio.h</td>
<td>_SSN_Handle_T _C_Get_Ssn_Handle (void);</td>
<td>Returns a handle to the C session for use with DSM APIs.</td>
</tr>
<tr>
<td>_C_Quickpool_Debug</td>
<td>stdio.h</td>
<td>_C_Quickpool_Debug_T _C_Quickpool_Debug(_C_Quickpool_Debug_T *newval);</td>
<td>Modifies Quick Pool memory characteristics.</td>
</tr>
<tr>
<td>_C_Quickpool_Init</td>
<td>stdio.h</td>
<td>int _C_Quickpool_Init(unsigned int numpools, unsigned int *cell_sizes, unsigned int *num_cells);</td>
<td>Initializes the use of the Quick Pool memory management algorithm.</td>
</tr>
<tr>
<td>_C_Quickpool_Report</td>
<td>stdio.h</td>
<td>void _C_Quickpool_Report(void);</td>
<td>Generates a spooled file that contains a snapshot of the memory used by the Quick Pool memory management algorithm in the current activation group.</td>
</tr>
</tbody>
</table>
### Table 47. ILE C Library Extensions (continued)

<table>
<thead>
<tr>
<th>Function</th>
<th>System Include file</th>
<th>Function prototype</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>_C_TS_malloc64</td>
<td>stdlib.h</td>
<td>void *C_TS_malloc64(unsigned long long int);</td>
<td>Same as _C_TS_malloc, but takes an unsigned long long int so the user can ask for more than 2 GB of storage on a single request.</td>
</tr>
<tr>
<td>_C_TS_malloc_info</td>
<td>mallocinfo.h</td>
<td>int _C_TS_malloc_info(struct _C_mallinfo_t *output_record, size_t sizeofoutput);</td>
<td>Returns current memory usage information.</td>
</tr>
<tr>
<td>_C_TS_malloc_debug</td>
<td>mallocinfo.h</td>
<td>int _C_TS_malloc_debug(unsigned int dump_level, unsigned int verify_level, struct _C_mallinfo_t *output_record, size_t sizeofoutput);</td>
<td>Returns the same information as _C_TS_malloc_info, plus produces a spool file of detailed information about the memory structure used by C_TS_malloc functions.</td>
</tr>
<tr>
<td>_GetExcData</td>
<td>signal.h</td>
<td>void _GetExcData (_INTRPT_Hndlr_Parms_T *parms);</td>
<td>Retrieves information about an exception from within a signal handler.</td>
</tr>
<tr>
<td>QXXCHGDA</td>
<td>xxdtaa.h</td>
<td>void QXXCHGDA(_DTAA_NAME_T dtaname, short int offset, short int len, char *dtaptr);</td>
<td>Changes the data area specified on dtaname using the data pointed to by dtaptr.</td>
</tr>
<tr>
<td>QXXDTOP</td>
<td>xxcvt.h</td>
<td>void QXXDTOP(unsigned char *pptr, int digits, int fraction, double value);</td>
<td>Converts a double value to a packed decimal value with digits total digits and fraction fractional digits.</td>
</tr>
<tr>
<td>QXXDTOZ</td>
<td>xxcvt.h</td>
<td>void QXXDTOZ(unsigned char *zptr, int digits, int fraction, double value);</td>
<td>Converts a double value to a zoned decimal value with digits total digits and fraction fractional digits.</td>
</tr>
<tr>
<td>QXXITOP</td>
<td>xxcvt.h</td>
<td>void QXXITOP(unsigned char *pptr, int digits, int fraction, int value);</td>
<td>Converts an integer value to a packed decimal value.</td>
</tr>
<tr>
<td>QXXITOZ</td>
<td>xxcvt.h</td>
<td>void QXXITOZ(unsigned char *zptr, int digits, int fraction, int value);</td>
<td>Converts an integer value to a zoned decimal value.</td>
</tr>
<tr>
<td>QXXPTOD</td>
<td>xxcvt.h</td>
<td>double QXXPTOD(unsigned char *pptr, int digits, int fraction);</td>
<td>Converts a packed decimal number to a double value with digits total digits and fraction fractional digits.</td>
</tr>
<tr>
<td>QXXPTOI</td>
<td>xxcvt.h</td>
<td>int QXXPTOI(unsigned char *pptr, int digits, int fraction);</td>
<td>Converts a packed decimal number to an integer value with digits total digits and fraction fractional digits.</td>
</tr>
<tr>
<td>QXXRTVDA</td>
<td>xxdtaa.h</td>
<td>void QXXRTVDA(_DTAA_NAME_T dtaname, short int offset, short int len, char *dtaptr);</td>
<td>Retrieves a copy of the data area specified on dtaname.</td>
</tr>
<tr>
<td>Function</td>
<td>System Include file</td>
<td>Function prototype</td>
<td>Description</td>
</tr>
<tr>
<td>-----------</td>
<td>---------------------</td>
<td>--------------------</td>
<td>-------------</td>
</tr>
<tr>
<td>QXXZTOD</td>
<td>xxcvt.h</td>
<td>double QXXZTOD(unsigned char *zptr, int digits, int fraction);</td>
<td>Converts a zoned decimal number to a double value with ( digits ) total digits and ( fraction ) fractional digits.</td>
</tr>
<tr>
<td>QXXZTOI</td>
<td>xxcvt.h</td>
<td>int QXXZTOI(unsigned char *zptr, int digits, int fraction);</td>
<td>Converts a zoned decimal value to an integer value with ( digits ) total digits and ( fraction ) fractional digits.</td>
</tr>
<tr>
<td>_Racquire</td>
<td>recio.h</td>
<td>int _Racquire(_RFILE *fp, char *dev);</td>
<td>Prepares a device for record I/O operations.</td>
</tr>
<tr>
<td>_Rclose</td>
<td>recio.h</td>
<td>int _Rclose(_RFILE *fp);</td>
<td>Closes a file that is opened for record I/O operations.</td>
</tr>
<tr>
<td>_Rcommit</td>
<td>recio.h</td>
<td>int _Rcommit(char *cmtid);</td>
<td>Completes the current transaction, and establishes a new commitment boundary.</td>
</tr>
<tr>
<td>_Rdelete</td>
<td>recio.h</td>
<td>_RIOFB_T * _Rdelete(_RFILE *fp);</td>
<td>Deletes the currently locked record.</td>
</tr>
<tr>
<td>_Rdevatr</td>
<td>xxfdbk.h, recio.h</td>
<td>_XXDEV_ATR_T * _Rdevatr(_RFILE *fp, char *pgmdev);</td>
<td>Returns a pointer to a copy of the device attributes feedback area for the file referenced by ( fp ) and the device pgmdev.</td>
</tr>
<tr>
<td>_Rfeod</td>
<td>recio.h</td>
<td>int _Rfeod(_RFILE *fp);</td>
<td>Forces an end-of-file condition for the file referenced by ( fp ).</td>
</tr>
<tr>
<td>_Rfeov</td>
<td>recio.h</td>
<td>int _Rfeov(_RFILE *fp);</td>
<td>Forces an end-of-volume condition for the tape file referenced by ( fp ).</td>
</tr>
<tr>
<td>_Rformat</td>
<td>recio.h</td>
<td>void Rformat(_RFILE *fp, char *fmt);</td>
<td>Sets the record format to ( fmt ) for the file referenced by ( fp ).</td>
</tr>
<tr>
<td>_Rindara</td>
<td>recio.h</td>
<td>void _Rindara (_RFILE *fp, char *indic_buf);</td>
<td>Sets up the separate indicator area to be used for subsequent record I/O operations.</td>
</tr>
<tr>
<td>_Riofbk</td>
<td>xxfdbk.h, recio.h</td>
<td>_XXIOFB_T * _Riofbk(_RFILE *fp);</td>
<td>Returns a pointer to a copy of the I/O feedback area for the file referenced by ( fp ).</td>
</tr>
<tr>
<td>_Rlocate</td>
<td>recio.h</td>
<td>_RIOFB_T * _Rlocate(_RFILE *fp, void *key, int klen_rrn, int opts);</td>
<td>Positions to the record in the file associated with ( fp ) and specified by the key, klen_rrn and opt parameters.</td>
</tr>
<tr>
<td>_Ropen</td>
<td>recio.h</td>
<td>_RFILE * _Ropen(const char *filename, const char *mode ...);</td>
<td>Opens a file for record I/O operations.</td>
</tr>
<tr>
<td>_Ropnfbk</td>
<td>xxfdbk.h, recio.h</td>
<td>_XXOPFB_T * _Ropnfbk(_RFILE *fp);</td>
<td>Returns a pointer to a copy of the open feedback area for the file referenced by ( fp ).</td>
</tr>
<tr>
<td>Function</td>
<td>System Include file</td>
<td>Function prototype</td>
<td>Description</td>
</tr>
<tr>
<td>------------</td>
<td>---------------------</td>
<td>-------------------</td>
<td>-------------</td>
</tr>
<tr>
<td>_Rpgmdev</td>
<td>recio.h</td>
<td>int _Rpgmdev(_RFILE *fp, char *dev);</td>
<td>Sets the default program device.</td>
</tr>
<tr>
<td>_Rreadd</td>
<td>recio.h</td>
<td>_RIOFB_T *_Rreadd(_RFILE *fp, void *buf, size_t size, int opts, long rrn);</td>
<td>Reads a record by relative record number.</td>
</tr>
<tr>
<td>_Readf</td>
<td>recio.h</td>
<td>_RIOFB_T *_Readf(_RFILE *fp, void *buf, size_t size, int opts);</td>
<td>Reads the first record.</td>
</tr>
<tr>
<td>_Readindv</td>
<td>recio.h</td>
<td>_RIOFB_T *_Readindv(_RFILE *fp, void *buf, size_t size, int opts);</td>
<td>Reads a record from an invited device.</td>
</tr>
<tr>
<td>_Readk</td>
<td>recio.h</td>
<td>_RIOFB_T *_Readk(_RFILE *fp, void *buf, size_t size, int opts, void *key, int klen);</td>
<td>Reads a record by key.</td>
</tr>
<tr>
<td>_Readl</td>
<td>recio.h</td>
<td>_RIOFB_T *_Readl(_RFILE *fp, void *buf, size_t size, int opts);</td>
<td>Reads the last record.</td>
</tr>
<tr>
<td>_Readn</td>
<td>recio.h</td>
<td>_RIOFB_T *_Readn(_RFILE *fp, void *buf, size_t size, int opts);</td>
<td>Reads the next record.</td>
</tr>
<tr>
<td>_Readnc</td>
<td>recio.h</td>
<td>_RIOFB_T *_Readnc(_RFILE *fp, void *buf, size_t size);</td>
<td>Reads the next changed record in the subfile.</td>
</tr>
<tr>
<td>_Readp</td>
<td>recio.h</td>
<td>_RIOFB_T *_Readp(_RFILE *fp, void *buf, size_t size, int opts);</td>
<td>Reads the previous record.</td>
</tr>
<tr>
<td>_Reads</td>
<td>recio.h</td>
<td>_RIOFB_T *_Reads(_RFILE *fp, void *buf, size_t size, int opts);</td>
<td>Reads the same record.</td>
</tr>
<tr>
<td>_Rrelease</td>
<td>recio.h</td>
<td>int _Rrelease(_RFILE *fp, char *dev);</td>
<td>Makes the specified device ineligible for record I/O operations.</td>
</tr>
<tr>
<td>_Rrlslck</td>
<td>recio.h</td>
<td>int _Rrlslck(_RFILE *fp);</td>
<td>Releases the currently locked record.</td>
</tr>
<tr>
<td>_Rrollbck</td>
<td>recio.h</td>
<td>int _Rrollbck(void);</td>
<td>Reestablishes the last commitment boundary as the current commitment boundary.</td>
</tr>
<tr>
<td>_Rupdate</td>
<td>recio.h</td>
<td>_RIOFB_T *_Rupdate(_RFILE *fp, void *buf, size_t size);</td>
<td>Writes to the record that is currently locked for update.</td>
</tr>
<tr>
<td>_Rupfb</td>
<td>recio.h</td>
<td>_RIOFB_T *_Rupfb(_RFILE *fp);</td>
<td>Updates the feedback structure with information about the last record I/O operation.</td>
</tr>
<tr>
<td>_Rwrite</td>
<td>recio.h</td>
<td>_RIOFB_T *_Rwrite(_RFILE *fp, void *buf, size_t size);</td>
<td>Writes a record to the end of the file.</td>
</tr>
<tr>
<td>_Rwrited</td>
<td>recio.h</td>
<td>_RIOFB_T *_Rwrited(_RFILE *fp, void *buf, size_t size, unsigned long rrn);</td>
<td>Writes a record by relative record number. It only writes over deleted records.</td>
</tr>
<tr>
<td>_Rwriterd</td>
<td>recio.h</td>
<td>_RIOFB_T *_Rwriterd(_RFILE *fp, void *buf, size_t size);</td>
<td>Reads and writes a record.</td>
</tr>
<tr>
<td>Function</td>
<td>System Include file</td>
<td>Function prototype</td>
<td>Description</td>
</tr>
<tr>
<td>------------</td>
<td>---------------------</td>
<td>------------------------------------------------------------------------------------</td>
<td>-------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>_Rwrread</td>
<td>recio.h</td>
<td>_RIOFB_T * _Rwrread(_RFILE *fp, void *inbuf, size_t in_buf_size, void *out_buf, size_t out_buf_size);</td>
<td>Functions as _Rwriterd, except separate buffers may be specified for input and output data.</td>
</tr>
<tr>
<td>__wcsicmp</td>
<td>wchar.h</td>
<td>int __wcsicmp(const wchar_t *string1, const wchar_t *string2);</td>
<td>Compares wide character strings without case sensitivity.</td>
</tr>
<tr>
<td>__wcsnicmp</td>
<td>wchar.h</td>
<td>int __wcsnicmp(const wchar_t *string1, const wchar_t *string2, size_t count);</td>
<td>Compares wide character strings without case sensitivity.</td>
</tr>
</tbody>
</table>
Related information

For additional information about topics related to ILE C/C++ programming on the IBM i platform, refer to the following IBM i publications and IBM i Information Center topics:

(http://www.ibm.com/systems/i/infocenter/)

• The Application programming interfaces topic in the Programming category of the IBM i Information Center provides information for experienced application and system programmers who want to use the application programming interfaces (APIs).

• Application Display Programming, SC41-5715-02 provides information about using DDS to create and maintain displays, creating and working with display files, creating online help information, using UIM to define displays, and using panel groups, records, and documents.

• The Backup and recovery topic in the Systems management category of the IBM i Information Center includes information about how to plan a backup and recovery strategy, how to back up your system, how to manage tape libraries, and how to set up disk protection for your data. It also includes information about the Backup, Recovery and Media Services plug-in to IBM i Navigator, information about recovering your system, and answers to some frequently asked questions about backup and recovery.

• Recovering your system, SC41-5304-09 provides general information about recovery and availability options for the IBM i platform. It describes the options available on the system, compares and contrasts them, and tells where to find more information about them.

• The Control language topic in the Programming category of the IBM i Information Center provides a description of the control language commands. It also provides a wide-ranging discussion of programming topics including a general discussion on objects and libraries, CL programming, controlling flow and communicating between programs, working with objects in CL programs, and creating CL programs. Other topics include predefined and impromptu messages and message handling, defining and creating user-defined commands and menus, application testing, including debug mode, breakpoints, traces, and display functions.

• Communications Management, SC41-5406-02 provides information about work management in a communications environment, communications status, tracing and diagnosing communications problems, error handling and recovery, performance, and specific line speed and subsystem storage information.

• The Files and file systems category in the IBM i Information Center provides information about using files in application programs.

• The globalization topic in the Programming category of the IBM i Information Center provides information for planning, installing, configuring, and using globalization and multilingual support of the IBM i product. It also provides an explanation of the database management of multilingual data and application considerations for a multilingual system.

• The ICF Programming, SC41-5442-00 manual provides information needed to write application programs that use communications and the intersystem communications function (IBM i -ICF). It also contains information about data description specifications (DDS) keywords, system-supplied formats, return codes, file transfer support, and program examples.

• ILE Concepts, SC41-5606-08 explains concepts and terminology pertaining to the Integrated Language Environment architecture of the IBM i licensed program. Topics covered include creating modules, binding, running programs, debugging programs, and handling exceptions.

• The Printing category of information in the IBM i Information Center provides information about how to plan for and configure printing functions, as well as basic printing information.

• The Basic printing topic provides specific information about printing elements and concepts of the IBM i product, printer file and print spooling support, and printer connectivity.
• The Security category in the IBM i Information Center provides information about how to set up and plan for your system security, how to secure network and communications applications, and how to add highly secure cryptographic processing capability to your product. It also includes information about object signing and signature validation, identity mapping, and solutions to Internet security risks.

• Security reference, SC41-5302-10 tells how system security support can be used to protect the system and data from being used by people who do not have the proper authorization, protect data from intentional or unintentional damage or destruction, keep security information up-to-date, and set up security on the system.

• The Systems management category in the IBM i Information Center provides information about the system unit control panel, starting and stopping the system, using tapes and diskettes, working with program temporary fixes, as well as handling problems.

• ILE C/C++ Language Reference contains reference information for the C/C++ languages.

• ILE C/C++ Compiler Reference contains reference information about using preprocessor statements, macros defined by and pragmas recognized by the ILE C/C++ compiler, command line options for both IBM i and QShell working environments, and I/O considerations for the IBM i environment.

• ILE C/C++ Programmer's Guide provides information about how to develop applications using the ILE C language. It includes information about creating, running and debugging programs. It also includes programming considerations for interlanguage program and procedure calls, locales, handling exceptions, database, externally described and device files. Some performance tips are also described.

For more information about programming utilities, see the following books at the IBM Publications Center:

• ADTS/400: Programming Development Manager, SC09-1771-00
• ADTS for AS/400: Screen Design Aid, SC09-2604-00
• ADTS for AS/400: Source Entry Utility, SC09-2605-00
This information was developed for products and services offered in the U.S.A.

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 grant 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 (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.
1623-14, Shimotsuruma, Yamato-shi
Kanagawa 242-8502 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:

IBM Corporation
Software Interoperability Coordinator, Department YBWA
3605 Highway 52 N
Rochester, MN 55901
U.S.A.
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 document 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.

Any performance data contained herein was determined in a controlled environment. Therefore, the results obtained in other operating environments may vary significantly. Some measurements may have been made on development-level systems and there is no guarantee that these measurements will be the same on generally available systems. Furthermore, some measurements may have been estimated through extrapolation. Actual results may vary. Users of this document should verify the applicable data for their specific environment.

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.

All IBM prices shown are IBM's suggested retail prices, are current and are subject to change without notice. Dealer prices may vary.

This information is for planning purposes only. The information herein is subject to change before the products described become available.

This information contains examples of data and reports used in daily business operations. To illustrate them as completely as possible, the examples include the names of individuals, companies, brands, and products. All of these names are fictitious and any similarity to the names and addresses used by an actual business enterprise is entirely coincidental.

COPYRIGHT LICENSE:

This information contains 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.

Each copy or any portion of these sample programs or any derivative work, must include a copyright notice as follows:

© (your company name) (year). Portions of this code are derived from IBM Corp. Sample Programs.

© Copyright IBM Corp. _enter the year or years_.

If you are viewing this information softcopy, the photographs and color illustrations may not appear.

Programming interface information

This ILE C/C++ Runtime Library Functions publication documents intended Programming Interfaces that allow the customer to write programs to obtain the services of IBM i.
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.

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 of Oracle, Inc. in the United States, other countries, or both.

Other product and service names might be trademarks of IBM or other companies.

Terms and conditions

Permissions for the use of these publications is granted subject to the following terms and conditions.

**Personal Use:** You may reproduce these publications for your personal, noncommercial use provided that all proprietary notices are preserved. You may not distribute, display or make derivative works of these publications, or any portion thereof, without the express consent of IBM.

**Commercial Use:** You may reproduce, distribute and display these publications solely within your enterprise provided that all proprietary notices are preserved. You may not make derivative works of these publications, or reproduce, distribute or display these publications or any portion thereof outside your enterprise, without the express consent of IBM.

Except as expressly granted in this permission, no other permissions, licenses or rights are granted, either express or implied, to the publications or any information, data, software or other intellectual property contained therein.

IBM reserves the right to withdraw the permissions granted herein whenever, in its discretion, the use of the publications is detrimental to its interest or, as determined by IBM, the above instructions are not being properly followed.

You may not download, export or re-export this information except in full compliance with all applicable laws and regulations, including all United States export laws and regulations.

IBM MAKES NO GUARANTEE ABOUT THE CONTENT OF THESE PUBLICATIONS. THE PUBLICATIONS ARE PROVIDED "AS-IS" AND WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING BUT NOT LIMITED TO IMPLIED WARRANTIES OF MERCHANTABILITY, NON-INFRINGEMENT, AND FITNESS FOR A PARTICULAR PURPOSE.
Index

Special Characters

__EXBDY built-in 4
__VBDY built-in 4
_C_Get_Ssn_Handle() function 79
_C_Quickpool_Debug() function
Quick Pool Memory Manager Debug 91
_C_Quickpool_Init() function
Initialize Quick Pool Memory Manager 93
_C_Quickpool_Report() function
Generate Quick Pool Memory Manager Report 95
_C_TS_malloc 218, 610
_C_TS_malloc_debug 610
_C_TS_malloc_debug() function 103
_C_TS_malloc_info 610
_C_TS_malloc_info() function 105
_C_TS_malloc64 218, 610
_EXBDY macro 4
_fputchar() function 144
_gctv() function 173
_GetExcData() function 176
_INTRPT_Hndlr_Parms_T 2
_itoa() function 198
_ltoa() function 214
_Racquire() function 285
_Rclosel() function 286
_Rcommit() function 287
_Rdelete() function 289
_Rdevatr() function 290
_Rfeod() function 304
_Rfeov() function 305
_Rformat() function 306
_Rindara() function 308
_Riofbk() function 310
_Rlocate() function 312
_Ropen() function 315
_Ropnfbk() function 319
_Rpgmdev() function 320
_Rread() function 321
_Rreadf() function 323
_Rreadindv() function 325
_Rreadk() function 326
_Rread() function 331
_Rreadn() function 332
_Rreadnc() function 334
_Rreadp() function 337
_Rreads() function 339
_Rrelease() function 341
_Rrlslck() function 342
_Rrollbck() function 343
_Ruptdate() function 345
_Rupfb() function 346
_Rwrd() function 348
_Rwrted() function 350
_Rwrted() function 353
_Rwrread() function 354
_ultoa() function 451

_VBDY macro 4
_wcsicmp() function 492
_wcsnicmp() function 500

A

abnormal program end 60
abort() function 60
abs() function 61
absolute value
abs() function 61
fabs 115
labs 199
access mode 117, 134
acos() function 62
acquire a program device 285
adding data to streams 117
append mode
using fopen() function 134
appending data to files 117
arccosine 62
arcsine 66
arctangent 69
argument list functions 44
asctime_r() function 65
asctime() function 63
asin() function 66
assert.h include file 1
assert() function 68
atan() function 69
atan2() function 69
atexit() function 70
atol() function 71
atol() function 72
atoll() function
strings to long long values 74

B

bessel functions 26, 75
binary files 136
binary search 76
blksize 136
block size 136
bsearch() function 76
btowc() function 78
buffers
assigning 364
comparing 235
copying 236, 239
flushing 121
searching 234
setting characters 240
bufsiz constant 14
bultins
__EXBDY 4
builtins (continued)

__VBDY 4

C

calculating
absolute value 61
absolute value of long integer 199
arccosine 62
arctangent 69
base 10 logarithm 213
calculate the next representable floating-point value 245
cosine 89
error functions 112
exponential function 114
floating-point absolute value 115
floating-point remainder 133
hyperbolic cosine 90
hyperbolic sine 377
hypotenuse 191
logarithm 212
natural logarithm 212
quotient and remainder 111
sine 376
time difference 108, 109
calloc() function 80
cancel handler reason codes 551
case mapping functions 54
catclose() function 81
catgets() function 82
catopen() function 84
ceil() function 86
ceiling function 86
changing
  data area 274
  environment variables 264
  file position 157
  reserved storage block size 291
character
  converting
    to floating-point 74
    to integer 72
    to long integer 74
  reading 123, 174
  setting 240
  ungetting 452
  writing 142, 263
character case mapping
tolower 447
toupper 447
towlower 449
towupper 449
character testing
ASCII value 193
character property 196
isalnum 192
isalpha 192
iscntrl 192
isdigit 192
isgraph 192
islower 192
isspace 192
ispunct 192

character testing (continued)
isspace 192
isupper 192
isxdigit 192
wide alphabetic character 194
wide alphanumeric character 194
wide control character 194
wide decimal-digit character 194
wide hexadecimal digit 194
wide lowercase character 194
wide non-alphanumeric character 194
wide non-space character 194
wide printing character 194
wide uppercase character 194
wide whitespace character 194
character testing functions 51
clear error indicators 87
clearerr 87
clock() function 88
CLOCKS_PER_SEC 89
closing
  file 286
  message catalog 81
  stream 116
comparing
  buffers 235
  strings 388, 391, 393, 408
comparing strings 384, 405
compile regular expression 294
Compute
  Compute the Quantum Exponent 270
  Determine if Quantum Exponents X and Y are the Same 356
concatenating strings 385, 406
conversion functions
  QXXDTOP 275
  QXXDTOZ 276
  QXXITOP 277
  QXXITOZ 277
  QXXPTOD 278
  QXXPTOI 279
  QXXZTOD 281
  QXXZTOI 282
converting
  character case 447
  character string to decimal floating-point 425
  character string to double 422
  character string to long integer 430
  date 414, 502
double to zoned decimal 276
floating-point numbers to integers and fractions 244
floating-point to packed decimal 275
from structure to string 63
from structure to string (restartable version) 65
to a character in the ASCII character set 446
to packed decimal 277
integer to zoned decimal 277
local time 241, 242
monetary value to string 396
multibyte character to a wide character 223
multibyte character to wchar_t 233
multibyte string to a wide character string 227
packed decimal to double 278
packed decimal to integer 279
converting (continued)
  single byte to wide character 78
  string segment to unsigned integer 432
  string to formatted date and time 490
  strings to floating-point values 71
  strings to integer values 72
  strings to long values 74
time 183, 185, 187, 189, 207, 208, 210, 211, 414, 502
  wide character case 449
  wide character string to multibyte string 505
  wide character to a multibyte character 478
  wide character to byte 525
  wide character to long integer 515
  wide character to multibyte character 527
  wide-character string to decimal floating-point 512
  wide-character string to double 510
  wide-character string to unsigned long 520
  zoned decimal to double 281
  zoned decimal to integer 282
copying
  bytes 236, 239
  strings 392, 409
cos() function 89
cosh() function 90
creating
  a temporary file 444, 445
cftime_r() function 100
cftime() function 96
cftime64_r() function 101
cftime64() function 98
ctype functions 192
cctype.h include file
currency functions 26

data conversion
  atof() function 71
  atoi() function 72
  atol() function 74
data items 149
data type compatibility
  CL 559, 562, 563
  COBOL 561
  ILE COBOL 557
  RPG 554, 559
data type limits 5
date
  correcting for local time 207, 208, 210, 211
  functions 26
date and time conversion 414, 502
decimal.h include file
default memory manager 579
deleting
  file 301
  record 289, 301
determine the display width of a wide character 531
determining
  display width of a wide character string 523
  display width of a wide-character string 524
  length of a multibyte character 221
differential equations 26
difftime() function 108
difftime64() function 109
divf() function 111

e
end-of-file
  clearing 303
  macro 14
  resetting error indicator 87
end-of-file indicator 87, 119
ending a program 60, 113
environment
  functions 47
  interaction 47
  retrieving information 202
  table 176
  variables 176, 264
environment variables
  adding 264
  changing 264
  searching 176
erf() function 112
erfc() function 112
errno 2
errno macros 543–545
errno values for Integrated File System 545–547
errno variable 249
errno.h include file 2
error handling
  assert 68
  clearerr 87
  ferror 120
  functions 19
  perror 249
  stream I/O 120
  strerror 396
error indicator 120
error macros, mapping stream I/O exceptions 547, 548
error messages
  printing 249
except.h include file 2
exception class
  listing 553
  mapping 550
EXIT_FAILURE 15, 113
EXIT_SUCCESS 15, 113
exit() function 113
exp() function 114
exponential functions
  exp 114
  frexp 154
  ldexp 200
  log 212
  log10 213
  pow 250
  sqrt 380

f
fabs() function 115
fclose() function 116
fdopen() function 117
feof() function 119
ferror() function 120
fflush() function 121
fgetc() function 123
fgetpos() function 124
fgets() function 126
fgetwc() function 127
fgets() function 129

file
appending to 117
handle 131
include 1
maximum opened 14
name length 14
positioning 303
renaming 302
updating 117

file errors 87
file handling
   remove 301
   rename 302
   tmpnam 445
file name length 14
file names, temporary 14
file positioning 124, 157, 159, 161
FILE type 14
fileno() function 131
float.h include 4
floor() function 132
flushing buffers 121
fmod() function 133
topen, maximum simultaneous files 14
topen() function 134
format data as wide characters 165
formatted I/O 141
tpos_t 15
tprintf() function 141
tputc() function 142
tputs() function 145
fputwc() function 146
fputws() function 148
fread() function 149
free() function 151
freopen() function 153
trexp() function 154
tscanf() function 155
tseek() function 157
tseeko() function 157
tsetpos() function 159
tell() function 161
fwrite() function 162
fwprintf() function 165
fwrite() function 168
fwidth() function 169

G

gamma() function 172
getc() function 174
getchar() function 174
getenv() function 176
gets() function 178

getting
   handle for character mapping 528
   handle for character property classification 529

getting (continued)
   wide character from stdin 181
getwc() function 179
getwchar() function 181
gmtime_r() function 187
gmtime() function 183
gmtime64_r() function 189
gmtime64() function 185

H

diabetes interrupt signals 374
heap memory 578, 583
heap memory manager 578
HUGE_VAL 6
hypot() function 191
hypotenuse 191

I

I/O errors 87
include files
   assert.h 1
ctype.h 1
decimal.h 1
erno.h 2
except.h 2
float.h 4
inintypes.h 5
limits.h 5
locale.h 5
math.h 6
pointer.h 7
reco.h 7
regex.h 10
setjmp.h 11
signal.h 11
stdlib.h 15
string.h 15
time.h 16
xctvt.h 17
xxdtaa.h 17
xxenv.h 18
xxfdbc.h 18

indicators, error 87
initial strings 409
integer
   pseudo-random 284
Integrated File System errno values 545–547
internationalization 5
interrupt signal 374
inttypes.h include 5
invariant character
   hexadecimal representation 565
isalnum() function 192
isalpha() function 192
iscntrl() function 193
isascii() function 193
isntrn() function 192
isdigit() function 192
isgraph() function 192
islower() function 192
isprint() function 192
ispunct() function 192
isspace() function 192
isupper() function 192
iswalnum() function 194
iswcntrl() function 194
iswctype() function 196
iswdigit() function 194
iswgraph() function 194
iswlower() function 194
iswprint() function 194
iswpunct() function 194
iswspace() function 194
iswupper() function 194
iswxdigit() function 194
isxdigit() function 192

library functions (continued)
character testing (continued)
isxdigit 192
conversion
QXXDTOP 275
QXXDTOZ 276
QXXITOP 277
QXXITOZ 277
QXXPTOD 278
QXXPTO1 279
QXXZTOD 281
QXXZTO1 282
strfmon 396
strptime 414
wcsftime 490
wcsptime 502
data areas
QXXCHGDA 274
QXXRTVDA 280
error handling
_GetExcData 176
clearerr 87
raise 283
strerror 396
exponential
exp 114
frexp 154
ldexp 200
log 212
log10 213
pow 250
file handling
fileno 131
remove 301
rename 302
tmpfile 444
tmpnam 445
locale
localeconv 202
nl_langinfo 246
setlocale 366
strxfrm 434
math
acos 62
asin 66
atan 69
atan2 69
bessel 75
ceil 86
cos 89
cosh 90
div 111
erf 112
erfc 112
floor 132
fmod 133
frexp 154
gamma 172
hypot 191
ldiv 201
log 212
log10 213
modf 244
sin 376
library functions (continued)

math (continued)

\texttt{sinh 377}
\texttt{sqrt 380}
\texttt{tan 439}
\texttt{tanh 440}

memory management

\texttt{\_C\_TS\_malloc\_debug 103}
\texttt{\_C\_TS\_malloc\_info 105}
\texttt{calloc 80}
\texttt{free 151}
\texttt{malloc 217}
\texttt{realloc 291}

memory operations

\texttt{memchr 234}
\texttt{memcmp 235}
\texttt{memmove 239}
\texttt{memset 240}
\texttt{wmemchr 532}
\texttt{wmemcmp 533}
\texttt{wmemmove 535}
\texttt{wmemmove 536}
\texttt{wmemset 537}

message catalog

\texttt{catclose 81}
\texttt{catgets 82}
\texttt{catopen 84}

miscellaneous

\texttt{assert 68}
\texttt{getenv 176}
\texttt{longjmp 215}
\texttt{perror 249}
\texttt{rand 284}
\texttt{rand\_r 284}
\texttt{setjmp 365}
\texttt{srand 381}

multibyte

\texttt{\_wcsicmp 492}
\texttt{\_wcsnicmp 500}
\texttt{btowc 78}
\texttt{mblen 219}
\texttt{mbrlen 221}
\texttt{mbtowc 223}
\texttt{mbsinit 226}
\texttt{mbsrtowcs 227}
\texttt{mbstowcs 229}
\texttt{mbtowc 233}
\texttt{towctrans 448}
\texttt{wcrtomb 478}
\texttt{wcsat 483}
\texttt{wchs 484}
\texttt{wcscmp 485}
\texttt{wcscol 487}
\texttt{wcscpy 488}
\texttt{wcscspn 489}
\texttt{wcsl 494}
\texttt{wcslalenconv 495}
\texttt{wcsn 496}
\texttt{wcsncmp 497}
\texttt{wcspn 499}
\texttt{wcspbrk 501}
\texttt{wcspbrk 502}

program

\texttt{abort 60}
\texttt{atexit 70}
\texttt{exit 113}
\texttt{signal 374}

regular expression

\texttt{regcomp 294}
\texttt{regerror 296}
\texttt{regexec 297}
\texttt{regfree 300}

searching

\texttt{bsearch 76}
\texttt{qsort 272}

stream input/output

\texttt{fclose 116}
\texttt{feof 119}
\texttt{ferror 120}
\texttt{fflush 121}
\texttt{fgetc 123}
\texttt{fgetpos 124}
\texttt{fget 126}
\texttt{fgetwc 127}
\texttt{fgetws 129}
\texttt{fprintf 141}
\texttt{fputc 142}
\texttt{ fputs 145}
\texttt{fputwc 146}
\texttt{fputws 148}
\texttt{fread 149}
\texttt{freopen 153}
\texttt{fread 155}
\texttt{fseek 157}
\texttt{fscanf 159}
\texttt{ftime 161}
\texttt{fwide 162}
\texttt{fwprintf 165}
\texttt{fwrite 168}
\texttt{fscanf 169}
\texttt{getc 174}
\texttt{getchar 174}
\texttt{gets 178}
\texttt{getwc 179}
\texttt{getwchar 181}
\texttt{printf 251}
\texttt{putc 263}
\texttt{putchar 263}
\texttt{puts 265}
\texttt{putwc 266}
\texttt{putwchar 268}
\texttt{scanf 358}
\texttt{setbuf 364
library functions (continued)
stream input/output (continued)
setvbuf 372
sprint 379
sscanf 382
swPRINTF 435
swscanf 437
ungetc 452
ungetwc 453
vfprintf 457
vfscanf 458
vfwprintf 460
vfwscanf 462
vprintf 464
vscanf 465
vsnprintf 467
vsprintf 468
vsscanf 469
vswprintf 471
vswscanf 473
vprintf 475
wscanf 476
wprintf 478
string manipulation
strcat 385
strchr 386
strcmp 388
strcoll 391
strcPy 392
strcspn 393
strlen 404
strncmp 408
strncpy 409
strpbrk 413
strrchr 418
strspn 419
strstr 421
strtok 422
strtok 428
strtok 429
strtol 430
strtof 432
strxfrm 434
wcsstr 439
wcsxfrm 440
wprintf 441

library functions (continued)
time (continued)
mktime 241
mktime64 242
strftime 399
wcsftime 490
trigonometric
acos 62
asin 66
atan 69
atan2 69
cos 89
cosh 90
sin 376
sinh 377
tan 439
tanh 440
type conversion
atof 71
atoi 72
atol 74
strtol 430
strtod 422
strtof 432
toascii 446
wcstol 510
wcstol 515
wcstol 520
variable argument handling
va_end 455
va_start 455
vfprintf 457
vfscanf 458
vfwscanf 462
vprintf 464
vscanf 465
vsnprintf 467
vsprintf 468
vsscanf 469
vswscanf 473

locale.h include file 5
localeconv() function

Index 627
log() function 212
log10() function 213
logarithmic functions
log 212
log10 213
logic errors 68
logical record length 136
longjmp() function 215
lrecl 136
malloc() function 217
math functions
abs 61
acos 62
asin 66
atan 69
atan2 69
bessel 75
div 111
erf 112
erfc 112
exp 154
floor 132
fmod 133
frexp 154
gamma 172
hypot 191
labs 199
ldexp 200
ldiv 201
log 212
log10 213
modf 244
pow 250
sin 376
sinh 377
sqrt 380
tan 439
tanh 440
math.h include file 6
mathematical functions 20
maximum
file name 14
opened files 14
temporary file name 6
MB_CUR_MAX 15
mblen() function 219
mbrlen() function 221
mbtowc() function 223
mbsinit() function 226
mbsrtowcs() function 227
mbstowcs() function 229
mbtowc() function 233
memchr() function 234
memcmp() function 235
memcpy() function 236
memmove() function 239
memory allocation
_C_TS_malloc_debug 103
_C_TS_malloc_info 105
memory allocation (continued)
calloc 80
free 151
malloc 217
realloc 291
memory management
_C_TS_malloc_debug 103
_C_TS_malloc_info 105
calloc 80
free 151
malloc 217
realloc 291
memory object functions 46
memory operations
memcmp 235
memmove 239
memset 240
memchr 234
memcmp 235
memmove 239
memset 240
message problems 587
miscellaneous functions
assert 68
getenv 176
longjmp 215
perror 249
putenv 264
rand 284
setjmp 365
srand 381
mktime() function 241
mktime64() function 242
modf() function 244
monetary functions 26
monetary.h include file 6
multibyte functions
_wcsicmp 492
_wcsnicmp 500
btowc 78
mblen 219
mbstrlen 221
mbtowc 223
mbsinit 226
mbsrtowcs 227
mbstowcs 229
mbtowc 233
towctrans 448
wctomb 478
wcschr 483
wcschr 484
wchar 485
wcsx 487
wcsx 488
wcsspn 489
wcsx 492
wcsx 494
wcsx 495
wcsx 496

628 IBM i: ILE C/C++ Runtime Library Functions
multibyte functions (continued)
  wcsncmp 497
  wcscpy 499
  wcsncmp 500
  wcspbrk 501
  wcsrchr 504
  wcsrtombs 505
  wcsspn 507
  wcstombs 517
  wcscs 522
  wcswidth 523
  wcsxfrm 524
  wcscat 525
  wcscopy 527
  wctrans 528
  wctype 529
  wcswcs 522
  wcswidth 523
  wcsxfrm 524
  wctob 525
  wctomb 527
  wctrans 528
  wctype 529
  wcwidth 531

N
NDEBUG 1, 68
nextafter() function 245
nextafterl() function 245
nexttoward() function 245
nexttowardl() function 245
nl_langinfo() function 246
nltypes.h include 7
nonlocal goto 215, 365
NULL pointer 12, 14, 15

O
offsetof macro 12
opening
  message catalog 84

P
passing
  constants 563
  variables 564
  perror() function 249
  pointer.h include 7
  pow() function 250
pragma preprocessor directives
  default memory manager 579
  environment variables 586
  heap memory 578, 583
  message problems 587
  quick pool memory manager 581
preprocessor directives
  default memory manager 579
  environment variables 586
  heap memory 578, 583
  message problems 587
  quick pool memory manager 581
printf() function 251
printing
  error messages 249
process control
  signal 374
program termination
  abort 60
program termination (continued)
  atexit 70
  exit 113
pseudo-random integers 284
pseudorandom number functions
  rand 284
  rand_r 284
  srand 381
ptrdiff_t 11
pushing characters 452
putc() function 263
putchar() function 263
putenv() function 264
puts() function 265
putwc() function 266
putwchar() function 268

Q
qsort() function 272
quantexpd128() function 270
quantexpd32() function 270
quantexpd64() function 270
quantized128() function 271
quantized32() function 271
quantized64() function 271
Quantum Exponent
  Set the Quantum Exponent of X to the Quantum
  Exponent of Y 271
quick sort 272
QXXCHGD() function 274
QXXDTOP() function 275
QXXDTOZ() function 276
QXXITOP() function 277
QXXIT0Z() function 277
QXXPTOD() function 278
QXXPTOI() function 279
QXXRTVDA() function 280
QXXZTOD() function 281
QXXZTOI() function 282

R
raise() function 283
RAND_MAX 15
rand_r() function 284
rand() function 284
random access 157, 161
random number generator 284, 381
read operations
  character from stdin 174
  character from stream 174
  data items from stream 149
  formatted 155, 358, 382
  line from stdin 178
  line from stream 126
  reading a character 123
  scanning 155
reading
  character 174
  data 358
  data from stream using wide character 169
  data using wide-character format string 539

Index 629
reading (continued)
formatted data 155
items 149
line 178
messages 82
stream 126
wide character from stream 127, 179
wide-character string from stream 129
realloc() function 291
reallocation 291
recfm 136
recio.h include file 7
record format 136
record input/output
_Racquire 285
_Rclose 286
_Rcommit 287
_Rdelete 289
_Rdevatr 290
_Rread 304
_Rred 305
_Rfrmt 306
_Rindara 308
_Riofbk 310
_Rlocate 312
_Ropen 315
_Ropnfbk 319
_Rpgmdev 320
_Rread 321
_Rreadf 323
_Rreadindv 325
_Rreadk 328
_Rread3 331
_Rrdevatr 332
_Rreadnc 334
_Rreadp 337
_Rreads 339
_Rrelease 341
_Rrsclck 342
_Rrnlck 343
_Rupdate 345
_Rrupt 346
_Rrwrite 348
_Rrwriterd 350
_Rrwrites 353
_record program ending 70
redirection 153
regcomp() function 294
regerror() function 296
regex.h include file 10
regexec() function 297
regfree() function 300
remove() function 301
rename() function 302
reopening streams 153
reserving storage
_C_TS_malloc_debug 103
_C_TS_malloc_info 105
malloc 217
realloc 291
retrieve data area 280
retrieve locale information 246
rewind() function 303

S
samequantum128() function 356
samequantum32() function 356
samequantum64() function 356
scanf() function 358
searching
bsearch function 76
evironment variables 176
strings 386, 413, 419
strings for tokens 428, 429
searching and sorting functions 20
seed 381
send signal 283
separate floating-point value 154
setbuf() function 364
setjmp.h include file 11
setjmp() function 365
setlocale() function 366
setting
bytes to value 240
setvbuf() function 372
signal handling 548
signal.h include file 11
signal() function 374
sin() function 376
sine 376
sinh() function 377
size_t 11
snprintf() function 378
sorting
quick sort 272
sprintf() function 379
sqrt() function 380
srand() function 381
sscanf() function 382
standard types
FILE 14
stdarg.h include file 11
stdbool.h include file 12
stddef.h include file 11
stdint.h include file 12
stdio.h include file 13
stdlib.h include file 15
stopping
program 60
storage allocation 80
strcascmp() function 384
strcat() function 385
strchr() function 386
strcmp() function 388
strcspn() function 389
strcoll() function 391
strncpy() function 392
strerror() function 393
strdup() function 395
stream I/O functions 38
stream input/output
fclose 116
feof 119
ferror 120
fflush 121
fgetc 123
fgetls 126
stream input/output (continued)
  fopen 134
  fprintf 141
  fputc 142
  fputs 145
  fputwc 146
  fputws 148
  fread 149
  freopen 153
  fscanf 155
  fseek 157
  ftell 161
  fwrite 168
  ungetc 452
  va_arg 455
  va_end 455
  va_start 455
  vfprintf 457
  vfscanf 458
  vfwprintf 460
  vfwscanf 462
  vprintf 464
  vsprintf 468
  vsscanf 470
  vswprintf 471
  vswscanf 473
  vscanf 465
  vsnprintf 467
  vscanf 468
  wcsstr 509
  wcswrchr 512
  wcsstr 514
  wcsstr() function 539
  wprintf 538
  wscanf 539
  strlen() function 404
  strncasecmp() function 405
  strncat() function 406
  strncmp() function 408
  strncpy() function 409
  strnicmp() function 411
  strnset() function 412
  strpbrk() function 413
  strstr() function 421
  strtok() function 428
  strtok_r() function 429
  strtol() function 430
  strxfrm() function 434
  strpbrk() function 413
  strptime() function 414

stream orientation 162

streams
  access mode 153
  appending 134, 153
  binary mode 153
  buffering 364
  changing current file position 157, 161
  changing file position 303
  formatted I/O 155, 251, 358, 379, 382
  opening 134
  reading characters 123, 174
  reading data items 149
  reading lines 126, 178
  reopening 153

strings
  comparing 393, 408
  concatenating 385
  converting
    to floating-point 74
    to integer 72
    to long integer 74
  copying 392
  ignoring case 388, 391, 393
  initializing 409
  length of 404
  reading 126
  searching 386, 413, 419
  searching for tokens 428, 429
  strstr() function 421
  writing 145

streams (continued)
  rewinding 303
  text mode 153
  translation mode 153
  ungetting characters 452
  updating 134, 153
  writing characters 142, 263
  writing data items 168
  writing lines 265
  writing strings 145

strings (continued)
  strerror() function 396
  strftime() function 396
  strfmon() function 396
  strtime() function 399
  stricmp() function 403
  string manipulation
    strcasecmp 384
    strcat 385
    strchr 386
    strcmp 388
    strcoll 391
    strcspn 392
    strcspn 393
    strlen 404
    strncasecmp 405
    strncat 406
    strncmp 408
    strncpy 409
    strnicmp 411
    strnset 412
    strpbrk 413
    strstr 414
    strtok 428
    strtok_r 429
    strtol 430
    strxfrm 434
    wcsstr 509
    wcswrchr 512
    wcstok 514
    string.h include file 15
strchr() function 418
strset() function 412
strspn() function 419
strstr() function 421
strtod() function 422
strtod128() function 425
strtod32() function 425
strtod64() function 425
strtok_r() function 429
strtok() function 428
strtol() function 430
strtol() function
    character string to long long integer 430
strtoull() function 432
strtooll() function
    character string to unsigned long long integer 432
strxfrm() function 434
swprintf() function 435
swscanf() function 437
system() function 438

T

tan() function 439
tangent 439
tanh() function 440
testing
    ASCII value 193
    character property 196
    isalnum 192
    isalpha 192
    iscntrl 192
    isdigit 192
    isgraph 192
    islower 192
    isprint 192
    ispunct 192
    isspace 192
    isupper 192
    isxdigit 192
    state object for initial state 226
    wide alphabetic character 194
    wide alphanumeric character 194
    wide control character 194
    wide decimal-digit character 194
    wide hexadecmal digit 194
    wide lowercase character 194
    wide non-alphanumeric character 194
    wide non-space character 194
    wide printing character 194
    wide uppercase character 194
    wide whitespace character 194
testing state object for initial state 226
time
    asctime 63
    asctime_r 65
    converting from structure to string 63
    converting from structure to string (restartable version)
    65
correcting for local time 207, 208, 210, 211
time 96
time_r 100
time64 98
time64_r 101

time (continued)
difftime 108
difftime64 109
function 207, 208, 210, 211
functions 26
gmtime 183
gmtime_r 187
gmtime64 185
gmtime64_r 189
localtime 207
localtime_r 210
localtime64 208
localtime64_r 211
mktime 241
mktime64 242
strftime 399
time 441
time64 443
time.h include file 16
time() function 441
time64() function 443
tm structure 183, 185, 187, 189
TMP_MAX 445
tmpfile() function
    names 14
    number of 14
tmpnam() function
    file names 14
toascii() function 446
tokens
    strtok 428
    strtok_r 429
tokenize string 428
tolower() function 447
toupper() function 447
towctrans() function 448
towlower() function 449
towupper() function 449
trigonometric functions
    acos 62
    asin 66
    atan 69
    atan2 69
    cos 89
    cosh 90
    sin 376
    sinh 377
tan 439
tanh 440
type conversion
    atof 71
    atoi 72
    atol 74
    strtod 422
    strtol 430
    strtoull 432
toascii 446
    wcstod 510
    wcstol 515
    wcstoul 520

U

ungetc() function 452
ungetwc() function 453
updating files 117

V
va_arg() function 455
va_copy() function 455
va_end() function 455
va_start() function 455
variable argument functions 44
verify condition 68
vfprintf() function 457
vfscanf() function 458
vfswprintf() function 460
vfswscanf() function 462
vprintf() function 464
vscanf() function 465
vsscanf() function 467
vswprintf() function 471
vswscanf() function 473
wprintf() function 475
wscanf() function 476

W
wchar.h include file 16
wcrtomb() function 478
wcscat() function 483
wcschr() function 484
wcsncmp() function 485
wcsncpy() function 487
wcsncmp() function 488
wcscpy() function 488
wcscspn() function 489
wcsftime() function 490
wcslen() function 494
wcslocaleconv() function 495
wcsncat() function 496
wcsncmp() function 497
wcsncpy() function 499
wcspbrk() function 501
wcsftime() function 502
wcschr() function 504
wcsrtombs() function 505
wcsspn() function 507
wcsstr() function 509
wcstod() function 510
wcstod128() function 512
wcstod32() function 512
wcstod64() function 512
wcstof() function 510
wcstok() function 514
wcstol() function 515
wcstold() function 510
wcstoll() function
  wide character to long long integer 515
wcstombs() function 517
wcstoul() function 520
wcstoull() function
  wide-character string to unsigned long long 520
wcswcs() function 522
wcswcs() function 523
wctob() function 525
wcscmp() function 527
wctrans() function 528
wctype.h include file 17
wctype() function 529
wcwidth() function 531
wide character string functions 55
wmemchr() function 532
wmemcmp() function 533
wmemcpy() function 535
wmemmove() function 536
wmemset() function 537
wprintf() function 538
write operations
  character to stdout 142, 263
  character to stream 142, 263, 452
  data items from stream 168
  formatted 141, 251, 379
  line to stream 265
  printing 168
  string to stream 145
writing
  character 142, 263
  data items from stream 168
  formatted data to a stream 141
  string 145, 265
  wide character 146, 266, 268
  wide characters to a stream 165
  wide-character string 148
wscanf() function 539

X
xxcvt.h include file 17
xxdtaa.h include file 17
xxenv.h include file 18
xxfdbg.h include file 18