

z/OS



Using REXX and z/OS UNIX System Services

Version 2 Release 1

Note

Before using this information and the product it supports, read the information in "Notices" on page 259.

This edition applies to Version 2 Release 1 of z/OS (5650-ZOS) and to all subsequent releases and modifications until otherwise indicated in new editions.

© **Copyright IBM Corporation 1996, 2013.**

US Government Users Restricted Rights – Use, duplication or disclosure restricted by GSA ADP Schedule Contract with IBM Corp.

Contents

Figures vii

Tables ix

About this document xi

Who should read Using REXX and z/OS UNIX xi
System Services xi
Finding more information about REXX xi
How to read syntax diagrams xi
 Symbols. xi
 Syntax items xii
 Syntax examples xii
z/OS information xiii

z/OS Version 2 Release 1 summary of changes xv

How to send your comments to IBM xvii
If you have a technical problem xvii

Chapter 1. Using TSO/E REXX for z/OS UNIX processing 1

Host command environments for z/OS UNIX processing 1
The SYSCALL environment 2
 Running a REXX program from TSO/E or MVS batch 2
 Establishing the SYSCALL environment 2
 Ending the SYSCALL environment 3
 Establishing and deleting the signal interface routine 3
The SH environment. 4
 Running a REXX program from the z/OS shells or from a program 4
 Using external functions and subroutines. 5
The TSO command environment. 5
 Command input 6
 Command output. 6
 Return codes 6
 Examples 6
Variable scope 7
Writing setuid and setgid REXX programs 7
Input and output for z/OS UNIX processing 7
 Using standard input, output, and error (file descriptors 0, 1, and 2) 8
 Using SYSCALL commands 8
 Using EXECIO. 8
 Exit status from a REXX program 9
Tokens returned from the PARSE SOURCE instruction 9
 Running from a z/OS shell or from a program 9
 Running from TSO/E or batch 10
Using the REXX signal services. 10
Using immediate commands. 11

Moving a REXX program from TSO/E to a z/OS shell 11
 Using argv and environment variables 12
Customizing the z/OS UNIX REXX environment. 13
 Performance in the SYSCALL environment 13
Authorization 14

Chapter 2. z/OS UNIX REXX programming services 15

Creating a z/OS UNIX REXX environment from an application 15
Running the REXX program. 16
Example: C/370 program. 17

Chapter 3. The syscall commands 19

Specifying a syscall command 19
 Specifying numerics 20
 Specifying strings 20
Using predefined variables 21
Return values. 21
 Returned from the SYSCALL environment 21
 Returned from the SH environment 22
Syscall command descriptions 22
 access 22
 aclddelete 23
 aclddeleteentry. 24
 aclfree 25
 aclget 25
 aclgetentry 26
 aclinit 27
 aclset 27
 aclupdateentry 28
 alarm 29
 catclose. 30
 catgets 31
 catopen. 31
 cert 32
 chattr 33
 chaudit 36
 chdir 37
 chmod 37
 chown 39
 close. 40
 closedir. 40
 creat. 41
 dup 42
 dup2 42
 exec 43
 extlink 43
 fchattr 43
 fchaudit 46
 fchmod. 47
 fchown. 48
 f_closfd. 49
 f_control_cvt 50

fcntl	52	pfctl	107
f_dupfd	52	pipe	107
f_dupfd2	52	pt3270	108
f_getfd	53	quiesce	109
f_getfl	53	rddir	110
f_getlk	54	read	111
fork	56	readdir	113
forkexecm	56	readfile	114
fpathconf	57	readlink	114
f_setfd	59	realpath	115
f_setfl	59	rename	115
f_setlk	60	rewinddir	116
f_setlkw	62	rmdir	117
f_settag	64	setegid	118
fstat	65	seteuid	118
fstatvfs	67	setgid	119
fsync	68	setgrent	120
ftrunc	69	setgroups	120
getcwd	69	setpgid	121
getegid	70	setpwent	121
geteuid	70	setregid	122
getgid	70	setreuid	122
getgrent	71	setrlimit	123
getgrgid	71	setsid	125
getgrnam	72	setuid	125
getgroups	73	shmat	127
getgroupsbyname	74	shmdt	127
getlogin	74	shmget	127
getment	75	shmldestroy	128
getmntent	76	shmldinit	129
getpgrp	79	shmldobtain	129
getpid	79	shmldrelease	130
getppid	79	shmmid	130
getpsent	80	shmset	131
getpwent	82	shmstat	131
getpwnam	83	sigaction	132
getpwuid	84	sigpending	134
getrlimit	84	sigprocmask	134
getuid	85	sigsuspend	135
gmtime	85	sleep	136
ioctl	86	spawn	137
isatty	87	spawnp	140
kill	87	stat	140
lchown	89	statfs	142
link	90	statvfs	143
lseek	91	strerror	144
lstat	92	symlink	145
mkdir	93	sysconf	146
mkfifo	93	time	147
mknod	94	times	147
mount	95	trunc	148
msgget	98	ttyname	149
msgrcv	99	umask	149
msgrmid	100	uname	150
msgset	100	unlink	151
msgsnd	101	unmount	152
msgstat	102	unquiesce	153
open	102	utime	153
opendir	104	wait	154
pathconf	105	waitpid	155
pause	106	write	157

writefile	159
Chapter 4. Examples: Using syscall commands	161
Read the root directory into a stem and print it	161
Open, write, and close a file	161
Open a file, read, and close it	161
List all users and groups	162
Display the working directory and list a specified directory	162
Parse arguments passed to a REXX program: the getopts function	162
Count newlines, words, and bytes	164
Obtain information about the mounted file system	165
Mount a file system	165
Unmount a file system	167
Run a shell command and read its output into a stem	169
Print the group member names	170
Obtain information about a user	170
Set up a signal to enforce a time limit for a program	171
List the ACL entries for a file	172
Chapter 5. z/OS UNIX REXX functions	173
REXX I/O functions	173
Opening a stream implicitly	173
Opening a stream explicitly	173
bpxwunix()	174
charin()	176
charout()	177
chars()	178
chmod()	178
convd2e()	179
directory()	180
environment()	180
exists()	181
getpass()	182
linein()	182
lineout()	183
lines()	184
outtrap()	184
procinfo()	185
rexxopt()	189
sleep()	190
stream()	190
submit()	194
syscalls()	194
Chapter 6. BPXWDYN: a text interface to dynamic allocation and dynamic output	197
Calling conventions	197
REXX external function parameter list	197
Conventional MVS variable-length parameter string	197
Conventional MVS parameter list	198
Null-terminated parameter string	199
Request types	200
Keywords	200

BPXWDYN return codes.	201
Key errors	201
Error codes for dynamic allocation	201
Error codes for dynamic output	202
Message processing	202
Requesting dynamic allocation	202
Requesting dynamic concatenation	207
Requesting dynamic unallocation.	207
Requesting allocation information	208
Requesting dynamic output	212
Freeing an output descriptor	214
Examples: Calling BPXWDYN from a REXX program	214
Example: calling BPXWDYN from C.	214

Chapter 7. Virtual file system (VFS) server syscall commands	215
Security	215
Tokens	215
v_close	215
v_create	216
v_fstatfs	217
v_get	218
v_getattr	219
v_link	219
v_lockctl	220
v_lookup	224
v_open	225
v_mkdir	226
v_read	227
v_readdir	228
v_readlink	229
v_reg	229
v_rel	230
v_remove	231
v_rename	232
v_rmdir	233
v_rpn	233
v_setattr	234
v_setattro	235
v_symlink	236
v_write	237
Chapter 8. Examples: Using virtual file system syscall commands.	239
List the files in a directory	239
Remove a file or empty directory.	239

Appendix A. REXX predefined variables	241
--	------------

Appendix B. Setting permissions for files and directories	253
Position 1	253
Positions 2, 3, and 4	253
Example: using BITOR and BITAND to set mode bits	254

Appendix C. Accessibility 255
 Accessibility features 255
 Using assistive technologies 255
 Keyboard navigation of the user interface 255
 Dotted decimal syntax diagrams 255

Notices 259
 Policy for unsupported hardware. 260

Minimum supported hardware 261
 Trademarks 261
 Acknowledgments. 261

Index 263

Figures

- | | | | | | |
|----|--|-----|----|--|-----|
| 1. | Example of parameter list | 198 | 4. | Three-digit permissions specified in octal
format | 254 |
| 2. | Example of a conventional MVS parameter
list | 198 | | | |
| 3. | Example of a null-terminated parameter
string | 200 | | | |

Tables

1. Syntax examples	xii	8. Keys used to specify which allocation request information is to be returned for	209
2. REXX statements for defining signal sets	10	9. Additional keys used for dynamic information retrieval	209
3. Common keys used for dynamic allocation	203	10. Keys used for dynamic output	212
4. Additional keys used for dynamic allocation	205	11. Key used to free an output descriptor	214
5. Keys used for dynamic concatenation	207	12. List of predefined variables	241
6. Common keys used for dynamic unallocation	207		
7. Additional keys used for dynamic unallocation	208		

About this document

This document presents the information you need to write REXX programs that access z/OS UNIX System Services (z/OS UNIX). It describes the features and usage requirements for the z/OS UNIX REXX extensions, or *syscall commands*, which are interfaces between the z/OS operating system and the functions specified in the POSIX.1 standard (ISO/IEC 9945-1:1990[E] IEEE Std 1003.1-1990: First edition 1990-12-07; Information technology—Portable Operating System Interface [POSIX] Part 1; System Application Program Interface [API] [C Language]). These functions are used by z/OS UNIX. This document also describes syscall commands that are not related to the standards.

Who should read Using REXX and z/OS UNIX System Services

This document is for programmers who are already familiar with the REXX language and experienced with the workings of TSO/E and z/OS UNIX. It describes how to include in a REXX program syscall commands that access z/OS UNIX services.

BPX messages from the REXX processor are documented in *z/OS MVS System Messages, Vol 3 (ASB-BPX)*.

Finding more information about REXX

The following publication is useful: *The REXX Language: A Practical Approach to Programming*, by Michael Cowlshaw (Englewood Cliffs, NJ: Prentice-Hall, 1990).

How to read syntax diagrams

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

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

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

Symbols

The following symbols may be displayed in syntax diagrams:

Symbol

Definition

- ▶▶— Indicates the beginning of the syntax diagram.
- ▶ Indicates that the syntax diagram is continued to the next line.
- ▶— Indicates that the syntax is continued from the previous line.
- ▶◀ Indicates the end of the syntax diagram.

Syntax items

Syntax diagrams contain many different items. Syntax items include:

- Keywords - a command name or any other literal information.
- Variables - variables are italicized, appear in lowercase, and represent the name of values you can supply.
- Delimiters - delimiters indicate the start or end of keywords, variables, or operators. For example, a left parenthesis is a delimiter.
- Operators - operators include add (+), subtract (-), multiply (*), divide (/), equal (=), and other mathematical operations that may need to be performed.
- Fragment references - a part of a syntax diagram, separated from the diagram to show greater detail.
- Separators - a separator separates keywords, variables or operators. For example, a comma (,) is a separator.

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

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

Item type

Definition

Required

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

Optional

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

Default

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

Syntax examples

The following table provides syntax examples.

Table 1. Syntax examples


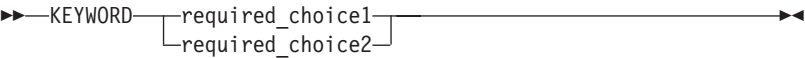
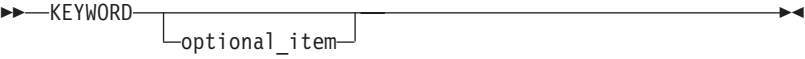
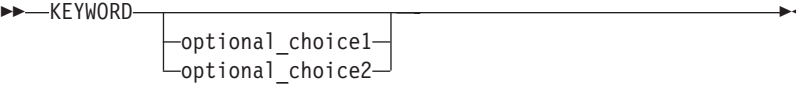
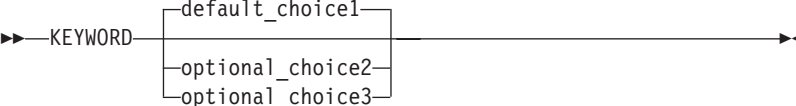

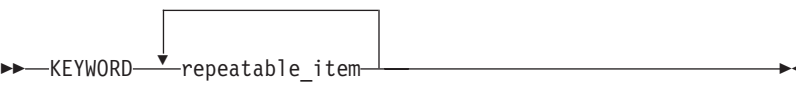
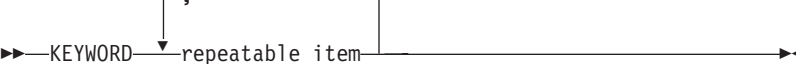

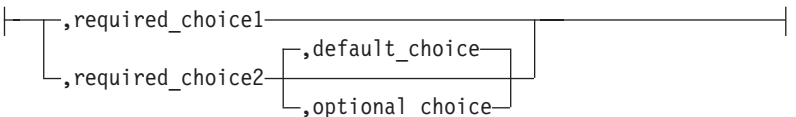
Item	Syntax example
Required item.	
Required items appear on the main path of the horizontal line. You must specify these items.	
Required choice.	
A required choice (two or more items) appears in a vertical stack on the main path of the horizontal line. You must choose one of the items in the stack.	
Optional item.	
Optional items appear below the main path of the horizontal line.	

Table 1. Syntax examples (continued)

Item	Syntax example
Optional choice.	
<p>An optional choice (two or more items) appears in a vertical stack below the main path of the horizontal line. You may choose one of the items in the stack.</p>	
Default.	
<p>Default items appear above the main path of the horizontal line. The remaining items (required or optional) appear on (required) or below (optional) the main path of the horizontal line. The following example displays a default with optional items.</p>	
Variable.	
<p>Variables appear in lowercase italics. They represent names or values.</p>	
Repeatable item.	
<p>An arrow returning to the left above the main path of the horizontal line indicates an item that can be repeated.</p>	
<p>A character within the arrow means you must separate repeated items with that character.</p>	
<p>An arrow returning to the left above a group of repeatable items indicates that one of the items can be selected, or a single item can be repeated.</p>	
Fragment.	
<p>The fragment symbol indicates that a labelled group is described below the main syntax diagram. Syntax is occasionally broken into fragments if the inclusion of the fragment would overly complicate the main syntax diagram.</p>	<p>fragment:</p> 

z/OS information

This information explains how z/OS references information in other documents and on the web.

When possible, this information uses cross document links that go directly to the topic in reference using shortened versions of the document title. For complete titles and order numbers of the documents for all products that are part of z/OS®, see *z/OS Information Roadmap*.

To find the complete z/OS library, including the z/OS Information Center, see z/OS Internet Library (<http://www.ibm.com/systems/z/os/zos/bkserv/>).

z/OS Version 2 Release 1 summary of changes

See the following publications for all enhancements to z/OS Version 2 Release 1 (V2R1):

- *z/OS Migration*
- *z/OS Planning for Installation*
- *z/OS Summary of Message and Interface Changes*
- *z/OS Introduction and Release Guide*

How to send your comments to IBM

We appreciate your input on this publication. Feel free to comment on the clarity, accuracy, and completeness of the information or provide any other feedback that you have.

Use one of the following methods to send your comments:

1. Send an email to mhvrcfs@us.ibm.com.
2. Send an email from the "Contact us" web page for z/OS (<http://www.ibm.com/systems/z/os/zos/webqs.html>).
3. Mail the comments to the following address:
IBM Corporation
Attention: MHVRCFS Reader Comments
Department H6MA, Building 707
2455 South Road
Poughkeepsie, NY 12601-5400
US
4. Fax the comments to us, as follows:
From the United States and Canada: 1+845+432-9405
From all other countries: Your international access code +1+845+432-9405

Include the following information:

- Your name and address.
- Your email address.
- Your telephone or fax number.
- The publication title and order number:
z/OS V2R1.0 Using REXX and z/OS UNIX System Services
SA23-2283-00
- The topic and page number that is related to your comment.
- The text of your comment.

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

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

If you have a technical problem

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

- Contact your IBM service representative.
- Call IBM technical support.
- Visit the IBM Support Portal at z/OS support page (<http://www.ibm.com/systems/z/support/>).

Chapter 1. Using TSO/E REXX for z/OS UNIX processing

A REXX program is recognized by the word *REXX* (not case-sensitive) as the first word on the first line and within a REXX comment. For example, the following is a simple REXX program:

```
/* rexx */  
say 'hello world'
```

Restriction: Blank spaces cannot precede the opening REXX comment symbol (*/**).

The set of z/OS UNIX extensions to the TSO/E Restructured Extended Executor (REXX) language enable REXX programs to access z/OS UNIX callable services. The z/OS UNIX extensions, called *syscall commands*, have names that correspond to the names of the callable services they invoke—for example, **access**, **chmod**, and **chown**.

You can run an interpreted or compiled REXX program with syscall commands from TSO/E, from MVS™ batch, from the z/OS shells, or from a program. You can run a REXX program with syscall commands only on a system with z/OS UNIX System Services installed. For a complete description of each of the syscall commands, see Chapter 3, “The syscall commands,” on page 19.

The set of z/OS UNIX REXX functions also extend the REXX language on z/OS in the z/OS UNIX environment. There are functions that provide:

- Standard REXX I/O
- Access to some common file services and environment variables

All of the z/OS UNIX functions, except **bpwunix()** and **syscalls()**, must be run in a z/OS UNIX environment. For a complete description of each of the functions, see Chapter 5, “z/OS UNIX REXX functions,” on page 173.

For dynamic allocation and dynamic output, BPXWDYN is a text interface to a subset of the SVC 99 and SVC 109 services that is designed to be called from REXX. For a complete description of BPXWDYN, see Chapter 6, “BPXWDYN: a text interface to dynamic allocation and dynamic output,” on page 197.

Host command environments for z/OS UNIX processing

Host command environments and external function packages that are available in the MVS REXX environment can be used by a REXX program that has z/OS UNIX extensions. These additional host command environments are also available:

SYSCALL

For a REXX program with syscall commands that will be run from TSO/E or MVS batch, you need to initialize the environment by beginning a REXX program with a **syscalls('ON')** call.

SH

For a REXX program with syscall commands that will be run from a z/OS shell or from a program, SH is the initial host environment. The SYSCALL environment is automatically initialized as well, so you do not need to begin the REXX program with a **syscalls('ON')** call. Syscall commands within the REXX program (for example, **chmod**) are interpreted as z/OS shell commands, not as syscall commands.

TSO A REXX program can run TSO/E commands, but you cannot use TSO commands to affect your REXX environment, or have REXX statements or other host command environments affect your TSO process. Commands that are addressed to TSO will be run in a TMP running in a separate address space and process from your REXX program. The TSO process is started when the first TSO command is run, and persists until your REXX program terminates or you run the TSO LOGOFF command.

When a REXX program is run from a z/OS shell or from a program, both the SH and SYSCALL host command environments are available to it. When a REXX program is run from TSO/E or MVS batch, only the SYSCALL environment is available.

For background information about the concept of a host command environment, see *z/OS TSO/E User's Guide*.

The SYSCALL environment

The SYSCALL environment can be used by any REXX program with syscall commands, whether it runs from TSO/E or the z/OS shells (where the environment is automatically initialized).

Running a REXX program from TSO/E or MVS batch

To run a REXX program with syscall commands from TSO/E or MVS batch, use the **syscalls('ON')** function at the beginning of the REXX program. This function:

- Ensures that the SYSCALL command environment (ADDRESS syscall) is established.
- Ensures that the address space is a process; this is known as *dubbing*.
- Initializes the predefined variables in the current REXX variable pool.
- Sets the signal process mask to block all signals that can be blocked. See “Using the REXX signal services” on page 10 for more information about signals.
- Clears the `__argv.` and `__environment.` stems. For this reason, it is not recommended that you use **syscalls('ON')** in a z/OS shell environment.

For REXX programs run from TSO/E or MVS batch, you use the **syscalls()** function to control the SYSCALL host command environment. You control the SYSCALL environment by using:

- **syscalls('ON')** to establish the SYSCALL environment
- **syscalls('OFF')** to end the SYSCALL environment
- **syscalls('SIGON')** to establish the signal interface routine
- **syscalls('SIGOFF')** to delete the signal interface routine

Rule: The words **ON**, **OFF**, **SIGON**, and **SIGOFF** must be in uppercase letters.

Establishing the SYSCALL environment

The **syscalls('ON')** function establishes the SYSCALL environment. It sets up the REXX predefined variables and blocks all signals. The variable `syscall_constants` is set to a space-delimited list of all the other predefined variable names. It can be used to expose the variables in a procedure. For example:

```
procedure expose (syscall_constants)
```

The function sets this return value:

- 0 Successful completion.
- 4 The signal process mask was not set.
- 7 The process was dubbed, but the SYSCALL environment was not established.
- 8 The process could not be dubbed.

The following example shows how you can use the **syscalls('ON')** function at the beginning of a REXX program:

```
if syscalls('ON')>3 then
  do
    say 'Unable to establish the SYSCALL environment'
    return
  end
```

Ending the SYSCALL environment

The **syscalls('OFF')** function ends the connection between the current task and z/OS UNIX and the REXX program continues. This action might also undub the process. It is usually not necessary to make a **syscalls('OFF')** call.

The **syscalls('OFF')** function has one return value:

- 0 Successful completion.

Before using OFF, consider ending connections with child processes including any TSO coprocess that you have started using ADDRESS TSO 'LOGOFF'.

Establishing and deleting the signal interface routine

The **syscalls('SIGON')** function establishes the signal interface routine (SIR). After you establish the SIR, use the **sigaction** syscall command to catch the signals you want to process and the **sigprocmask** syscall command to unblock those signals.

Note: For a REXX program run from a z/OS shell or from a program, the SIR is established by default.

The **syscalls('SIGON')** function has these return values:

- 0 Successful completion.
- 4 The SIR could not be established. The usual cause for this is that another SIR has already been established for the process.

If you are writing a REXX program that runs a program that requires a signal interface routine (for example, a program that uses the C runtime library), you must delete the SIR. The **syscalls('SIGOFF')** function deletes the SIR and uses **sigprocmask()** to reset the signal process mask so that it blocks all signals that can be blocked.

The **syscalls('SIGOFF')** function has two return values:

- 0 Successful completion.
- 4 The SIR could not be deleted. The usual cause for this is that a SIR did not exist for the process.

The SH environment

The SH environment is the default host command environment when a REXX program is run from a z/OS shell or from a program using `exec()`. It is available to a REXX program only in those two situations. In the SH environment, a syscall command runs as a z/OS shell command that has been issued this way:

```
/bin/sh -c shell_command
```

All open file descriptors and environment variables are available to the shell command. Note that built-in shell commands run in the shell process, not your REXX process and cannot alter the REXX environment. For example, address `sh 'cd /'` will not change the current directory of your REXX process.

If you are running the REXX program from a z/OS shell or from a program, the SYSCALL environment is automatically initialized.

See Chapter 4, “Examples: Using syscall commands,” on page 161 for some sample REXX programs that show how REXX and z/OS shell commands can work together—for example, a REXX program that can read output from a z/OS shell command. The **mount** and **unmount** sample programs are shipped in the **/samples** directory as files **mountx** and **unmountx**.

Running a REXX program from the z/OS shells or from a program

You can run a REXX program from the z/OS shells, or you can call it from any program just as you would call an executable program. The REXX program runs as a separate process; it does not run in a TSO/E address space.

A REXX program that is invoked from a z/OS shell or from a program must be a text file or a compiled REXX program that resides in the z/OS UNIX file system. It must have read and execute access permissions. Each line in the text file must be terminated by a newline character and must not exceed 2048 characters. Lines are passed to the REXX interpreter as they are. Sequence numbers are not supported; if you are using the ISPF editor to create the REXX program, be sure to set NUMBER OFF.

If you are working in a z/OS shell environment and use only a filename to invoke the REXX program, the PATH environment variable is used to locate it. For example, **myrexx** uses PATH to locate the program, but **./myrexx** searches only the working directory.

For a REXX program that is run from a z/OS shell or from a program, the SIR is established by default. If the REXX program calls a C program that is running POSIX(ON) or a program that requires an SIR, use the `syscalls('SIGOFF')` function to delete the SIR before calling that program.

CEXEC output from the REXX compiler is supported in the z/OS shell environments. To compile and put CEXEC output into the z/OS UNIX file system, you can use the REXXOEC cataloged procedure; it compiles under TSO/E and then uses the TSO/E OCOPY command to copy the compiled program from a data set to a file in the file hierarchy.

Using external functions and subroutines

You can call external functions and subroutines from a REXX program that resides in the z/OS UNIX file system. The search path for an external routine is similar to that used for a REXX program that is invoked from a z/OS shell or from a program. If only the filename is used on the call to the function or subroutine, the PATH environment variable is used to locate it; otherwise, the function name determines the search. For an executable module, the link pack area (LPA), link list, and STEPLIB can also be searched. The default z/OS environment searches for executable modules first. See “Customizing the z/OS UNIX REXX environment” on page 13.

The search order for modules and execs that are invoked as functions or subroutines is controlled by the FUNCISOFL flag in the REXX parameter module. For a description of that flag, see *z/OS TSO/E REXX Reference*.

The following rule must be observed in naming and calling an external function or subroutine:

- If the name contains special characters or lowercase characters, you must enclose it in quotes— for example:

```
ans='myfunc'(p1,p2)
```

If the name is not quoted, REXX folds the name to uppercase. The function call then fails, because the file is not found.

Executable external functions or subroutines that are written in a language other than interpreted REXX and located in the z/OS UNIX file system are not supported.

The TSO command environment

The TSO command environment (ADDRESS TSO) can be used from a z/OS UNIX REXX environment, and is initialized with:

```
address tso [command]
```

where *command* can be any TSO/E command, CLIST, or REXX exec that can run in a batch TSO TMP.

Commands addressed to TSO are run in a TSO TMP that is running in a separate address space and process from your REXX program. This provides you with the capability to run TSO commands. It does not provide you with the capability to use TSO commands to affect your REXX environment, or to have REXX statements or other host command environments affect your TSO process.

The TSO process is started when the first TSO/E command is run, and persists until your REXX program terminates or you run the TSO LOGOFF command. You can use the **ps** shell command to observe this process as the program **bpxwrtso**. Unexpected termination of the TSO process causes the next TSO command to fail with return code 16. A subsequent command starts a new TSO process.

Specifying environment variable BPXWRFD in a REXX program before the TSO process is started causes the TSO process to inherit open file descriptors 10 thru 99. This inheritance stays in effect as long as the TSO process is active. Removing this inheritance can be accomplished by unsetting BPXWRFD and stopping the TSO

process; for example, by issuing ADDRESS TSO LOGOFF. Subsequent ADDRESS TSO commands will then start with a new TSO process, and file descriptors 10 thru 99 will not be inherited.

Command input

Most native TSO commands, including commands that prompt for missing arguments, use TGET for input. This results in a command error, and the command usually terminates.

For commands that are able to read input, the source of the input is first any data that is currently on your stack, and then any data in your REXX program's standard input stream. Regardless of whether the command processes input, all data on the stack is queued to the TSO command. The stack is empty after any TSO command has been run.

The standard input stream can also be queued as input to the TSO command. For example, if you have a file redirected as input and you run a TSO command before processing that file, some or all of the file can be queued to the TSO command. If input is the terminal, queued input can be queued to the TSO command. This characteristic can be used to interact with some TSO commands.

You can disable command input by using the `rexopt()` function with NOTSOIN specified.

Command output

By default, all command output is directed to your REXX process's standard output stream. You can use the `outtrap()` function to trap command output in variables.

Return codes

The special REXX variable RC usually contains the return code from the TSO command. If the command abends or is not found, or if another error is detected, special return codes are set, and a descriptive message can be written to the standard error stream:

- -3 usually means that the TSO command was not found.
- Other negative numbers are usually abend codes. These should be accompanied by a message containing an abend reason code.
- 16 usually means that a processing error was encountered.

Examples

To run the TSO/E TIME command:

```
address tso 'time'
```

To trap command output and print it:

```
call outtrap out.  
  address tso 'listc'  
  do i=1 to out.0  
    say out.i  
  end
```

To run a REXX exec in TSO/E:

```
address tso  
  "alloc fi(sysexec) da('schoen.rexx') shr"  
  "myexec"
```


This is a functional replacement for the **tsocmd** command:

```
/* rexx */
  address tso arg(1)
  return rc
```

In previous releases, the **tsocmd** command was available for download from the **Tools and Toys** z/OS UNIX Web page at <http://www.ibm.com/systems/z/os/zos/features/unix/bpxa1toy.html>. As of V1R12, **tsocmd** is shipped as part of the base release. The base release version of **tsocmd** takes advantage of this REXX support.

Variable scope

When the REXX program is initialized and the SYSCALL environment is established, the predefined variables are set up. If you call an internal subroutine that uses the PROCEDURE instruction to protect existing variables by making them unknown to that subroutine (or function), the predefined variables also become unknown. If some of the predefined variables are needed, you can either list them on the PROCEDURE EXPOSE instruction or issue another **syscalls('ON')** to reestablish the predefined variables. The predefined variables are automatically set up for external functions and subroutines. For example:

```
subroutine: procedure
junk = syscalls('ON')
parse arg dir
'readdir (dir) dir. stem.'
```

Writing setuid and setgid REXX programs

Setting the set-group-ID-on-execution (setgid) permission means that when a file is run, the calling process's effective GID is set to the file's owner GID; the process seems to be running under the GID of the file's owner, rather than that of the actual invoker.

Setting the set-user-ID-on-execution (setuid) permission means that when a file is run, the calling process's effective UID is set to the file's owner UID; the process seems to be running under the UID of the file's owner, rather than that of the actual invoker.

Like any other setuid or setgid program, a REXX program should not allow the user of the program to get control in your environment. Some examples of instructions that can let a user obtain control are:

- Interactive trace.
- Calling external functions or subroutines: Using a relative pathname can let the user get control if the user sets the PATH variable. External functions and subroutines run under the UID and GID of the main program, regardless of their setuid and setgid mode bits.

Input and output for z/OS UNIX processing

When a REXX program runs, open file descriptors are inherited from the process that issued the **exec()**.

Using standard input, output, and error (file descriptors 0, 1, and 2)

For a REXX program that is run from a z/OS shell or from a program, file descriptors 0, 1, and 2 (conventionally, standard input, standard output, and standard error files) are typically inherited.

Attention: A read or write error on file descriptors 0, 1, or 2 results in a halt interruption if the read or write was from a PARSE EXTERNAL instruction, SAY instruction, or EXECIO.

If the REXX program issues a PARSE EXTERNAL instruction, either explicitly or implicitly (such as from a PARSE PULL instruction with an empty stack), it reads standard input for a single text record. The newline character is stripped from the record before it is returned to the REXX program. Standard input is assumed to be a text file, such as your terminal input.

If the REXX program issues a SAY instruction, the text is directed to standard output, and a newline character is appended to the end of the text. Messages issued by REXX, including error and trace messages, are similarly directed to standard output.

If PARSE EXTERNAL is used after standard input has reached the end of the file, null lines are returned. The end-of-file condition can be detected by EXECIO. For more information, see “Using EXECIO.”

Using SYSCALL commands

The SYSCALL host command environment gives you more direct control over input and output. You can use:

- **readfile** to read an entire text file. See “readfile” on page 114 for more information.
- **writefile** to write an entire text file. See “writefile” on page 159 for more information.
- **read** to read bytes from any kind of file. See “read” on page 111 for more information.
- **write** to write bytes to any kind of file. See “write” on page 157 for more information.

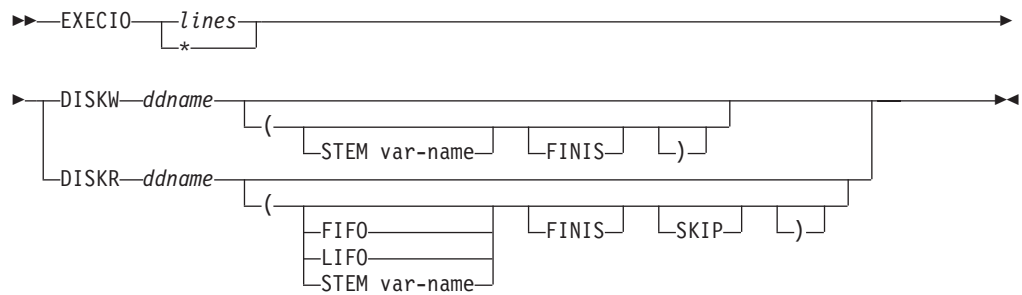
Using EXECIO

EXECIO can be used to read or write zero or more lines in a file. EXECIO differs from readfile and writefile in that it operates on open files. The file must be opened in a mode consistent with the requested read or write operation. Read operations will read from the current file cursor position. Write operations will write from the current file cursor position, or, if the file is opened for append, write operations will append to the end of the file.

Example: To read the open file associated with file descriptor 7 into the stem called **data.**:

```
address mvs 'execio * diskr 7 (stem data. fini'
```

The data can come from and go to the stack or a stem. You can also use it to read or write an entire file. As shown in the following syntax diagram, z/OS UNIX supports all the TSO/E REXX operands except OPEN, DISKRU, and *linenum*.



For the *ddname* operand, you can use the following pseudo-ddnames for processing, when the REXX program is run from a shell or from a program:

- File descriptors 0 to 7
- STDIN, STDOUT, STDERR

Note: When using EXECIO to read a file, the maximum allowable length of a line in the file is 1024 characters, including the newline character. For information about reading blocks of data, see “read” on page 111.

For information about EXECIO, see *z/OS TSO/E REXX Reference*.

Exit status from a REXX program

When a REXX program is run from a z/OS shell or from a program it can return a return code. If the program returns a value in the range 0–255, that value is returned. Otherwise, a value of 255 is returned. If a program is terminated, REXX returns a value of 255.

Tokens returned from the PARSE SOURCE instruction

The tokens that are returned from the PARSE SOURCE instruction depend on where the REXX program is run: from a z/OS shell, from a program, from TSO/E, or from batch.

Running from a z/OS shell or from a program

When a REXX program runs in a z/OS shell environment or is called from a program, the PARSE SOURCE instruction returns nine tokens, in this order:

1. The string TSO
2. The string COMMAND, FUNCTION, or SUBROUTINE, depending on whether the program was invoked as a host command, from a function call in an expression, or with the CALL instruction
3. The path name of the REXX program
4. The string PATH
5. The path name of the REXX program
6. ? (question mark)
7. The name of the initial host command environment in uppercase: SH
8. The name of the address space in uppercase: OMVS
9. An 8-character user token: OpenMVS

To determine whether the REXX program was run from a z/OS shell, use token 8 or 9.

Example

If **myexec** is invoked from the z/OS shells and resides in the working directory, and if PATH is set to **./bin**, a PARSE SOURCE instruction returns the following tokens:

```
TSO COMMAND myexec PATH ./myexec ? SH OMVS OpenMVS
```

Running from TSO/E or batch

If the REXX program runs from TSO/E or MVS batch, the PARSE SOURCE instruction returns the tokens that are described in *z/OS TSO/E REXX Reference*.

Using the REXX signal services

The REXX signal services consist of the following syscall commands:

- alarm**
- kill**
- pause**
- sigaction**
- sigpending**
- sigprocmask**
- sigsuspend**
- sleep**

REXX does not include a service that allows you to attach your own signal catcher. Instead, you have the following options:

- To use the REXX signal catcher as the action for a signal, you can specify the SIG_CAT variable as the signal handler on **sigaction**.

SIG_CAT can terminate various wait conditions without causing the process to end. If a signal arrives when the process is not currently waiting and the signal is not blocked, it might be lost.

There are two primary uses for SIG_CAT: when you are using the **alarm** command, and when you want to avoid unexpected process termination for other unblocked signals.

SIG_CAT causes a signal to interrupt conditions such as waits and blocks, but the application cannot determine which signal was delivered. It is not a traditional signal catcher, as implemented in the C language.

- To set the action to the default action, you can specify SIG_DFL as the signal handler on **sigaction**.
- To set the action to ignore the signal, you can specify SIG_IGN as the signal handler on **sigaction**.

POSIX.1 defines several C functions to manipulate signal sets. REXX does not define these functions; however, you can define each function using a single REXX statement, as shown in Table 2.

Table 2. REXX statements for defining signal sets

C function	Equivalent REXX statement
sigsetempty()	sigsetempty: return copies(0,64) <ul style="list-style-type: none">• Parameters: none• Returns: signal set
sigfillset()	sigfillset: return copies(1,64) <ul style="list-style-type: none">• Parameters: none• Returns: signal set

Table 2. REXX statements for defining signal sets (continued)

C function	Equivalent REXX statement
sigaddset()	sigaddset: return overlay(1,arg(1),arg(2)) <ul style="list-style-type: none"> Parameters: signal set, signal number Returns: signal set
sigdelset()	sigdelset: return overlay(0,arg(1),arg(2)) Parameters: signal set, signal number Returns: signal set
sigismember()	sigismember: return substr(arg(1),arg(2),1) <ul style="list-style-type: none"> Parameters: signal set, signal number Returns: 0 (not member) or 1 (is member)

Using immediate commands

Immediate commands are TSO/E REXX commands, provided with the TSO/E implementation of the language. Immediate commands change characteristics that control the execution of an exec or program.

In response to an interrupt signal, usually <Ctrl-C>, the REXX interrupt handler suspends execution of the REXX program and prompts for an immediate command. The command is specified by number. In the z/OS UNIX REXX environment, the following commands are supported:

Command	Description
1	Continue execution
2	Issue a Halt Interruption
3	Start trace
4	End trace
5	Halt type
6	Resume type

You can use the **rexopt()** function to disable this capability or attach this signal handler to other signals. As with any signal handler, the kernel may defer delivery of the signal depending on what the program is executing at the time.

Note: REXX programs that are run as setuid or setgid programs cannot be interrupted to issue an immediate command.

Moving a REXX program from TSO/E to a z/OS shell

If you write a REXX program to run in TSO/E, it is likely that you will have to alter the REXX program to run it in a z/OS shell environment. Some of the differences between the two environments that you need to consider are:

- You can use the **spawn** syscall command to run z/OS shell commands from the TSO/E environment.
- Using the **syscalls('ON')** function at the beginning of the REXX program is required in TSO/E, but not in a z/OS shell environment. If you use **syscalls('ON')** in a z/OS shell environment, it clears the `__argv.` and `__environment.` stems. For this reason, it is not recommended that you use **syscalls('ON')** in a z/OS shell environment. Using **syscalls('ON')** in a z/OS shell

environment also sets up the REXX predefined variables and blocks all signals. On entry to a REXX program in a z/OS shell environment, the REXX predefined variables are already set.

- In TSO/E, the `syscalls('OFF')` function ends the process, but the REXX program continues to run. In the z/OS shells, the `syscalls('OFF')` function undubs the task and the REXX program continues.
- PARSE SOURCE returns different tokens in TSO/E and in a z/OS shell environment. A REXX program uses the tokens to determine how it was run.
- In TSO/E, the variables `__argv.0` and `__environment.0` are set to zero (0).

See “The TSO command environment” on page 5 for information about running TSO/E commands from a REXX program.

Using argv and environment variables

Environment variables are text strings in the form `VNAME=value`, where `VNAME` is the name of the variable and `value` is its value. The stem variables `__argv` and `__environment` are always set to the original values passed to the first-level REXX program, and they are visible to external REXX functions. You may want to use `PARSE ARG` instead of the `__argv` stem in external REXX programs. As the following two sample programs show, using the `__argv` stem from an external exec returns the same data as it did from the initial exec. In order for an external REXX program to get the arguments a caller is sending it, it must use `arg()` or `PARSE ARG`:

PGM1:

```
/* rexx */
say 'this is the main pgm'
say 'it was passed' __argv.0 'arguments:'
do i = 1 to __argv.0
  say ' Argument' i': "__argv.i"'
end

call 'pgm2' 'arguments', 'to pgm2'
```

PGM2:

```
/* rexx */
say 'This is pgm2'
say 'Using __argv stem, there are' __argv.0 'arguments. They are:'
do i = 1 to __argv.0
  say ' Argument' i': "__argv.i"'
end

say 'Using arg(), there are' arg() 'arguments:'
do i = 1 to arg()
  say ' Argument' i': "arg(i)'"
end
```

Sample execution

```
$ pgm1 'arguments to' 'pgm1'
this is the main pgm
it was passed 3 arguments:
  Argument 1: "pgm1"
  Argument 2: "arguments to"
  Argument 3: "pgm1"
This is pgm2
Using __argv stem, there are 3 arguments. They are:
  Argument 1: "pgm1"
  Argument 2: "arguments to"
```

```
Argument 3: "pgm1"
Using arg(), there are 2 arguments:
Argument 1: "arguments"
Argument 2: "to pgm2"
```

Customizing the z/OS UNIX REXX environment

When a REXX program is run from the z/OS shells or called from a program using `exec()`, the z/OS UNIX REXX environment that is established is created from the module `BPXWRXE.V`. The source for this module is member `BPXWRX01` in `SYS1.SAMPLIB`.

This environment is inherited from the default MVS REXX environment. However, the default handling of error messages from the REXX processor is overridden so that the messages are written to `STDOUT`. This is the same place to which output from the `SAY` instruction and trace information is sent.

You can further customize the sample member to alter the REXX environment for REXX programs running under z/OS UNIX without affecting REXX programs running in the z/OS environment. For detailed information about how to change the default values for initializing an environment, see *z/OS TSO/E REXX Reference*.

Performance in the SYSCALL environment

`syscalls('ON')` ensures that the SYSCALL host command environment is available in your REXX environment. If the call detects that SYSCALL is not available in your environment, it dynamically adds it.

Performance characteristics for dynamically added host commands are not as good as for host commands that are included in the initial environment: Every time a command is directed to the SYSCALL host command environment, the TSO/E REXX support loads the module for the SYSCALL host command.

To avoid this, include the SYSCALL host command in the three default TSO/E environments:

Module name	SYS1.SAMPLIB member name	REXX environment
IRXPARMS	IRXREXX1	MVS
IRXTSPRM	IRXREXX2	TSO
IRXISPRM	IRXREXX3	ISPF

Customizing `IRXISPRM` provides dramatic performance improvement for REXX programs that use `syscall` commands from TSO/E or MVS batch.

Make the following changes to the `SYS1.SAMPLIB` members to add the SYSCALL host command to that default environment:

1. Find the label `SUBCOMTB_TOTAL` and add 1 to its value. For example:
Change `SUBCOMTB_TOTAL DC F'14'` to `SUBCOMTB_TOTAL DC F'15'`.
2. Find the label `SUBCOMTB_USED` and add 1 to its value. For example:
Change `SUBCOMTB_USED DC F'14'` to `SUBCOMTB_USED DC F'15'`.
3. Find the end of the subcommand table, just before the label `PACKTB` or `PACKTB_HEADER`, and add the following lines:

```
SUBCOMTB_NAME_REXXIX    DC  CL8'SYSCALL  '  
SUBCOMTB_ROUTINE_REXXIX DC  CL8'BPXWREXX'  
SUBCOMTB_TOKEN_REXXIX   DC  CL16'  '
```

4. Assemble and link-edit the module and replace the default TSO/E module. These are normally installed in SYS1.LPALIB.

See *z/OS TSO/E REXX Reference* for additional information about customizing the default environments.

Authorization

Users authorized to perform special functions are defined as having *appropriate privileges*, and they are called *superusers*. Appropriate privileges also belong to users with:

- A user ID of zero
- RACF-supported user privileges *trusted* and *privileged*, regardless of their user ID

A user can switch to superuser authority (with an effective UID of 0) if the user is permitted to the BPX.SUPERUSER FACILITY class profile within RACF®. Either the ISPF Shell or the **su** shell command can be used to switch to superuser authority.

This document assumes that your operating system contains Resource Access Control Facility (RACF). You could use an equivalent security product updated to handle z/OS UNIX security.

Chapter 2. z/OS UNIX REXX programming services

An application that supports scripting or macro languages, such as an editor, can use REXX as the macro language. An application that is written in a programming language such as C can create a z/OS UNIX REXX environment and run a REXX program directly. For information on using the TSO/E REXX programming services, such as IRXJCL and IRXEXEC, see *z/OS TSO/E REXX Reference*.

Creating a z/OS UNIX REXX environment from an application

To create a z/OS UNIX REXX environment, fetch and call the module BPXWRBLD from a key 8 problem state program. z/OS linkage for C (that is, standard OS linkage) is required.

The BPXWRBLD module requires the following parameters:

16K area

A 16,000-byte environment area. This must persist for the life of the REXX environment.

arg count

The count of the number of REXX initialization arguments.

arg pointer array

An array of pointers to null-terminated strings, one for each REXX initialization argument. The array and the null-terminated strings must persist for the life of the REXX environment.

env count

The count of the number of environment variables to be exported to the REXX program.

env length pointer array

An array of pointers to fullwords, one for each environment variable. The fullword contains the length of the string that defines the environment variable, including the terminating null. The last element of the array must point to a fullword of 0.

The array and the fullwords must persist for the life of the REXX environment.

env pointer array

An array of pointers to null-terminated strings, one for each environment variable. Each string defines one environment variable. The array and the null-terminated strings must persist for the life of the REXX environment.

The format of the string is

NAME=value

where NAME is the environment variable name, and value is the value for the environment variable followed by a null character.

REXX env addr

The address of a fullword where the address of the newly created REXX environment is returned.

If BPXWRBLD fails to create the environment, it returns the return code it received from the IRXINIT service. BPXWRBLD does not return any other codes.

The parameter list is a standard MVS variable-length parameter list. On entry, the following registers must be set:

Register 1

Address of the parameter list

Register 13

Address of a register save area

Register 14

Return address

Register 15

Entry point address

Register 1 contains the address of the parameter list:

```
-----+
|  --+----> 16K area
-----+
|  --+----> arg count
-----+
|  --+----> arg pointer array
-----+
|  --+----> env count
-----+
|  --+----> env length pointer array
-----+
|  --+----> env pointer array
-----+
|1  --+----> REXX env addr
-----+
```

When constructing arguments to the REXX program that are also passed to BPXWRBLD, keep in mind that:

- The only use of the argument count and argument array is to populate the `__argv` REXX variables. You can set the argument count to 0 if the REXX programs will always get their arguments using PARSE ARG or the **arg(1)** REXX function call. In this case, `__argv.0` is set to 0 when the REXX program is run.
- After the call to BPXWRBLD, do not alter the data that is pointed to by the environment pointer arrays or the arg pointer array.

Signals are not supported in this environment.

Running the REXX program

Before calling a TSO/E REXX service to run the program, ensure that file descriptors 0, 1, and 2 are open. The REXX program will fail if it attempts a PARSE EXTERNAL, EXECIO, or SAY and that function fails.

After the REXX environment is established, the program can call either the IRXJCL or the IRXEXEC TSO/E REXX service to run the REXX program.

- If the IRXJCL service is used, the name of the REXX program is the first word of the IRXJCL parameter string. It is limited to 8 characters.
- If you request the IRXEXEC service to load the program, you must provide the name of the REXX program in the member field of the EXECBLK. Set the DDNAME field to spaces. This also limits the name of the REXX program to 8

characters. Names longer than 8 characters can be supported with additional programming effort. You would need to preload the program and build an INSTBLK instead of an EXECBLK for the IRXEXEC call. If the REXX program is compiled in CEXEC format, load it as a single-record program.

If the name of the REXX program does not contain a slash (/), the PATH environment variable is used to locate the program.

The current REXX environment must be the z/OS UNIX REXX environment. You cannot pass the environment to be used in Register 0.

When the REXX program is being loaded, the IRXEXEC or the IRXJCL service uses one file descriptor to open the file, read it, and close it. If no file descriptor is available because the maximum number of file descriptors are already open, the program cannot be loaded.

Example: C/370 program

This C/370™ program creates a REXX environment and runs a REXX program:

```
#pragma strings(readonly)
#include <stdlib.h>
#include <string.h>
#include <stdio.h>

typedef int EXTF();
#pragma linkage(EXTF,OS)

int main(int argc, char **argv) {
extern char **environ;           /* access environ variables */
EXTF *irxjcl;                   /* pointer to IRXJCL routine */
EXTF *bpxwrbl;                 /* pointer to BPXWRBLD routine */
char *penvb;                   /* addr of REXX environment */
int i,j;                       /* temps */
long rcinit;                   /* return code */
int **environlp;               /* ptr to env length pointers */
int *environl;                 /* ptr to env lengths */
char rxwork[16000];            /* OE MVS env work area */
char *execname="execname";     /* name of exec up to 8 chars */
char *execparm="exec parameter string"; /* parm to exec */
struct s_rxparm {              /* parm to IRXJCL */
    short len;                 /* halfword length of parm */
    char name[8];              /* area to hold exec name */
    char space;                /* one space */
    char text[253];           /* big area for exec parm */
} *rxparm;

/* if stdin or stdout are not open you might want to open file */
/* descriptors 0 and 1 here */

/* if no environ, probably tso or batch - make one */
if (environ==NULL) {
    environ=(char **)malloc(8); /* create one */
    environ[0]="PATH=";         /* set PATH to cwd */
    environ[1]=NULL;           /* env terminator */
};

/* need to build the environment in the same format as expected by */
/* the exec() callable service. See */
/* Assembler Callable Services for UNIX System Services. */

/* the environ array must always end with a NULL element */
for (i=0;environ[i]!=NULL;i++); /* count vars */
environlp=(int **)malloc(i*4+4); /* get array for len ptrs */
environl=(int *)malloc(i*4+4); /* get words for len vals */
```

```

for (j=0;j<i;j++) {
    environlp[j]=&environl[j];      /* point to len      */
    environl[j]=strlen(environ[j])+1; /* set len word      */
};
environlp[j]=NULL;                  /* null entry at end */
environl[j]=0;

/* load routines */
irxjcl=(EXTF *)fetch("IRXJCL  ");
bpxwrblid=(EXTF *)fetch("BPXWRBLD ");

/* build the REXX environment */
rcinit=bpxwrblid(rxwork,
                argc,argv,
                i,environlp,environ,
                &penvb);
if (rcinit!=0) {
    printf("environment create failed rc=%d\n",rcinit);
    return 255;
};

/* if you need to add subcommands or functions to the environment, */
/* or create a new environment inheriting the current one, this is */
/* the place to do it. The user field in the environment is used */
/* by the z/OS UNIX REXX support and must be preserved. */

/* run exec */
rxparm=(struct s_rxparm *)malloc(strlen(execname)+
                                strlen(execparm)+
                                sizeof(struct s_rxparm));
memset(rxparm->name,' ',sizeof(rxparm->name));
memcpy(rxparm->name,execname,strlen(execname));
rxparm->space=' ';
memcpy(rxparm->text,execparm,i(strlen(execparm)));
rxparm->len=sizeof(rxparm->name)+sizeof(rxparm->space)+i;
return irxjcl(rxparm);
}

```

Chapter 3. The syscall commands

In most cases, syscall commands invoke the z/OS UNIX callable service that corresponds to the command verb (the first word of the command). The parameters that follow the command verb are specified in the same order as in POSIX.1 and the z/OS UNIX callable services, where applicable.

For complete information about the processing of a particular syscall command, read about the callable service it invokes, as described in *z/OS UNIX System Services Programming: Assembler Callable Services Reference*.

Specifying a syscall command

You must specify the syscall command parameters in the order indicated in the syscall command description.

syscall command name

The syscall command name is not case-sensitive: you can specify it as uppercase, lowercase, or mixed case.

Parameters

You can specify several types of parameters, but most fall into the following categories:

pathname

The path name is case sensitive, and it is specified as a string. The syscall commands can take a relative or absolute path name as a parameter. The search for a relative path name begins in your working directory:

- If you are running a REXX program from a z/OS shell, your working directory is inherited from your z/OS shell session.
- If you are running a REXX program in TSO/E, your working directory is typically your home directory.

Portable path names can use only the characters in the POSIX portable filename character set:

- Uppercase or lowercase A to Z
- Numbers 0 to 9
- Period (.)
- Underscore (_)
- Hyphen (-)

Do not include any nulls in a path name.

mode The mode is a three- or four-digit number that corresponds to the access permission bits. Each digit must be in the range 0–7, and at least three digits must be specified. See Appendix B, “Setting permissions for files and directories,” on page 253 for more information about permissions.

stem The name of a stem variable. A stem can be used for input, output, or both. A stem is indicated by a . (period) at the end of the variable name.

- The variable name for the first value consists of the name of the stem variable with a 1 appended to it. The number is incremented for each value. For example, *vara.1*, *vara.2*, and *vara.3*.
- The variable name that contains the number of variables returned (excluding 0) consists of the name of the stem variable with a 0 appended to it. For example, *vara.0*

If you omit the period from the end of the variable name, a numeric suffix is appended to the name (for example, *foo* would become *foo0*, *foo1*, and so on).

The name of a stem variable is not case-sensitive.

variable

The name of a REXX variable. The name is not case-sensitive.

Specifying numerics

All numbers are numeric REXX strings. Negative numbers can be preceded by a minus sign (-); others must be unsigned.

The SYSCALL environment supports a 10-digit field. If you are performing arithmetic on a field longer than nine digits, you must set precision to 10. A range of up to $2^{31}-1$ is supported.

Specifying strings

You can specify a string in any of these ways:

String	Example
Any series of characters not containing a space. This example shows a path name with no space in it.	"creat /u/wjs/file 700"
Any series of characters delimited by ' and not containing '. This example shows a path name with a space in it.	"creat 'u/wjs/my file' 700"
Any series of characters delimited by " and not containing ". This example shows a path name with a space in it.	'creat "u/wjs/my file" 700'
A variable name enclosed in parentheses. Strings that contain both the single and double quote characters must be stored in a variable, and you must use the variable name.	file='/u/wjs/my file' "creat (file) 700"

The following example uses a variable enclosed in parentheses to avoid problems with a blank in the filename:

```
file='/u/wjs/my file'
"creat (file) 700"
```

If you incorrectly coded the second line as:

```
"creat /u/wjs/my file 700"
```

it would contain four tokens instead of three.

Using predefined variables

The predefined variables that are available to a REXX program in a z/OS shell environment make symbolic references easier and more consistent—for example, when you are specifying a flag or using a stem variable. Instead of coding a numeric value, you can specify the predefined variable that is used to derive that numeric value.

Appendix A, “REXX predefined variables,” on page 241 lists all the predefined variables alphabetically and shows their numeric value and data type. If a variable is a stem variable, this shows the data type for the stem variable.

You can also use the index of this document as a reference: under the name of each syscall command are grouped the names of the predefined variables associated with it.

Return values

A command can be issued to the SYSCALL environment or the SH environment, and the return values are different in the two environments.

Returned from the SYSCALL environment

When a syscall command completes, the environment can set four reserved variables:

RC A numeric return code from the command execution.

Value Range

Meaning

- | | |
|--------------|---|
| 0 | The command finished successfully. If there is an error code for the requested function, it is returned in RETVAL and ERRNO. |
| >0 | The command finished successfully, but a function-specific warning is indicated. |
| -3 | The command environment has not been called. Probably the syscalls('ON') function did not end successfully, or the current address environment is not SYSCALL. |
| -20 | The command was not recognized, or there was an improper number of parameters specified on the command. |
| -21,-22, ... | The first, second, ... parameter is in error. (The parameter is indicated by the second digit.) |
| <0 | Other negative values might be returned by the REXX language processor. A negative value means that the command did not finish successfully. |

RETVAL

A numeric return value from the callable service. This indicates the success or failure of the service. For most successful calls to services, RETVAL is set to zero; for unsuccessful calls, this value is -1.

However, there are some services (such as `getgrgid` and `getgrnam`) that return zero instead of -1 when the service fails. In addition, some services

return a positive RETVAL to indicate success. For details about a specific service, see *z/OS UNIX System Services Programming: Assembler Callable Services Reference*.

ERRNO

A hexadecimal error number from the callable service. This variable is valid only if the return code (RC) is not negative and RETVAL is -1.

ERRNOJR

A hexadecimal reason code from the callable service. This variable is valid only if the return code (RC) is not negative and RETVAL is -1.

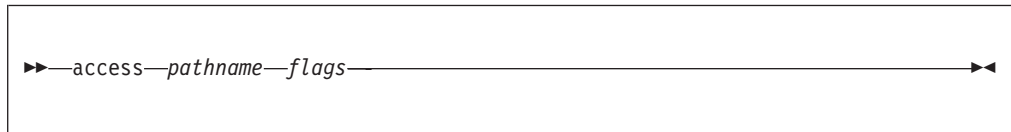
Returned from the SH environment

When a command completes in the SH environment, the return code is set in the variable RC. Unusual situations cause the return code to be set to a negative value:

- 1xxx Terminated by signal xxx
- 2xxx Stopped by signal xxx
- 3xxx Fork failed with error number xxx
- 4xxx Exec failed with error number xxx
- 5xxx Wait failed with error number xxx

Syscall command descriptions

access



Function

access invokes the access callable service to determine if the caller can access a file.

Parameters

pathname

The pathname of the file to be checked for accessibility.

flags

One or more numeric values that indicate the accessibility to be tested. You can specify a numeric value (see Appendix A, “REXX predefined variables,” on page 241) or the predefined variable used to derive the appropriate numeric value. The predefined variables you can specify are:

Variable	Description
F_OK	Test for file existence
R_OK	Test for permission to read
W_OK	Test for permission to write
X_OK	Test for permission to execute

For example, R_OK+W_OK tests for read and write permission.

Usage notes

1. Testing for file permissions is based on the real user ID (UID) and real group ID (GID), not the effective UID or effective GID of the calling process.

2. The caller can test for the existence of a file or for access to the file, but not both.
3. In testing for permission, the caller can test for any combination of read, write, and execute permission. If the caller is testing a combination of permissions, a -1 is returned if any one of the accesses is not permitted.
4. If the caller has appropriate privileges, the access test is successful even if the permission bits are off, except when testing for execute permission. When the caller tests for execute permission, at least one of the execute permission bits must be on for the test to be successful.

Example

To test for permission to execute `grep`:

```
"access '/bin/grep'" x_ok
```

acldelate

```
▶▶ acldelate pathname acltype ◀◀
```

Function

`acldelate` deletes an access control list (ACL) associated with *pathname*.

Parameters

pathname

The pathname of the file or directory the ACL is associated with.

acltype

Indicates the type of ACL. This parameter can have the following values:

Variable	Description
ACL_TYPE_ACCESS (1)	An access ACL
ACL_TYPE_FILEDEFAULT (2)	A file default ACL
ACL_TYPE_DIRDEFAULT (3)	A directory default ACL

Usage notes

1. For regular files, the *acltype* must indicate it is an access ACL that is to be deleted.
2. For a directory, the *acltype* must indicate one of the three types of ACLs (access, file default, or directory default).

Example

To delete the access ACL from the `/tmp` directory, this example assumes the user has set the appropriate stem variable before the call:

```
"acldelate /tmp/ acl." acl_type_access
```

For a complete example that uses several of the ACL services to list ACLs, see “List the ACL entries for a file” on page 172.

aclddelete

For more information about access control lists, see Using access control lists (ACLs) in *z/OS UNIX System Services Planning*.

aclddeleteentry



Function

aclddeleteentry deletes a specific entry in the access control list (ACL) represented by *variable*.

Parameters

variable

The name of a REXX variable that contains a token to access an ACL.

stem

The name of a stem variable that contains an ACL entry. STEM.0 contains a count of the number of variables set in the stem. The following variables may be used to access the stem variables. The number in parentheses is the actual value of the variable:

Variable	Description
ACL_ENTRY_TYPE (1)	Indicates the type of ACL entry: ACL_ENTRY_USER (1) (User ACL) ACL_ENTRY_GROUP (2) (Group ACL)
ACL_ID (2)	The numeric id, uid or gid of the entry
ACL_READ (3)	Indicates read access (1 = yes, 0 = no)
ACL_WRITE (4)	Indicates write access (1 = yes, 0 = no)
ACL_EXECUTE (5)	Indicates execute or search access (1 = yes, 0 = no)
ACL_DELETE (6)	Indicates that the ACL entry is deleted (1 = yes, 0 = no)

Usage notes

1. The entry to delete is identified by the entry type and ID, and is contained in *stem*.
2. If the entry does not exist, the service will return **retval= -1** and **errno=enoent**

Example

To delete the ID in an ACL, this example assumes the user has set the appropriate stem variable before the call:

```
"aclddeleteentry tokenvar acl." acld_id
```

For a complete example that uses several of the ACL services to list ACLs, see "List the ACL entries for a file" on page 172.

For more information about access control lists, see Using access control lists (ACLs) in *z/OS UNIX System Services Planning*.

aclfree

```
▶▶ aclfree variable ◀◀
```

Function

aclfree releases resources associated with the access control list (ACL) represented by *variable* and obtained using the **aclinit** syscall command.

Parameters

variable

The name of a REXX variable that contains a token to access an ACL.

Example

```
"aclfree tokenvar"
```

For a complete example that uses several of the ACL services to list ACLs, see “List the ACL entries for a file” on page 172.

For more information about access control lists, see Using access control lists (ACLs) in *z/OS UNIX System Services Planning*.

aclget

```
▶▶ aclget variable pathname acltype ◀◀
```

Function

aclget reads an access control list (ACL) of the specified type associated with the file identified by *pathname*. The ACL is associated with the specified *variable*, and is accessed and altered using the **acldeleteentry**, **aclgetentry**, and **aclupdateentry** services.

Before using **aclget**, the variable must be initialized by using **aclinit**.

Parameters

variable

The name of a REXX variable that contains a token to access an ACL.

pathname

The pathname of the file or directory the ACL is associated with.

acltype

Indicates the type of ACL. This parameter can have the following values:

Variable	Description
ACL_TYPE_ACCESS (1)	An access ACL

aclget

Variable	Description
ACL_TYPE_FILEDEFAULT (2)	A file default ACL
ACL_TYPE_DIRDEFAULT (3)	A directory default ACL

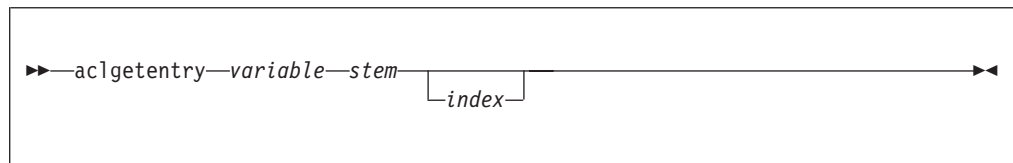
Example

```
"aclget tokenvar /lpp/payroll" acl_type_access
```

For a complete example that uses several of the ACL services to list ACLs, see “List the ACL entries for a file” on page 172.

For more information about access control lists, see Using access control lists (ACLs) in *z/OS UNIX System Services Planning*.

aclgetentry



Function

aclgetentry reads an access control list (ACL) entry from the ACL represented by *variable*.

Parameters

variable

The name of a REXX variable that contains a token to access an ACL.

stem

The name of a stem variable that contains an ACL entry. STEM.0 contains a count of the number of variables set in the stem. The following variables may be used to access the stem variables. The number in parentheses is the actual value of the variable:

Variable	Description
ACL_ENTRY_TYPE (1)	Indicates the type of ACL entry: ACL_ENTRY_USER (1) (User ACL) ACL_ENTRY_GROUP (2) (Group ACL)
ACL_ID (2)	The numeric id, uid or gid of the entry
ACL_READ (3)	Indicates read access (1 = yes, 0 = no)
ACL_WRITE (4)	Indicates write access (1 = yes, 0 = no)
ACL_EXECUTE (5)	Indicates execute or search access (1 = yes, 0 = no)
ACL_DELETE (6)	Indicates that the ACL entry is deleted (1 = yes, 0 = no)

index

Specifies the relative ACL entry to access. The first entry is 1.

Usage notes

1. An entry is identified by *index*, if *index* is specified. Otherwise, the entry is identified by the type and ID specified in *stem*.
2. If the entry does not exist, the service will return **retval= -1** and **errno=enoent**.

Example

To read an ACL entry, this example assumes the user has set the appropriate stem variable before the call:

```
"aclgetentry tokenvar acl."
```

For a complete example that uses several of the ACL services to list ACLs, see “List the ACL entries for a file” on page 172.

For more information about access control lists, see Using access control lists (ACLs) in *z/OS UNIX System Services Planning*.

aclinit

```
▶▶aclinit—variable—————▶▶
```

Function

aclinit obtains resources necessary to process access control lists (ACLs) and associates those resources with *variable*.

Parameters

variable

The name of a REXX variable that contains a token to access an ACL.

Usage notes

1. The *variable* associated with the obtained resources must be passed to other services that operate on an ACL.
2. Any one *variable* can only represent one ACL at a time.
3. **aclfree** must be used to release the resources obtained by **aclinit**.

Example

```
"aclinit tokenvar"
```

For a complete example that uses several of the ACL services to list ACLs, see “List the ACL entries for a file” on page 172.

For more information about access control lists, see Using access control lists (ACLs) in *z/OS UNIX System Services Planning*.

aclset

```
▶▶aclset—variable—pathname—acltype—————▶▶
```

aclset

Function

aclset replaces the access control list (ACL) associated with *pathname* with the ACL represented by *variable*.

Parameters

variable

The name of a REXX variable that contains a token to access an ACL.

pathname

The pathname of the file or directory the ACL is associated with.

acltype

Indicates the type of ACL. This parameter can have the following values:

Variable	Description
ACL_TYPE_ACCESS (1)	An access ACL
ACL_TYPE_FILEDEFAULT (2)	A file default ACL
ACL_TYPE_DIRDEFAULT (3)	A directory default ACL

Example

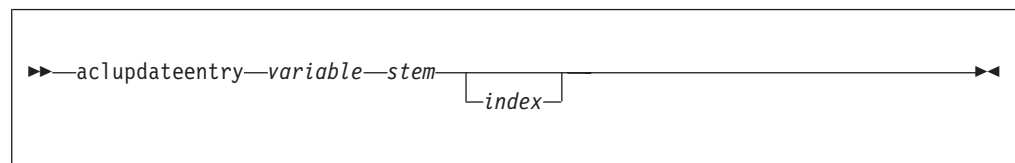
This example replaces the directory default ACL:

```
"aclset tokenvar /u/dept58" acl_type_dirdefault
```

For a complete example that uses several of the ACL services to list ACLs, see "List the ACL entries for a file" on page 172.

For more information about access control lists, see Using access control lists (ACLs) in *z/OS UNIX System Services Planning*.

aclupdateentry



Function

aclupdateentry updates an existing access control list (ACL) entry or creates a new entry if the entry does not already exist in the ACL contained in *variable*.

Parameters

variable

The name of a REXX variable that contains a token to access an ACL.

stem

The name of a stem variable that contains an ACL entry. STEM.0 contains a count of the number of variables set in the stem. The following variables may be used to access the stem variables. The number in parentheses is the actual value of the variable:

Variable	Description
ACL_ENTRY_TYPE (1)	Indicates the type of ACL entry: ACL_ENTRY_USER (1) (User ACL) ACL_ENTRY_GROUP (2) (Group ACL)
ACL_ID (2)	The numeric id, uid or gid of the entry
ACL_READ (3)	Indicates read access (1 = yes, 0 = no)
ACL_WRITE (4)	Indicates write access (1 = yes, 0 = no)
ACL_EXECUTE (5)	Indicates execute or search access (1 = yes, 0 = no)
ACL_DELETE (6)	Indicates that the ACL entry is deleted (1 = yes, 0 = no)

index

Specifies the relative ACL entry to access. The first entry is 1.

Usage notes

1. An entry is identified by either the entry type and ID contained in *stem* or by the relative entry number if *index* is specified. If *index* is 0 or greater than the current number of ACL entries, a new entry is created.
2. **aclupdateentry** can update a deleted ACL entry, mark an entry as deleted, mark the entry as not deleted, or add a new entry that is also marked as deleted by appropriate setting of *stem.acl_delete*. Also, duplicate entries can be added when using *index*. This may result in an unexpected ACL and should be avoided.
3. The read, write, execute, and deleted attributes are set based on the corresponding stem variables. A value of 1 indicates the attribute is to be set to 1. Any other value, including not setting the variable, results in the attribute being set to 0.
4. If the requested entry cannot be located, a new entry is created and the relative index for that entry is returned in **retval**. If an existing entry is updated, **retval** will contain 0. If **retval= -1** and **errno=enoent**, then a new entry could not be created because it would exceed the maximum number of entries (1024).

Example

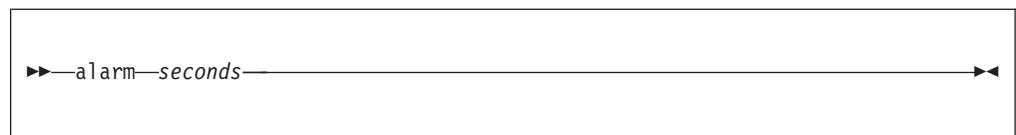
To update an ACL entry, this example assumes the user has set the appropriate stem variable before the call.

```
"aclupdateentry tokenvar acl."
```

For a complete example that uses several of the ACL services to list ACLs, see "List the ACL entries for a file" on page 172.

For more information about access control lists, see Using access control lists (ACLs) in *z/OS UNIX System Services Planning*.

alarm



alarm

Function

alarm invokes the alarm callable service to generate a **SIGALRM** signal after the number of seconds specified have elapsed.

Parameters

seconds

The number of seconds to pass between receipt of this request and generation of the **SIGALRM** signal.

Usage notes

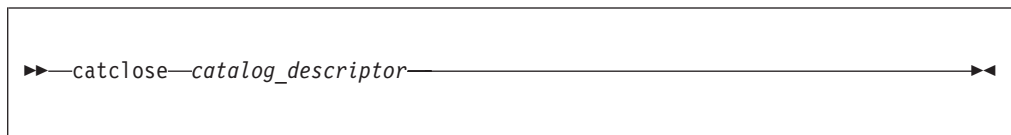
1. The default action for an alarm signal is to end a process.
2. The alarm callable service is always successful, and no return value is reserved to indicate an error.
3. An abend is generated when failures are encountered that prevent the alarm callable service from completing successfully.
4. Alarm requests are not stacked; only one **SIGALRM** can be scheduled to be generated at a time. If the previous alarm time did not expire and a new alarm is scheduled, the most recent alarm reschedules the time that **SIGALRM** is generated.
5. See "Using the REXX signal services" on page 10 for additional information on using signals.

Example

To generate a **SIGALRM** after 10 seconds:

```
"alarm 10"
```

catclose



Function

catclose closes a message catalog that was opened by **catopen**.

Parameters

catalog_descriptor

The catalog descriptor (a number) returned by **catopen** when the message catalog was opened.

Usage notes

If it is unsuccessful, **catclose** returns -1 and sets **ERRNO** to indicate the error.

Example

See the example for "catgets" on page 31.

catgets

```
►►—catgets—catalog_descriptor—set_number—message_number—variable—◄◄
```

Function

catgets locates and returns a message in a message catalog.

Parameters

catalog_descriptor

The catalog descriptor (a number) returned by **catopen** when a message catalog was opened earlier.

set_number

A number that identifies a message set in the message catalog.

message_number

A number that identifies a message in a message set in the message catalog.

variable

The name of the buffer in which the message string is returned.

Usage notes

1. Set *variable* to a default message text prior to invoking the **catgets** command. If the message identified by *message_number* is not found, *variable* is not altered and can be used after the command has been invoked.
2. If the command is unsuccessful, *variable* is returned and ERRNO may be set to indicate the error.

Example

```
"catopen mymsgs.cat"
cd=retval
:
msg='error processing request'
"catgets (cd) 1 3 msg"
say msg
:
"catclose" cd
```

catopen

```
►►—catopen—catalog_name—◄◄
```

Function

catopen opens a message catalog that has been built by the **genocat** utility. (For more information about **genocat**, see the **genocat** command description in *z/OS UNIX System Services Command Reference*.) The catalog descriptor is returned in RETVAL.

catopen

Parameters

catalog name

The pathname for the message catalog. If the pathname contains a slash (/), the environment variables `NLSPATH` and `LANG` do not affect the resolution of the pathname.

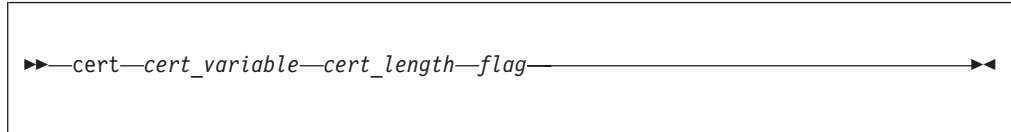
Usage notes

1. The catalog descriptor returned in `RETV` can be used with the `catgets` and `catclose` commands. Do not use the catalog descriptor with any other commands.
2. If it is unsuccessful, `catopen` returns a -1 and sets `ERRNO` to indicate the error.

Example

See the example for “`catgets`” on page 31.

cert



Function

`cert` calls `BPX1SEC` to register or deregister a certificate with the calling user.

Parameters

cert_variable

The name of the variable that contains the certificate.

cert_length

The length of the certificate

flag

Valid values for the flag are:

- 0 — register
- 1 — deregister

Usage notes

1. The intent of the `cert` service is to provide a way for the caller to associate/disassociate a certificate with the calling user. No new security environment is created and no authentication of the user is conducted.
2. The caller needs access to the `RACDCERT` facility class (as defined in the `initACEE` documentation) to register/deregister a certificate. No other authority above that is required to use `cert`.
3. The certificate is a data area that includes a 4-byte length field, header information for some certificate types, the actual certificate, and trailer information for some certificate types. The length passed in on the `cert` syscall is the whole length of that data area. It is up to the caller to build the appropriate structure.

Example

```
'cert newcert' length(newcert) 1
```

chattr

```
▶—chattr—pathname—attribute_list—◀
```

Function

chattr invokes the chattr callable service to set the attributes associated with a file. You can change the file mode, owner, access time, modification time, change time, reference time, audit flags, general attribute flags, and file size.

Parameters

pathname

The pathname of the file.

attribute_list

A list of attributes to be set and their values. The attributes are expressed either as numeric values (see Appendix A, “REXX predefined variables,” on page 241), or as the predefined variables beginning with `ST_`, followed by arguments for that attribute. The attributes that may be changed and their parameters are:

Variable	Description
<code>ST_CCSID</code>	Coded character set ID; first 4 characters are the file tag.
<code>ST_MODE</code>	1 argument: permission bits as 3 octal digits.
<code>ST_UID</code>	2 arguments: UID and GID numbers.
<code>ST_SIZE</code>	1 argument: new file size.
<code>ST_ETIME</code>	1 argument for access time: new time or -1 for TOD.
<code>ST_MTIME</code>	1 argument for modification time: new time or -1 for TOD.
<code>ST_CTIME</code>	1 argument for change time: new time or -1 for TOD.
<code>ST_SETUID</code>	No arguments.
<code>ST_SETGID</code>	No arguments.
<code>ST_AAUDIT</code>	1 argument: new auditor audit value.
<code>ST_UAUDIT</code>	1 argument: new user audit value.
<code>ST_STICKY</code>	No arguments.
<code>ST_GENVALUE</code>	2 arguments: names of two variables. The first variable contains the general attribute mask and the second contains the general attribute value.
<code>ST_RUNTIME</code>	1 argument for reference time: new time or -1 for TOD.

Variable	Description
ST_FILEFMT	Format of the file. To specify the format, you can specify a numeric value (see Appendix A, "REXX predefined variables," on page 241) or one of the following predefined variables used to derive the appropriate numeric value:
S_FFBINARY	Binary data
S_FFCR	Text data delimited by a carriage return character
S_FFCRLF	Text data delimited by carriage return and line feed characters
S_FFCRNL	A text file with lines delimited by carriage-return and newline characters.
S_FFLF	Text data delimited by a line feed character
S_FFLFCR	Text data delimited by a line feed and carriage return characters
S_FFNA	Text data with the file format not specified
S_FFNL	Text data delimited by a newline character
S_FFRECORD	File data consisting of records with prefixes. The record prefix contains the length of the record that follows.

Usage notes

1. Some of the attributes changed by the chattr service can also be changed by other services.
2. When changing the mode:
 - The effective UID of the calling process must match the file's owner UID, or the caller must have appropriate privileges.
 - Setting the set-group-ID-on-execution permission (in mode) means that when this file is run (through the exec service), the effective GID of the caller is set to the file's owner GID, so that the caller seems to be running under the GID of the file, rather than that of the actual invoker.
 The set-group-ID-on-execution permission is set to zero if both of the following are true:
 - The caller does not have appropriate privileges.
 - The GID of the file's owner does not match the effective GID, or one of the supplementary GIDs, of the caller.
 - Setting the set-user-ID-on-execution permission (in mode) means that when this file is run, the process's effective UID is set to the file's owner UID, so that the process seems to be running under the UID of the file's owner, rather than that of the actual invoker.
3. When changing the owner:
 - For changing the owner UID of a file, the caller must have appropriate privileges.
 - For changing the owner GID of a file, the caller must have appropriate privileges, or meet all of these conditions:
 - The effective UID of the caller matches the file's owner UID.

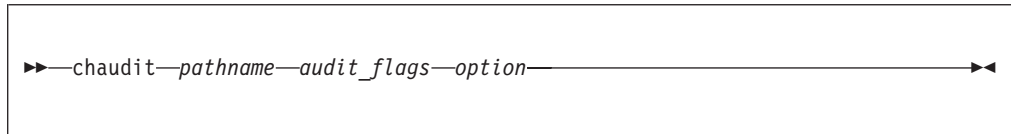
- The owner UID value specified in the change request matches the file's owner UID.
 - The GID value specified in the change request is the effective GID, or one of the supplementary GIDs, of the caller.
 - When changing the owner, the set-user-ID-on-execution and set-group-ID-on-execution permissions of the file mode are automatically turned off.
 - When the owner is changed, both UID and GID must be specified as they are to be set. If you want to change only one of these values, you need to set the other to its present value for it to remain unchanged.
4. For general attribute bits to be changed, the calling process must have write permission for the file.
 5. When changing the file size:
 - The change is made beginning from the first byte of the file. If the file was previously larger than the new size, the data from *file_size* to the original end of the file is removed. If the file was previously shorter than *file_size*, bytes between the old and new lengths are read as zeros. The file offset is not changed.
 - If *file_size* is greater than the current file size limit for the process, the request fails with EFBIG, and the SIGXFSZ signal is generated for the process.
 - Successful change clears the set-user-ID, the set-group-ID, and the save-text (sticky bit) attributes of the file, unless the caller is a superuser.
 6. When changing times:
 - For the access time or the modification time to be set explicitly (using either *st_atime* or *st_mtime* with the new time), the effective ID must match that of the file's owner, or the process must have appropriate privileges.
 - For the access time or modification time to be set to the current time (using either *st_atime* or *st_mtime* with -1), the effective ID must match that of the file's owner, the calling process must have write permission for the file, or the process must have appropriate privileges.
 - For the change time or the reference time to be set explicitly (using either *st_ctime* or *st_rtime* with the new time), the effective ID must match that of the file's owner, or the process must have appropriate privileges.
 - For the change time or reference time to be set to the current time (using either *st_ctime* or *st_rtime* with -1), the calling process must have write permission for the file.
 - When any attribute field is changed successfully, the file's change time is also updated.
 7. For auditor audit flags to be changed, the user must have auditor authority. The user with auditor authority can set the auditor options for any file, even those to which they do not have path access or authority to use for other purposes.
Auditor authority is established by issuing the TSO/E command ALTUSER AUDITOR.
 8. For the user audit flags to be changed, the user must have appropriate privileges or be the owner of the file.
 9. The tagging of /dev/null, /dev/zero, /dev/random, and /dev/urandom is ignored.

Example

To set permissions for /u/project to 775:

```
"chattr /u/project" st_mode 775
```

chaudit



Function

chaudit invokes the chaudit callable service to change audit flags for a file.

Parameters

pathname

The pathname of the file.

audit_flags

One or more numeric values that indicate the type of access to be tested. You can specify a numeric value (see Appendix A, "REXX predefined variables," on page 241) or the predefined variable used to derive the appropriate numeric value. The predefined variables you can specify are:

Variable	Description
AUD_FREAD	Audit failed read requests
AUD_SREAD	Audit successful read requests
AUD_FWRITE	Audit failed write requests
AUD_SWRITE	Audit successful write requests
AUD_FEXEC	Audit failed execute or search requests
AUD_SEXEC	Audit successful execute or search requests

option

A number indicating whether user-requested or auditor-requested auditing is being changed:

- 0 if user-requested auditing is being changed.
- 1 if auditor-requested auditing is being changed.

Usage notes

1. If *option* indicates that the auditor audit flags are to be changed, you must have auditor authority for the request to be successful. If you have auditor authority, you can set the auditor options for any file, even those to which you do not have path access or authority to use for other purposes.

You can get auditor authority by entering the TSO/E command ALTUSER AUDITOR.

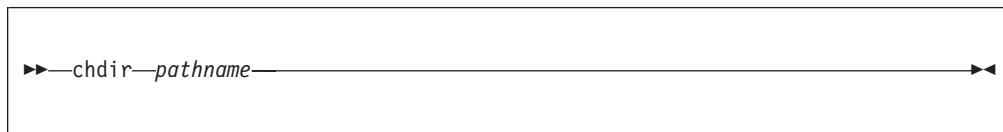
2. If *option* indicates that the user audit flags are to be changed, you must have appropriate privileges or be the owner of the file.

Example

In the following example, assume that *pathname* was assigned a value earlier in the exec. To change user-requested auditing so that failed read, write, and execute attempts for *pathname* are audited:

```
"chaudit (pathname)" aud_fread+aud_fwrite+aud_fexec 0
```

chdir



Function

`chdir` invokes the `chdir` callable service to change the working directory.

Parameters

`pathname`

The pathname of the directory.

Usage notes

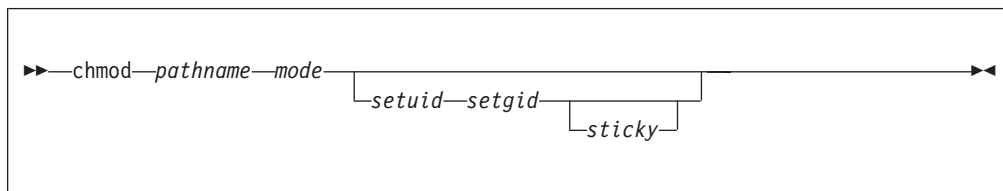
If you use `chdir` to change a directory in a REXX program that is running in a TSO/E session, the directory is typically reset to your home directory when the REXX program ends. When a REXX program changes directories and then exits, the thread is undubbed. If this was the only thread dubbed in your TSO/E session, the working directory is reset to the home directory the next time a syscall command is issued. However, if there is more than one dubbed thread in the address space, the remaining threads keep the working directory even when the REXX program exits.

Example

To change the working directory to `/u/lou/dirb`:

```
"chdir /u/lou/dirb"
```

chmod



Function

`chmod` invokes the `chmod` callable service to change the mode of a file or directory.

Parameters

pathname

The pathname of the file or directory.

mode

A three- or four-digit number, corresponding to the access permission bits. Each digit must be in the range 0–7, and at least three digits must be specified. For more information on permissions, see Appendix B, “Setting permissions for files and directories,” on page 253.

setuid

Sets the set-user-ID-on-execution permission. Specify 1 to set this permission on, or 0 to set it off. The default is 0.

setgid

Sets the set-group-ID-on-execution permission. Specify 1 to set this permission on, or 0 to set it off. The default is 0.

sticky

The sticky bit for a file indicates where the file should be fetched from. If the file resides in the link pack area (LPA), link list, or STEPLIB, specify 1. The default is 0.

Setting the sticky bit for a directory to 1 indicates that to delete or rename a file, the effective user ID of the process must be the same as that of the directory owner or file owner, or that of a superuser. Setting the sticky bit for a directory to 0 indicates that anyone who has write permission to the directory can delete or rename a file.

Usage notes

1. One bit sets permission for set-user-ID on access, set-group-ID on access, or the *sticky bit*. You can set this bit in either of two ways:
 - Specifying four digits on the *mode* parameter; the first digit sets the bit.
 - Specifying the *setuid*, *setgid*, or *sticky* parameters.
2. When a **chmod** or **fchmod** has occurred for an open file, **fstat** reflects the change in mode. However, no change in access authorization is apparent when the file is accessed through a previously opened file descriptor.
3. For mode bits to be changed, the effective UID of the caller must match the file's owner UID, or the caller must be a superuser.
4. When the mode is changed successfully, the file's change time is also updated.
5. Setting the set-group-ID-on-execution permission means that when this file is run (through the exec service), the effective GID of the caller is set to the file's owner GID, so that the caller seems to be running under the GID of the file, rather than that of the actual invoker.

The set-group-ID-on-execution permission is set to zero if both of the following are true:

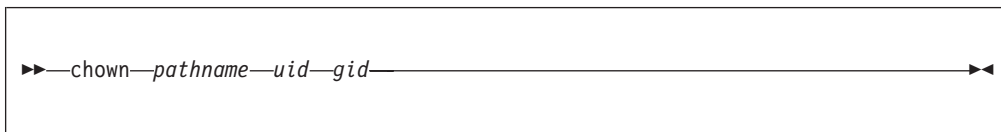
- The caller does not have appropriate privileges.
 - The GID of the file's owner does not match the effective GID or one of the supplementary GIDs of the caller.
6. Setting the set-user-ID-on-execution permission means that when this file is run, the process's effective UID is set to the file's owner UID, so that the process seems to be running under the UID of the file's owner, rather than that of the actual invoker.

Example

In the following example, assume that *pathname* was assigned a value earlier in the exec. This example changes the mode of the file to read-write-execute for the owner, and read-execute for all others:

```
"chmod (pathname) 755"
```

chown



Function

chown invokes the chown callable service to change the owner or group for a file or directory.

Parameters

pathname

The pathname of a file or directory.

uid

The numeric UID for the new owner of the file or the present UID, or -1 if there is no change.

gid

The numeric GID for the group for the file or the present GID, or -1 if there is no change.

Usage notes

1. The chown service changes the owner UID and owner GID of a file. Only a superuser can change the owner UID of a file.
2. The owner GID of a file can be changed by a superuser, or if a caller meets all of these conditions:
 - The effective UID of the caller matches the file's owner UID.
 - The *uid* value specified in the change request matches the file's owner UID.
 - The *gid* value specified in the change request is the effective GID, or one of the supplementary GIDs, of the caller.
3. The set-user-ID-on-execution and set-group-ID-on-execution permissions of the file mode are automatically turned off.
4. If the change request is successful, the change time for the file is updated.
5. Values for both *uid* and *gid* must be specified as they are to be set. If you want to change only one of these values, the other must be set to its present value to remain unchanged.

Example

In the following example, assume that *pathname*, *uid*, and *gid* were assigned a value earlier in the exec:

```
"chown (pathname) (uid) (gid)"
```

close



Function

close invokes the close callable service to close a file.

Parameters

fd The file descriptor (a number) for the file to be closed.

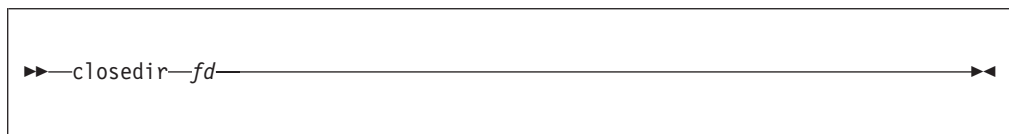
Usage notes

1. Closing a file closes, or frees, the file descriptor by which the file was known to the process. The system can then reassign the file descriptor to the same file or to another file when it is opened.
2. Closing a file descriptor also unlocks all outstanding byte range locks that a process has on the associated file.
3. If a file has been opened by more than one process, each process has a file descriptor. When the last open file descriptor is closed, the file itself is closed. If the file's link count is zero at that time, the file's space is freed and the file becomes inaccessible. When the last open file descriptor for a pipe or FIFO special file is closed, any data remaining in the file is discarded.

Example

In the following example, assume that *fd* was assigned a value earlier in the exec:
 "close" fd

closedir



Function

closedir invokes the closedir callable service to close a directory.

Parameters

fd The file descriptor (a number) for the directory to be closed.

Usage notes

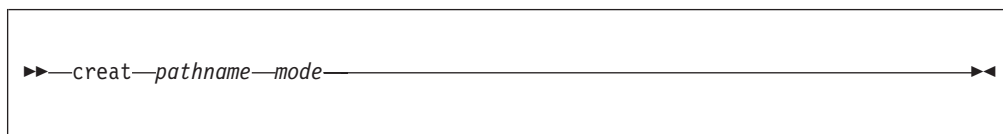
closedir closes a directory file descriptor opened by the **opendir** syscall command. The **rddir** command reads a directory in the readdir callable service format. You can use **opendir**, **rewinddir**, and **closedir** together with the **rddir** syscall command, but not with the **readdir** syscall command. Alternatively, you can simply use the

readdir syscall command to read an entire directory and format it in a stem.

Example

In the following example, assume that *fd* was assigned a value earlier in the exec:
`"closedir" fd`

creat



Function

creat invokes the open callable service to open a new file. The file descriptor is returned in RETVAL.

Parameters

pathname

The pathname of a file.

mode

A three- or four-digit number, corresponding to the access permission bits. Each digit must be in the range 0–7, and at least three digits must be specified. For more information on permissions, see Appendix B, “Setting permissions for files and directories,” on page 253.

Usage notes

Using **creat** is the equivalent of using the open callable service with the create, truncate, and write-only options:

- When a file is created with the create option, the file permission bits as specified in *mode* are modified by the process's file creation mask (see “umask” on page 149) and then used to set the file permission bits of the file being created.
- The truncate option opens the file as though it had been created earlier, but never written into. The mode and owner of the file do not change (although the change time and modification time do), but the file's contents are discarded. The file offset, which indicates where the next write is to occur, points to the first byte of the file.

Example

To open a new file, `/u/lou/test.exec`, with read-write-execute permission for the owner only:

```
"creat /u/lou/test.exec 700"
```

dup

dup



Function

dup invokes the `fcntl` callable service to duplicate an open file descriptor. The file descriptor is returned in `RETVAL`.

Parameters

fd An opened file descriptor (a number) to be duplicated.

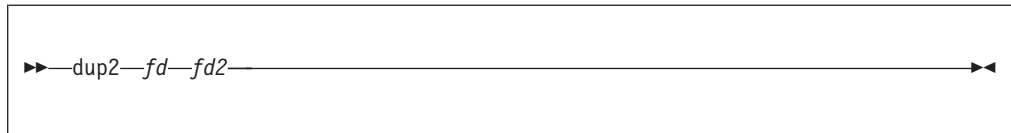
Usage notes

`dup fd` is equivalent to `F_DUPFD fd 0`.

Example

In the following example, assume that `fd` was assigned a value earlier in the exec:
"dup (fd)"

dup2



Function

dup2 invokes the `fcntl` callable service to duplicate an open file descriptor to the file descriptor of choice. The file descriptor returned is equal to `fd2`. If `fd2` is already in use, it is closed and `fd` is duplicated. If `fd` is equal to `fd2`, `fd2` is returned without closing it. The file descriptor is returned in `RETVAL`.

Parameters

fd An opened file descriptor (a number) to be duplicated.

fd2

The file descriptor (a number) to be changed.

Usage notes

`dup fd fd2` is equivalent to `F_DUPFD fd fd2`.

Example

In the following example, assume that `fd1` and `fd2` were assigned values earlier in the exec:

```
"dup2" fd1 fd2
```

exec

There is no **exec** syscall command. Instead of using **exec**, see “spawn” on page 137.

extlink

```
▶▶—extlink—extname—linkname————▶▶
```

Function

extlink invokes the extlink callable service to create a symbolic link to an external name. This creates a symbolic link file.

Parameters

extname

The external name of the file for which you are creating a symbolic link.

linkname

The pathname for the symbolic link.

Usage notes

1. The object identified by *extname* need not exist when the symbolic link is created, and refers to an object outside a hierarchical file system.
2. The external name contained in an external symbolic link is not resolved. The *linkname* cannot be used as a directory component of a pathname.

Example

To create a symbolic link named **mydsn** for the file **WJS.MY.DSN**:

```
"extlink WJS.MY.DSN /u/wjs/mydsn"
```

fchattr

```
▶▶—fchattr—fd—attribute_list————▶▶
```

Function

fchattr invokes the fchattr callable service to modify the attributes that are associated with a file represented by a file descriptor. You can change the mode, owner, access time, modification time, change time, reference time, audit flags, general attribute flags, and file size.

Parameters

fd The file descriptor for the file.

attribute_list

A list of attributes to be set and their values. The attributes are expressed either as numeric values (see Appendix A, "REXX predefined variables," on page 241), or as the predefined variables beginning with ST_ followed by arguments for that attribute. The attributes that may be changed and their parameters are:

Variable	Description
ST_CCSID	Coded character set ID; first 4 characters are the file tag.
ST_MODE	1 argument: permission bits as 3 octal digits.
ST_UID	2 arguments: UID and GID numbers.
ST_SIZE	1 argument: new file size.
ST_ETIME	1 argument for access time: new time or -1 for TOD.
ST_MTIME	1 argument for modification time: new time or -1 for TOD.
ST_CTIME	1 argument for change time: new time or -1 for TOD.
ST_SETUID	No arguments.
ST_SETGID	No arguments.
ST_AAUDIT	1 argument: new auditor audit value.
ST_UAUDIT	1 argument: new user audit value.
ST_STICKY	No arguments.
ST_GENVALUE	2 arguments: names of two variables. The first variable contains the general attribute mask and the second contains the general attribute value.
ST_RUNTIME	1 argument for reference time: new time or -1 for TOD.
ST_FILEFMT	Format of the file. To specify the format, you can specify a numeric value (see Appendix A, "REXX predefined variables," on page 241) or one of the following predefined variables used to derive the appropriate numeric value: S_FFBINARY Binary data S_FFCR Text data delimited by a carriage return character S_FFRLF Text data delimited by carriage return and line feed characters S_FFRCRNL A text file with lines delimited by carriage-return and newline characters. S_FFLF Text data delimited by a line feed character S_FFRCR Text data delimited by a line feed and carriage return characters S_FFNA Text data with the file format not specified S_FFNL Text data delimited by a newline character S_FFRECORD File data consisting of records with prefixes. The record prefix contains the length of the record that follows.

Usage notes

1. Some of the attributes changed by the fchattr service can also be changed by other services.
2. When changing the mode:
 - The effective UID of the calling process must match the file's owner UID, or the caller must have appropriate privileges.

- Setting the set-group-ID-on-execution permission (in mode) means that when this file is run (through the exec service), the effective GID of the caller is set to the file's owner GID, so that the caller seems to be running under the GID of the file, rather than that of the actual invoker.

The set-group-ID-on-execution permission is set to zero if both of the following are true:

- The caller does not have appropriate privileges.
 - The GID of the file's owner does not match the effective GID, or one of the supplementary GIDs, of the caller.
- Setting the set-user-ID-on-execution permission (in mode) means that when this file is run, the process's effective UID is set to the file's owner UID, so that the process seems to be running under the UID of the file's owner, rather than that of the actual invoker.
3. When changing the owner:
 - For changing the owner UID of a file, the caller must have appropriate privileges.
 - For changing the owner GID of a file, the caller must have appropriate privileges, or meet all of these conditions:
 - The effective UID of the caller matches the file's owner UID.
 - The owner UID value specified in the change request matches the file's owner UID.
 - The GID value specified in the change request is the effective GID, or one of the supplementary GIDs, of the caller.
 - When the owner is changed, the set-user-ID-on-execution and set-group-ID-on-execution permissions of the file mode are automatically turned off.
 - When the owner is changed, both UID and GID must be specified as they are to be set. If you want to change only one of these values, you need to set the other to its present value for it to remain unchanged.
 4. For general attribute bits to be changed, the calling process must have write permission for the file.
 5. When changing the file size:
 - The change is made beginning from the first byte of the file. If the file was previously larger than the new size, the data from *file_size* to the original end of the file is removed. If the file was previously shorter than *file_size*, bytes between the old and new lengths are read as zeros. The file offset is not changed.
 - If *file_size* is greater than the current file size limit for the process, the request fails with EFBIG and the SIGXFSZ signal is generated for the process.
 - Successful change clears the set-user-ID, set-group-ID, and save-text (sticky bit) attributes of the file unless the caller is a superuser.
 6. When changing times:
 - For the access time or the modification time to be set explicitly (using either *st_atime* or *st_mtime* with the new time), the effective ID must match that of the file's owner, or the process must have appropriate privileges.
 - For the access time or modification time to be set to the current time (using either *st_atime* or *st_mtime* with -1), the effective ID must match that of the file's owner, the calling process must have write permission for the file, or the process must have appropriate privileges.

fchattr

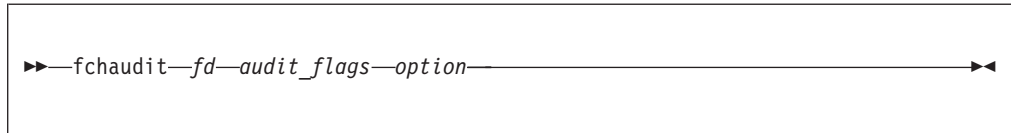
- For the change time or the reference time to be set explicitly (using either *st_ctime* or *st_rtime* with the new time) the effective ID must match that of the file's owner, or the process must have appropriate privileges.
 - For the change time or reference time to be set to the current time (using either *st_ctime* or *st_rtime* with -1), the calling process must have write permission for the file.
 - When any attribute field is changed successfully, the file's change time is also updated.
7. For auditor audit flags to be changed, the user must have auditor authority. The user with auditor authority can set the auditor options for any file, even those to which they do not have path access or authority to use for other purposes.
- Auditor authority is established by issuing the TSO/E command ALTUSER AUDITOR.
8. For the user audit flags to be changed, the user must have appropriate privileges or be the owner of the file.
9. The tagging of /dev/null, /dev/zero, /dev/random, and /dev/urandom is ignored.

Example

In the following example, assume that *fd* was assigned a value earlier in the exec. This truncates a file to 0 bytes and sets the file permissions to 600:

```
"fchattr" fd st_size 0 st_mode 600
```

fchaudit



Function

fchaudit invokes the fchaudit callable service to change audit flags for a file identified by a file descriptor. The file descriptor is specified by a number.

Parameters

fd The file descriptor for the file.

audit_flags

One or more numeric values that indicate the type of access to be tested. You can specify a numeric value (see Appendix A, "REXX predefined variables," on page 241) or the predefined variable used to derive the appropriate numeric value. The predefined variables you can specify are:

Variable	Description
AUD_FREAD	Audit failed read requests
AUD_SREAD	Audit successful read requests
AUD_FWRITE	Audit failed write requests
AUD_SWRITE	Audit successful write requests
AUD_FEXEC	Audit failed execute or search requests
AUD_SEXEC	Audit successful execute or search requests

option

A number indicating whether user-requested or auditor-requested auditing is being changed:

- 0 if user-requested auditing is being changed.
- 1 if auditor-requested auditing is being changed.

Usage notes

1. If *option* indicates that the auditor audit flags are to be changed, you must have auditor authority for the request to be successful. If you have auditor authority, you can set the auditor options for any file, even those to which you do not have path access or authority to use for other purposes.

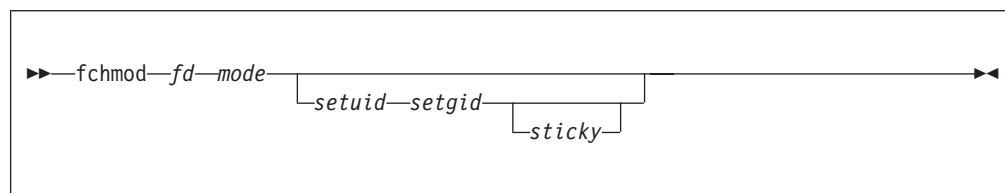
You can get auditor authority by entering the TSO/E command ALTUSER AUDITOR.

2. If *option* indicates that the user audit flags are to be changed, you must have appropriate privileges or be the owner of the file.

Example

To change user-requested auditing so that failed read requests for the file identified by file descriptor 0 are audited:

```
"fchaudit 0 (aud_fread) 0"
```

fchmod**Function**

fchmod invokes the fchmod callable service to change the mode of a file or directory indicated by a file descriptor. The file descriptor is specified by a number.

Parameters

fd The file descriptor for the file or directory.

mode

A three- or four-digit number, corresponding to the access permission bits. Each digit must be in the range 0–7, and at least three digits must be specified. For more information on permissions, see Appendix B, “Setting permissions for files and directories,” on page 253.

setuid

Sets the set-user-ID-on-execution permission. Specify 1 to set this permission on, or 0 to set it off. The default is 0.

setgid

Sets the set-group-ID-on-execution permission. Specify 1 to set this permission on, or 0 to set it off. The default is 0.

fchmod

sticky

Sets the sticky bit to indicate where the file should be fetched from. If the file resides in the link pack area (LPA), link list, or STEPLIB, specify 1. The default is 0.

Usage notes

1. One bit sets permission for set-user-ID on access, set-group-ID on access, or the *sticky bit*. You can set this bit in either of two ways:
 - Specifying four digits on the *mode* parameter; the first digit sets the bit.
 - Specifying the *setuid*, *setgid*, or *sticky* parameters.
2. When a **chmod** or **fchmod** has occurred for an open file, **fstat** reflects the change in mode. However, no change in access authorization is apparent when the file is accessed through a previously opened file descriptor.
3. For mode bits to be changed, the effective UID of the caller must match the file's owner UID, or the caller must be a superuser.
4. When the mode is changed successfully, the file's change time is also updated.
5. Setting the set-group-ID-on-execution permission means that when this file is run, through the exec service, the effective GID of the caller is set to the file's owner GID, so that the caller seems to be running under the GID of the file, rather than that of the actual invoker.

The set-group-ID-on-execution permission is set to zero if both of the following are true:

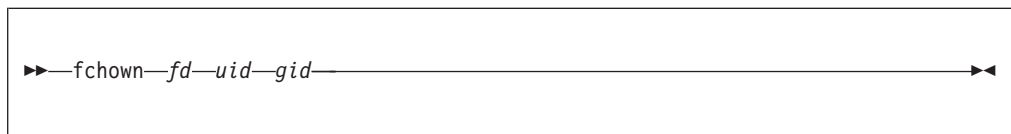
- The caller does not have appropriate privileges.
 - The GID of the file's owner does not match the effective GID or one of the supplementary GIDs of the caller.
6. Setting the set-user-ID-on-execution permission means that when this file is run, the process's effective UID is set to the file's owner UID, so that the process seems to be running under the UID of the file's owner, rather than that of the actual invoker.

Example

In the following example, assume that *fd* was assigned a value earlier in the exec. This changes the mode for the file identified by the file descriptor so that only a superuser can access the file:

```
"fchmod (fd) 000"
```

fchown



Function

fchown invokes the fchown callable service to change the owner and group of a file or directory indicated by a file descriptor. The file descriptor is specified by a number.

Parameters

fd The file descriptor for a file or directory.

uid

The numeric UID for the new owner of the file or the present UID, or -1 if there is no change.

gid

The numeric GID for the new group for the file or the present GID, or -1 if there is no change.

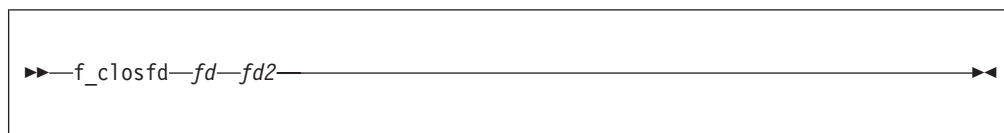
Usage notes

1. The fchown service changes the owner UID and owner GID of a file. Only a superuser can change the owner UID of a file.
2. The owner GID of a file can be changed by a superuser, or if a caller meets all of these conditions:
 - The effective UID of the caller matches the file's owner UID.
 - The *uid* value specified in the change request matches the file's owner UID.
 - The *gid* value specified in the change request is the effective GID, or one of the supplementary GIDs, of the caller.
3. The set-user-ID-on-execution and set-group-ID-on-execution permissions of the file mode are automatically turned off.
4. If the change request is successful, the change time for the file is updated.
5. Values for both *uid* and *gid* must be specified as they are to be set. If you want to change only one of these values, the other must be set to its present value to remain unchanged.

Example

In the following example, assume that *fd*, *uid*, and *gid* were assigned a value earlier in the exec:

```
"fchown" fd uid gid
```

f_closfd**Function**

`f_closfd` invokes the `fcntl` callable service to close a range of file descriptors.

Parameters

fd The file descriptor (a number) for a file. This is the first file descriptor to be closed.

fd2

The file descriptor (a number) for a file, which must be greater than or equal to *fd*. If a -1 is specified for *fd2*, all file descriptors greater than or equal to *fd* are closed.

Usage notes

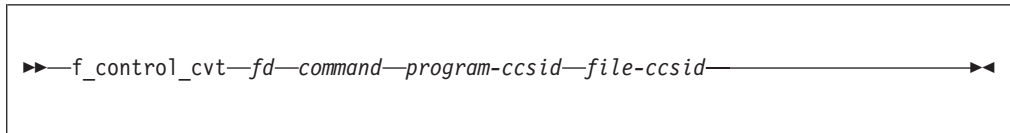
1. A process can use **f_closfd** to close a range of file descriptors. *fd2* must be greater than or equal to *fd*, or it can also be -1, which indicates that all file descriptors greater than or equal to *fd* are to be closed.
2. Use of **f_closfd** is meant to be consistent with the close callable service. You cannot close file descriptors that could not also be closed using the close service.
3. When a file descriptor cannot be closed, it is considered an error, but the request continues with the next file descriptor in the range. File descriptors that are not in use are ignored.

Example

In the following example, assume that *fd* and *fd2* were assigned values earlier in the exec:

```
"f_closfd" fd fd2
```

f_control_cvt



Function

f_control_cvt controls automatic file conversion and specifies the program and file CCSIDs (character code set identifiers) for an opened I/O stream.

Parameters

fd The file descriptor (a number) for the file. It must be a regular file, FIFO, or character special file.

command

The *command* can be one of these variables:

CVT_SETCVTOFF

Turns off any conversion that might be in effect. A hexadecimal value of 0 for both *program-ccsid* and *file-ccsid* is recommended. If automatic conversion is set to ALL and I/O has already started for the file, this command is ignored.

CVT_SETCVTON

Turns automatic conversion on for the stream and, optionally, sets the program CCSID or file CCSID, or both. A hexadecimal value of 0 for *program-ccsid* indicates using ThliCcsid (the current program CCSID) at the time of each read or write. ThliCcsid is initially 1047, but can be reset directly by the program, or indirectly by setting the appropriate runtime option or environment variable. A hex value of 0 for *file-ccsid* indicates not changing ThliCcsid.

Setting or referencing ThliCcsid is still valid but not recommended.

CVT_SETAUTOCVTON

Conditionally turns automatic conversion on for the stream and, optionally, sets the program CCSID or file CCSID, or both. Conversion

will be in effect for this stream only if the system or the local runtime environment has been enabled for conversion. A hex value of 0 for *program-ccsid* indicates using ThliCcsid at the time of each read or write. ThliCcsid is initially 1047, but can be changed by this variable, or by setting the appropriate runtime option or environment variable. A hex value of 0 for *file-ccsid* indicates not changing ThliCcsid.

CVT_QUERYCVT

Returns an indicator that lets you know whether automatic conversion is in effect or not, and also returns the program and file CCSIDs that are being used (if conversion is in effect). The variables *command*, *program-ccsid*, and *file-ccsid* are set when you have a successful return. *command* is set to either CVT_SETCVTOFF or CVT_SETCVTON or CVT_SETCVTALL. *program-ccsid* and *file-ccsid* are set to the CCSID values that would be in use if conversion were to occur.

CVT_SETCVTALL

Behaves the same as SetCvtOn, except automatic conversion is set to ALL, which enables UNICODE conversion. ThliCcsid is not used. CVT_SETCVTALL is ignored if I/O for the file has already started. A thread can set different program CCSIDs for each open file. However, an I/O error will result if any two threads have different program CCSIDs for the same open file that is shared by those two threads.

CVT_SETAUTOCVTALL

If conversion is enabled for the environment (by BPXPRMxx parmlib statement AUTOCVT setting of ALL or with the appropriate environment variable), this subcommand behaves identically to CVT_SETCVTALL. Otherwise, it has no effect.

program-ccsid

The name of a 2-byte hex variable that describes the CCSID for the running program.

Note: For EBCDIC (1047) the CCSID is '0417'X; for ASCII (819) it is '0333'X.

file-ccsid

The name of a 2-byte hex variable that describes the CCSID for the file opened with the file descriptor *fd*.

Usage notes

- When the file is /dev/null, /dev/zero, /dev/random, or /dev/urandom, the file tag is not hardened to disk.

Example

This example turns automatic file conversion on for a stream opened with *fd* as the file descriptor:

```
pccsid = '0000'x
fccsid = '0000'x
cmd    = cvt_setautocvton
"f_control_cvt (fd) cmd pccsid fccsid"
```

This example queries the current conversion state:

```
pccsid = '0000'x
fccsid = '0000'x
cmd    = cvt_querycvt
"f_control_cvt (fd) cmd pccsid fccsid"
```

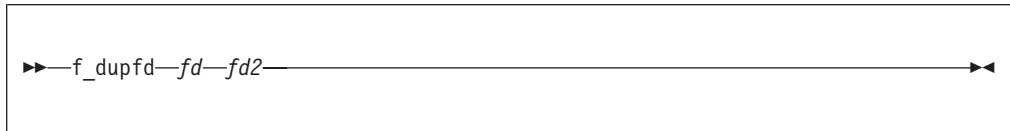
fcntl

Function

fcntl is supported as a set of syscall commands whose names begin with **f_**:

- “f_closfd” on page 49
- “f_dupfd”
- “f_dupfd2”
- “f_getfd” on page 53
- “f_getfl” on page 53
- “f_getlk” on page 54
- “f_setfd” on page 59
- “f_setfl” on page 59
- “f_setlk” on page 60
- “f_setlkw” on page 62

f_dupfd



Function

f_dupfd invokes the **fcntl** callable service to duplicate the lowest file descriptor that is equal to or greater than *fd2* and not already associated with an open file. The file descriptor is returned in **RETV**.

Parameters

fd The file descriptor (a number) that you want to duplicate.

fd2

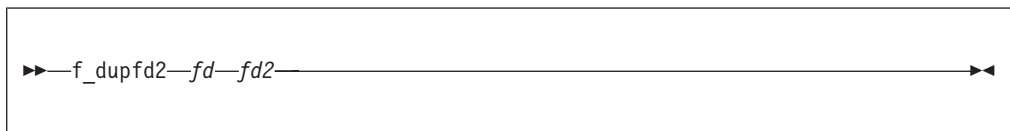
The file descriptor (a number) at which to start looking for an available file descriptor.

Example

In the following example, assume that *fd* and *fd2* were assigned values earlier in the exec:

```
"f_dupfd" fd fd2
```

f_dupfd2



Function

f_dupfd2 invokes the **fcntl** callable service to duplicate a file descriptor that is equal to *fd2*.

Parameters

fd An opened file descriptor (a number) to be duplicated.

fd2

The file descriptor of choice.

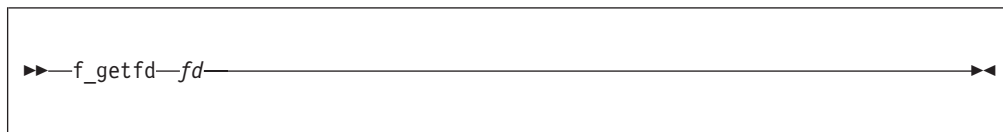
Usage notes

If *fd2* is already in use, **f_dupfd2** closes it. If *fd* is equal to *fd2*, *fd2* is returned but not closed.

Example

In the following example, assume that *fd* and *fd2* were assigned values earlier in the exec:

```
"f_dupfd" fd fd2
```

f_getfd**Function**

f_getfd invokes the `fcntl` callable service to get the file descriptor flags for a file.

Parameters

fd The file descriptor (a number) for the file.

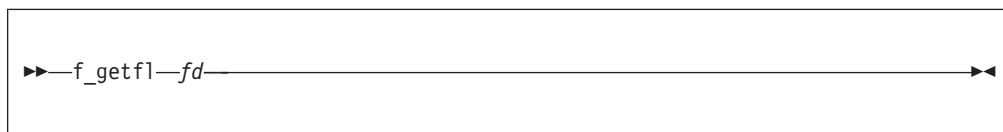
Usage notes

The file descriptor flags are returned in `RETVAL`. The only POSIX-defined flag is `FCTLCLOEXEC`. To determine if this flag is set, use this expression: $(\text{retval}/2)=1$

Example

To get the file descriptor flags for file descriptor 0:

```
"f_getfd 0"
```

f_getfl**Function**

f_getfl invokes the `fcntl` callable service to get the file status flags for a file.

f_getfl

Parameters

fd The file descriptor (a number) for the file.

Usage notes

RETVAL returns the file status flags as a numeric value (see Appendix A, "REXX predefined variables," on page 241):

Flag	Description
O_CREAT	Create the file if it does not exist.
O_EXCL	Fail if the file does exist and O_CREAT is set.
O_NOCTTY	Do not make this file a controlling terminal for the calling process.
O_TRUNC	If the file exists, truncate it to zero length.
O_APPEND	Set the offset to EOF before each write.
O_NONBLOCK	An open, read, or write on the file will not block (wait for terminal input).
O_RDWR	Open for read and write.
O_RDONLY	Open for read-only.
O_WRONLY	Open for write-only.
O_SYNC	Force synchronous updates.

You can use the open flags to test specific values. The easiest way to test a value is to convert both the RETVAL and the flags to binary data, and then logically AND them. For example, to test O_WRITE and O_TRUNC:

```
wrtr=D2C(O_WRITE+O_TRUNC,4)
If BITAND(D2C(retval,4),wrtr)=wrtr Then
  Do /* o_write and o_trunc are set */
  End
```

Example

In the following example, assume that *fd* was assigned a value earlier in the exec:
"f_getfl" fd

f_getlk



Function

f_getlk invokes the `fcntl` callable service to return information on a file segment for which locks are set, cleared, or queried.

Parameters

fd The file descriptor (a number) for the file.

stem

The name of a stem variable that is the flock structure used to query, set, or clear a lock; or to return information. To specify the information, you can specify a numeric value (see Appendix A, "REXX predefined variables," on page 241

page 241) or the predefined variables beginning with L_ used to derive the appropriate numeric value. For example, *stem.1* and *stem.l_type* are both the lock-type request:

Variable	Description
L_LEN	The length of the byte range that is to be set, cleared, or queried.
L_PID	The process ID of the process holding the blocking lock, if one was found.
L_START	The starting offset byte of the lock that is to be set, cleared, or queried.
L_TYPE	The type of lock being set, cleared, or queried. To specify the information, you can specify a numeric value (see Appendix A, "REXX predefined variables," on page 241) or one of the following predefined variables used to derive the appropriate numeric value: <ul style="list-style-type: none"> F_RDLCK Shared or read lock. This type of lock specifies that the process can read the locked part of the file, and other processes cannot write on that part of the file in the meantime. A process can change a held write lock, or any part of it, to a read lock, thereby making it available for other processes to read. Multiple processes can have read locks on the same part of a file simultaneously. To establish a read lock, a process must have the file accessed for reading. F_WRLCK Exclusive or write lock. This type of lock indicates that the process can write on the locked part of the file, without interference from other processes. If one process puts a write lock on part of a file, no other process can establish a read lock or write lock on that same part of the file. A process cannot put a write lock on part of a file if there is already a read lock on an overlapping part of the file, unless that process is the only owner of that overlapping read lock. In such a case, the read lock on the overlapping section is replaced by the write lock being requested. To establish a write lock, a process must have the file accessed for writing. F_UNLCK Unlock. This is used to unlock all locks held on the given range by the requesting process.
L_TYPE	
L_WHENCE	The flag for the starting offset. To specify the information, you can specify a numeric value (see Appendix A, "REXX predefined variables," on page 241) or one of the predefined variables beginning with SEEK_ used to derive the appropriate numeric value. Valid values are: <ul style="list-style-type: none"> SEEK_CUR The current offset SEEK_END The end of the file SEEK_SET The specified offset

Example

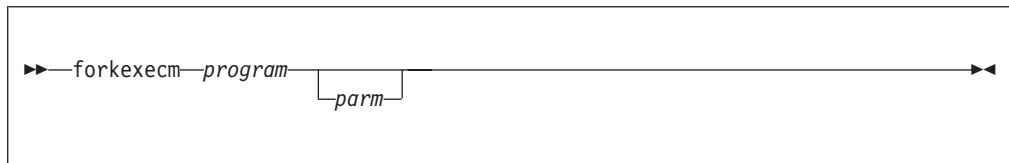
In the following example, assume that *rec* was assigned a value earlier in the exec:

```
lock.l_len=40
lock.l_start=rec*40
lock.l_type=f_wrlck
lock.l_whence=seek_set
"f_getlk" fd "lock."
if lock.l_type=f_unlck then
  /* lock is available for the requested 40 byte record */
```

fork

There is no **fork** syscall command. Instead of using **fork**, see “spawn” on page 137.

forkexecm



Function

forkexecm invokes the **fork** and **execmvs** callable services to fork and exec a program to be executed from the LINKLIB, LPALIB, or STEPLIB library.

Parameters

program

The name of the program.

parm

A parameter to be passed to the program.

Usage notes

1. If the exec fails, the child process ends with a **SIGABRT** signal.
2. If the **fork** succeeds, **RETVL** contains the PID for the child process.
3. The child process has a unique process ID (PID) that does not match any active process group ID.
4. If a hierarchical file system (HFS) file has its **FCTLFCLOFORK** flag set on, it is not inherited by the child process. This flag is set with the **fcntl** callable service.
5. The process and system utilization times for the child are set to zero.
6. Any file locks previously set by the parent are not inherited by the child.
7. The child process has no alarms set (similar to the results of a call to the alarm service with **Wait_time** specified as zero).
8. The child has no pending signals.
9. The following characteristics of the calling process are changed when the new executable program is given control by the **execmvs** callable service:
 - The prior process image is replaced with a new process image for the executable program to be run.

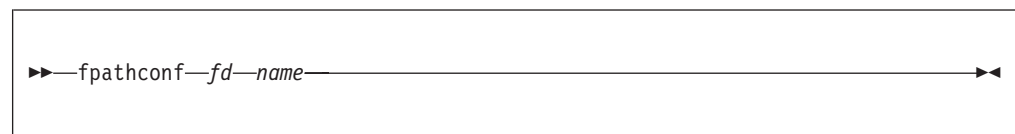
- All open files marked close-on-exec and all open directory streams are closed.
10. The input passed to the MVS executable file by the service is consistent with what is passed as input to MVS programs. On input, the MVS program receives a single-entry parameter list pointed to by register 1. The high-order bit of the single parameter entry is set to 1.
The single parameter entry is the address of a 2-byte length field followed by an argument string. The length field describes the length of the data that follows it. If a null argument and argument length are specified in the call, the length field specifies 0 bytes on input to the executable file.
 11. The call can invoke both unauthorized and authorized MVS programs:
 - Unauthorized programs receive control in problem program state, with PSW key 8.
 - Authorized programs receive control in problem program state, with PSW key 8 and APF authorization.
 12. The TASKLIB, STEPLIB, or JOBLIB DD data set allocations that are active for the calling task at the time of the call to the execmvs service are propagated to the new process image, if the data sets they represent are found to be cataloged. Uncataloged data sets are not propagated to the new process image. This causes the program that is invoked to run with the same MVS program search order as its invoker.

Example

In the following example, assume that *pgm* was assigned a value earlier in the exec:

```
"forkexecm (pgm) 'hello world'"
```

fpathconf



Function

fpathconf invokes the `fpathconf` callable service to let a program determine the current value of a configurable limit or variable associated with a file or directory. The value is returned in `RETV`.

Parameters

fd The file descriptor (a number) for the file or directory.

name

A numeric value that indicates which configurable limit will be returned. You can specify a numeric value (see Appendix A, “REXX predefined variables,” on page 241) or one of the following predefined variables used to derive the appropriate numeric value:

Variable	Description
<code>PC_ACL</code>	Test if access control lists (ACLs) are supported.

Variable	Description
PC_ACL_MAX	Maximum number of entries allowed in an ACL.
PC_LINK_MAX	Maximum value of a file's link count.
PC_MAX_CANON	Maximum number of bytes in a terminal canonical input line.
PC_MAX_INPUT	Minimum number of bytes for which space will be available in a terminal input queue; therefore, the maximum number of bytes a portable application may require to be typed as input before reading them.
PC_NAME_MAX	Maximum number of bytes in a filename (not a string length; count excludes a terminating null).
PC_PATH_MAX	Maximum number of bytes in a pathname (not a string length; count excludes a terminating null).
PC_PIPE_BUF	Maximum number of bytes that can be written atomically when writing to a pipe.
PC_POSIX_CHOWN_RESTRICTED	Change ownership ("chown" on page 39) function is restricted to a process with appropriate privileges, and to changing the group ID (GID) of a file only to the effective group ID of the process or to one of its supplementary group IDs.
PC_POSIX_NO_TRUNC	Pathname components longer than 255 bytes generate an error.
PC_POSIX_VDISABLE	Terminal attributes maintained by the system can be disabled using this character value.

Usage notes

1. If *name* refers to MAX_CANON, MAX_INPUT, or _POSIX_VDISABLE, the following applies:
 - If *fd* does not refer to a terminal file, the function returns -1 and sets the ERRNO to EINVAL.
2. If *name* refers to NAME_MAX, PATH_MAX, or _POSIX_NO_TRUNC, the following applies:
 - If *fd* does not refer to a directory, the function still returns the requested information using the parent directory of the specified file.
3. If *name* refers to PC_PIPE_BUF, the following applies:
 - If *fd* refers to a pipe or a FIFO, the value returned applies to the referred-to object itself. If *fd* refers to a directory, the value returned applies to any FIFOs that exist or that can be created within the directory. If *fd* refers to any other type of file, the function returns -1 in RETVAL and sets the ERRNO to EINVAL.
4. If *name* refers to PC_LINK_MAX, the following applies:
 - If *fd* refers to a directory, the value returned applies to the directory.

Example

To determine the maximum number of bytes that can be written atomically to the file identified by file descriptor 1:

```
"fpathconf 1 (pc_pipe_buf)"
```

f_setfd



Function

`f_setfd` invokes the `fcntl` callable service to set file descriptor flags.

Parameters

fd The file descriptor (a number) for the file.

close_exec

A numeric value to indicate whether this file descriptor should remain open after an exec:

- 0 indicates that it should remain open.
- 1 indicates that it should be closed.

Example

To set the flags for the file identified by file descriptor 0 and indicate that this file descriptor should remain open during an exec:

```
"f_setfd 0 0"
```

f_setfl



Function

`f_setfl` invokes the `fcntl` callable service to set file status flags.

Parameters

fd The file descriptor (a number) for the file.

flags

A value that sets the file status flags. To specify the information, you can specify a numeric value (see Appendix A, "REXX predefined variables," on page 241) or a predefined variable beginning with `O_` used to derive the appropriate numeric value. The permitted values are:

Variable	Description
<code>O_APPEND</code>	Set offset to EOF on write.
<code>O_NONBLOCK</code>	Do not block an open, a read, or a write on the file (do not wait for terminal input).
<code>O_SYNC</code>	Force synchronous updates.

f_setfl

Example

To set the O_APPEND file status flag for the file identified by file descriptor 1:

```
"f_setfl 1" o_append
```

f_setlk



Function

`f_setlk` invokes the `fcntl` callable service to set or release a lock on part of a file.

Parameters

fd The file descriptor (a number) for the file.

stem

The name of a stem variable that is the flock structure used to set or release the lock. To access the information, you can specify a numeric value (see Appendix A, "REXX predefined variables," on page 241) or the predefined variables beginning with `L_` that derive the appropriate numeric value. For example, `stem.1` and `stem.l_type` are both the lock-type request.

Variable	Description
<code>L_LEN</code>	The length of the byte range that is to be set, cleared, or queried.
<code>L_PID</code>	The process ID of the process holding the blocking lock, if one was found.
<code>L_START</code>	The starting offset byte of the lock that is to be set, cleared, or queried.
<code>L_TYPE</code>	The type of lock being set, cleared, or queried. To specify the information, you can specify a numeric value (see Appendix A, "REXX predefined variables," on page 241) or one of the following predefined variables used to derive the appropriate numeric value:

F_RDLCK

Shared or read lock. This type of lock specifies that the process can read the locked part of the file, and other processes cannot write on that part of the file in the meantime. A process can change a held write lock, or any part of it, to a read lock, thereby making it available for other processes to read. Multiple processes can have read locks on the same part of a file simultaneously. To establish a read lock, a process must have the file accessed for reading.

Variable	Description
	F_WRLCK Exclusive or write lock. This type of lock indicates that the process can write on the locked part of the file, without interference from other processes. If one process puts a write lock on part of a file, no other process can establish a read lock or write lock on that same part of the file. A process cannot put a write lock on part of a file if there is already a read lock on an overlapping part of the file, unless that process is the only owner of that overlapping read lock. In such a case, the read lock on the overlapping section is replaced by the write lock being requested. To establish a write lock, a process must have the file accessed for writing.
L_TYPE	F_UNLCK Unlock. This is used to unlock all locks held on the given range by the requesting process.
L_WHENCE	The flag for the starting offset. To specify the information, you can specify a numeric value (see Appendix A, "REXX predefined variables," on page 241) or one of the predefined variables beginning with SEEK_ used to derive the appropriate numeric value. Valid values are: SEEK_CUR The current offset SEEK_END The end of the file SEEK_SET The specified offset

Usage notes

If the lock cannot be obtained, a RETVAL of -1 is returned along with an appropriate ERRNO and ERRNOJR. You can also use F_SETLK to release locks already held, by setting *l_type* to F_UNLCK.

Multiple lock requests: A process can have several locks on a file simultaneously, but it can have only one type of lock set on any given byte. Therefore, if a process puts a new lock on part of a file that it had previously locked, the process has only one lock on that part of the file and the lock type is the one given by the most recent locking operation.

Releasing locks: When an f_setlk or f_setlkw request is made to unlock a byte region of a file, all locks held by that process within the specified region are released. In other words, each byte specified on an unlock request is freed from any lock that is held against it by the requesting process.

All of a process's locks on a file are removed when the process closes a file descriptor for that file. Locks are not inherited by child processes created with the fork service.

Advisory locking: All locks are advisory only. Processes can use locks to inform each other that they want to protect parts of a file, but locks do not prevent I/O on the locked parts. A process that has appropriate permissions on a file can perform

f_setlk

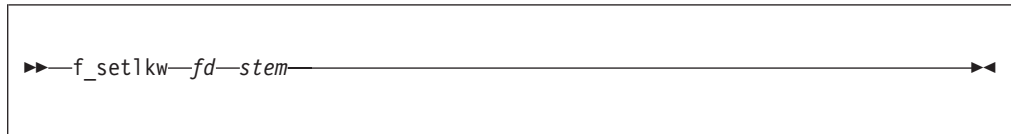
whatever I/O it chooses, regardless of which locks are set. Therefore, file locking is only a convention, and it works only when all processes respect the convention.

Example

The following example locks a 40-byte record (*rec*). Assume that *rec* was assigned a value earlier in the exec:

```
lock.l_len=40
lock.l_start=rec*40
lock.l_type=f_wrlck
lock.l_whence=seek_set
"f_setlk (fd) lock."
```

f_setlkw



Function

f_setlkw invokes the fcntl callable service to set or release a lock on part of a file and, if another process has a lock on some or all of the requested range, wait until the specified range is free and the request can be completed.

fd The file descriptor (a number) for the file.

stem

The name of a stem variable that is the flock structure used to set or release the lock. To specify the lock information, you can specify a numeric value (see Appendix A, "REXX predefined variables," on page 241) or the predefined variables beginning with L_ used to derive the appropriate numeric value. For example, *stem.1* and *stem.l_type* are both the lock-type request.

Variable	Description
L_LEN	The length of the byte range that is to be set, cleared, or queried.
L_PID	The process ID of the process holding the blocking lock, if one was found.
L_START	The starting offset byte of the lock that is to be set, cleared, or queried.
L_TYPE	The type of lock being set, cleared, or queried. To specify the information, you can specify a numeric value (see Appendix A, "REXX predefined variables," on page 241) or one of the following predefined variables used to derive the appropriate numeric value:

Variable	Description
	<p>F_RDLCK Shared or read lock. This type of lock specifies that the process can read the locked part of the file, and other processes cannot write on that part of the file in the meantime. A process can change a held write lock, or any part of it, to a read lock, thereby making it available for other processes to read. Multiple processes can have read locks on the same part of a file simultaneously. To establish a read lock, a process must have the file accessed for reading.</p>
	<p>F_WRLCK Exclusive or write lock. This type of lock indicates that the process can write on the locked part of the file, without interference from other processes. If one process puts a write lock on part of a file, no other process can establish a read lock or write lock on that same part of the file. A process cannot put a write lock on part of a file if there is already a read lock on an overlapping part of the file, unless that process is the only owner of that overlapping read lock. In such a case, the read lock on the overlapping section is replaced by the write lock being requested. To establish a write lock, a process must have the file accessed for writing.</p>
L_TYPE	<p>F_UNLCK Unlock. This is used to unlock all locks held on the given range by the requesting process.</p>
L_WHENCE	<p>The flag for the starting offset. To specify the information, you can specify a numeric value (see Appendix A, "REXX predefined variables," on page 241) or one of the predefined variables beginning with SEEK_ used to derive the appropriate numeric value. Valid values are:</p> <p>SEEK_CUR The current offset</p> <p>SEEK_END The end of the file</p> <p>SEEK_SET The specified offset</p>

Usage notes

If the lock cannot be obtained because another process has a lock on all or part of the requested range, the **f_setlkw** request waits until the specified range becomes free and the request can be completed. You can also use **f_setlkw** to release locks already held, by setting *l_type* to F_UNLCK.

If a signal interrupts a call to the fcntl service while it is waiting in an **f_setlkw** operation, the function returns with a RETVAL of -1 and an ERRNO of EINTR.

f_setlkw operations have the potential for encountering deadlocks. This happens when process A is waiting for process B to unlock a region, and B is waiting for A

f_setlkw

to unlock a different region. If the system detects that a `f_setlkw` might cause a deadlock, the `fcntl` service returns with a `RETV` of -1 and an `ERRNO` of `EDEADLK`.

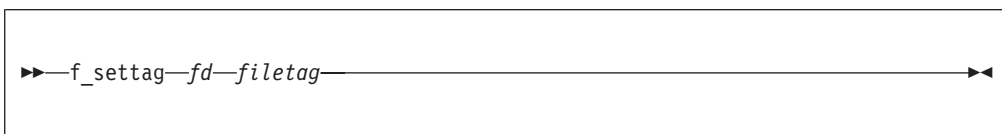
See “`f_setlkw`” on page 60 for more information about locks.

Example

The following example locks a 40-byte record (`rec`). Assume that `rec` was assigned a value earlier in the exec:

```
lock.l_len=40
lock.l_start=rec*40
lock.l_type=f_wrlck
lock.l_whence=seek_set
"f_setlkw (fd) lock."
```

f_settag



Function

`f_settag` can be used to directly control basic file tagging for an opened file. The setting of the file tag may be immediate or deferred until the first write, depending upon the input parameters.

fd The file descriptor (a number) for the file. It must be a regular file, FIFO, or character special file.

filetag

The name of a 4-byte hex variable describing the file tag. The `CCSID` (coded character set identifier) of the file tag occupies the first 2 bytes. For more information, see `BPXYSTAT` — Map the Response Structure for `stat` in *z/OS UNIX System Services Programming: Assembler Callable Services Reference*, or the C header file, `stat.h` in *z/OS XL C/C++ Runtime Library Reference*.

Usage notes

- When the file is `/dev/null`, `/dev/zero`, `/dev/random`, or `/dev/urandom`, the file tag is not hardened to disk.

Example

The following example sets a file tag for an opened file to ASCII ISO88591-1, with the text conversion flag on as well:

```
tag = '03338000'x
"f_settag (fd) tag"
```

fstat

```
►► fstat fd stem ◄◄
```

Function

fstat invokes the fstat callable service to get status information about a file that is identified by its file descriptor.

Parameters

fd The file descriptor (a number) for the file.

stem

The name of a stem variable used to return the status information. Upon return, *stem.0* contains the number of variables that are returned. To access the status values, you can use a numeric value (see Appendix A, “REXX predefined variables,” on page 241) or any of the predefined variables that begin with *ST_*. For example, *stem.9* and *stem.st_atime* are both the request for the time of last access.

Variable	Description
ST_AAUDIT	Auditor audit information
ST_ACCESSACL	1 if there is an access ACL (access control list)
ST_ATIME	Time of last access
ST_AUDITID	RACF File ID for auditing
ST_BLKSIZE	File block size
ST_BLOCKS	Blocks allocated
ST_CCSID	Coded character set ID; first 4 characters are the file tag
ST_CRTIME	File creation time
ST_CTIME	Time of last file status change
ST_DEV	Device ID of the file
ST_DMODELACL	1 if there is a directory model access control list (ACL)
ST_EXTLINK	External symbolic link flag, set to 0 or 1
ST_FID	File identifier

Variable	Description
ST_FILEFMT	Format of the file. To specify the format, you can specify a numeric value (see Appendix A, "REXX predefined variables," on page 241) or one of the following predefined variables used to derive the appropriate numeric value: S_FFBINARY Binary data S_FFCR Text data delimited by a carriage return character S_FFCRLF Text data delimited by carriage return and line feed characters S_FFCRNL A text file with lines delimited by carriage-return and newline characters. S_FFLF Text data delimited by a line feed character S_FFLFCR Text data delimited by a line feed and carriage return characters S_FFNA Text data with the file format not specified S_FFNL Text data delimited by a newline character S_FFRECORD File data consisting of records with prefixes. The record prefix contains the length of the record that follows.
ST_FMODELACL	1 if there is a file model ACL
ST_GENVALUE	General attribute values
ST_GID	Group ID of the group of the file
ST_INO	File serial number
ST_MAJOR	Major number for a character special file
ST_MINOR	Minor number for a character special file
ST_MODE	File mode, permission bits only
ST_MTIME	Time of last data modification
ST_NLINK	Number of links
ST_RUNTIME	File backup time stamp (reference time)
ST_SETGID	Set Group ID on execution flag, set to 0 or 1
ST_SETUID	Set User ID on execution flag, set to 0 or 1
ST_SIZE	File size for a regular file, in bytes. If file size exceeds 2 ³¹ -1 bytes, size is expressed in megabytes, using an M (for example, 3123M).
ST_SECLABEL	Security Label
ST_STICKY	Sticky bit flag (keep loaded executable in storage), set to 0 or 1

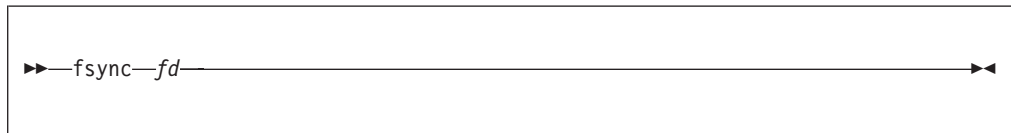
fstatvfs

Variable	Description
STFS_AVAIL	Space available to unprivileged users in block-size units.
STFS_BFREE	Total number of free blocks.
STFS_BLOCKSIZE	Block size.
STFS_FAVAIL	Number of free file nodes available to unprivileged users.
STFS_FFREET	Total number of free file nodes.
STFS_FILES	Total number of file nodes in the file system.
STFS_FRSIZE	Fundamental file system block size.
STFS_FSID	File system ID set by the logical file system.
STFS_INUSE	Allocated space in block-size units.
STFS_INVARSEC	Number of seconds the file system will remain unchanged.
STFS_NAMEMAX	Maximum length of file name.
STFS_NOSEC	Mount data set with no security bit.
STFS_NOSUID	SETUID and SETGID are not supported.
STFS_RDONLY	File system is read-only.
STFS_TOTAL	Total space in block-size units.

Example

In the following example, assume that *fd* was assigned a value earlier in the exec:
"fstatvfs" fd st.

fsync



Function

fsync invokes the fsync callable service to write changes on the direct access storage device that holds the file identified by the file descriptor.

Parameters

fd The file descriptor (a number) for the file.

Usage notes

Upon return from a successful call, all updates have been saved on the direct access storage that holds the file.

Example

To invoke **fsync** for file descriptor 1:

```
"fsync 1"
```

ftrunc

►► ftrunc fd file_size ◀◀

Function

ftrunc invokes the `ftrunc` callable service to change the size of the file identified by the file descriptor.

Parameters

fd The file descriptor (a number) for the file.

file_size

The new size of the file, in bytes.

Usage notes

1. The `ftrunc` service changes the file size to *file_size* bytes, beginning at the first byte of the file. If the file was previously larger than *file_size*, all data from *file_size* to the original end of the file is removed. If the file was previously shorter than *file_size*, bytes between the old and new lengths are read as zeros.
2. Full blocks are returned to the file system so that they can be used again, and the file size is changed to the lesser of *file_size* or the current length of the file.
3. The file offset is not changed.

Example

In the following example, assume that *fd* was assigned a value earlier in the exec. To truncate *fd* to 0 bytes:

```
"ftrunc" fd 0
```

getcwd

►► getcwd variable ◀◀

Function

getcwd invokes the `getcwd` callable service to get the pathname of the working directory.

Parameters

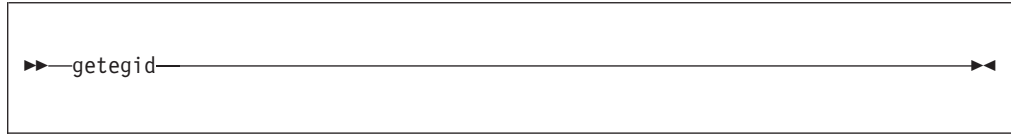
variable

The name of the variable where the pathname of the working directory is to be stored.

Example

To get the pathname of the working directory:
"getcwd cwd"

getegid



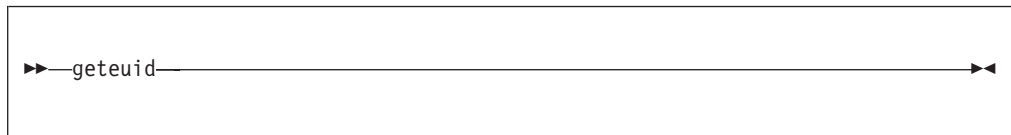
Function

getegid invokes the getegid callable service to get the effective group ID (GID) of the calling process.

Usage notes

1. The effective GID is returned in RETVAL.
2. If the service fails, the process abends.

geteuid



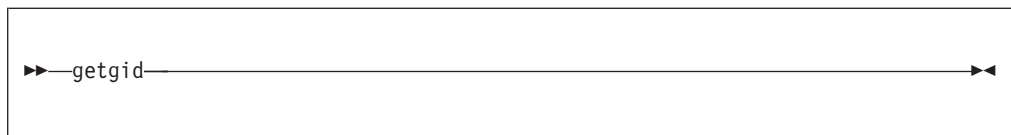
Function

geteuid invokes the geteuid callable service to get the effective user ID (UID) of the calling process.

Usage notes

1. The effective UID is returned in RETVAL.
2. If the service fails, the process abends.

getgid



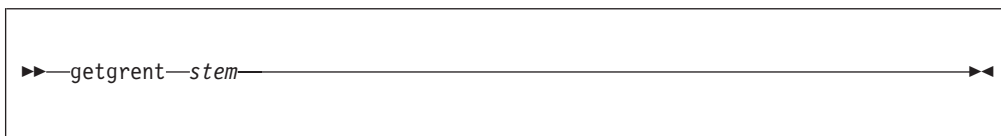
Function

getgid invokes the getgid callable service to get the real group ID (GID) of the calling process.

Usage notes

1. The GID is returned in RETVAL.
2. If the service fails, the process abends.

getgrent



Function

getgrent invokes the `getgrent` callable service to retrieve a group database entry.

stem

The name of a stem variable used to return the information. Upon return, `stem.0` contains the number of variables returned. To access the status values, you can specify a numeric value (see Appendix A, “REXX predefined variables,” on page 241) or the predefined variable beginning with `GR_` used to derive the appropriate numeric value. For example, you can specify `stem.2` or `stem.gr_gid` to access the group ID:

Variable	Description
<code>GR_GID</code>	The group ID
<code>GR_MEM</code>	The stem index of the first member name returned
<code>GR_MEMBERS</code>	The number of members returned
<code>GR_NAME</code>	The name of the group

Usage notes

1. The service is intended to be used to search the group database sequentially. The first call to this service from a given task returns the first group entry in the group database. Subsequent calls from the same task return the next group entry found, until no more entries exist, at which time a `RETV` of 0 is returned.
2. The `setgrent` service can be used to reset this sequential search.

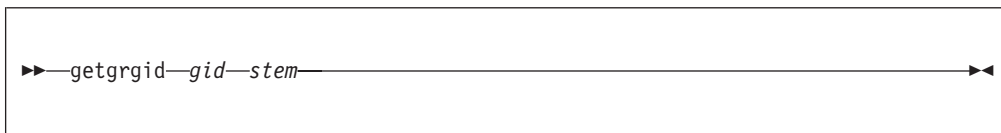
Example

To list all groups in the group data base:

```
"getgrent gr."
```

For a complete example, see “List all users and groups” on page 162.

getgrgid



Function

getgrgid invokes the `getgrgid` callable service to get information about a group and its members; the group is identified by its group ID (GID).

getgrgid

Parameters

gid

A numeric value.

stem

The name of a stem variable used to return the information. Upon return, *stem.0* contains the number of variables returned. To access the status values, you can specify a numeric value (see Appendix A, “REXX predefined variables,” on page 241) or the predefined variable beginning with GR_ used to derive the appropriate numeric value. For example, you can specify *stem.2* or *stem.gr_gid* to access the group ID:

Variable	Description
GR_GID	The group ID
GR_MEM	The stem index of the first member name returned
GR_MEMBERS	The number of members returned
GR_NAME	The name of the group

Usage notes

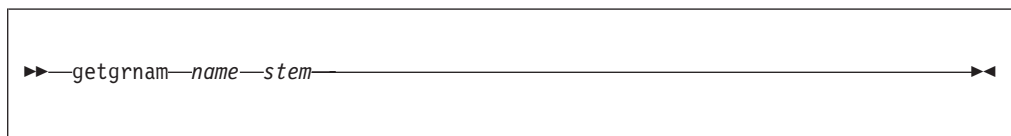
A RETVAL greater than zero indicates success. A RETVAL of -1 or 0 indicates failure. A RETVAL of 0 has an ERRNOJR, but no ERRNO.

Example

In the following example, assume that *gid* was assigned a value earlier in the exec. To get a list of names connected with a group ID:

```
"getgrgid (gid) gr."  
say 'Users connected to group number' gid ':'  
do i=gr_mem to gr.0  
  say gr.i  
end
```

getgrnam



Function

getgrnam invokes the getgrnam callable service to get information about a group and its members. The group is identified by its name.

Parameters

name

A string that specifies the group name as defined to the system.

stem

The name of a stem variable used to return the information. Upon return, *stem.0* contains the number of variables returned. To access the status values, you can specify a numeric value (see Appendix A, “REXX predefined

variables," on page 241) or the predefined variable beginning with GR_ used to derive the appropriate numeric value. For example, you can specify *stem.2* or *stem.gr_gid* to access the group ID:

Variable	Description
GR_GID	The group ID
GR_MEM	The stem index of the first member name returned
GR_MEMBERS	The number of members returned
GR_NAME	The name of the group

Usage notes

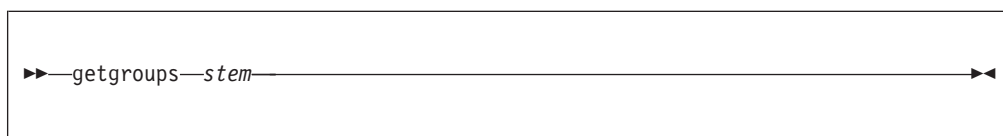
1. A RETVAL greater than zero indicates success. A RETVAL of -1 or 0 indicates failure. A RETVAL of 0 has an ERRNOJR, but no ERRNO.
2. The return values point to data that can change or disappear after the next getgrnam or getgrgid call from that task. Each task manages its own storage separately. Move data to your own dynamic storage if you need it for future reference.
3. The storage is key 0 non-fetch-protected storage that is managed by z/OS UNIX System Services.

Example

To get information about the group named SYS1:

```
"getgrnam SYS1 gr."
say 'Users connected to group SYS1:'
do i=gr_mem to gr.0
  say gr.i
end
```

getgroups



Function

getgroups invokes the getgroups callable service to get the number of supplementary group IDs (GIDs) for the calling process and a list of those supplementary group IDs.

Parameters

stem

The name of a stem variable used to return the information. Upon return, *stem.0* contains the number of variables returned. *stem.1* to *stem.n* (where *n* is the number of variables returned) each contain a group ID.

Usage notes

A RETVAL greater than zero indicates success. A RETVAL of -1 or 0 indicates failure. A RETVAL of 0 has an ERRNOJR, but no ERRNO.

getgroups

Example

To invoke getgroups:
"getgroups grps."

getgroupsbyname

```
►►—getgroupsbyname—name—stem—————◄◄
```

Function

getgroupsbyname invokes the getgroupsbyname callable service to get the number of supplementary group IDs (GIDs) for a specified user name and, optionally, get a list of those supplementary group IDs.

Parameters

name

A string that specifies the name of the user as defined to the system.

stem

The name of a stem variable used to return the information. Upon return, *stem.0* contains the number of variables returned. *stem.1* to *stem.n* (where *n* is the number of variables returned) each contain a supplementary group ID for *name*.

Usage notes

A RETVAL greater than zero indicates success. A RETVAL of -1 or 0 indicates failure. A RETVAL of 0 has an ERRNOJR, but no ERRNO.

Example

To get the number of supplementary group IDs for the user MEGA:
"getgroupsbyname MEGA supgrp."

getlogin

```
►►—getlogin—variable—————◄◄
```

Function

getlogin invokes the getlogin callable service to get the user login name associated with the calling process.

Parameters

variable

The name of the variable in which the login name is returned

Usage notes

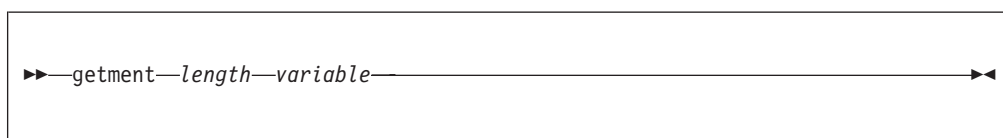
If the service fails, the process abends.

Example

To invoke getlogin and store the login name in the variable *myid*:

```
"getlogin myid"
```

getmnt



Function

getmnt invokes the `w_getmntent` callable service to get information about the mounted file systems and return the information in the format used by the z/OS UNIX callable services. Alternatively, you can use the `getmntent` syscall command to format the mount entries in a stem.

Parameters

length

The size of the specified variable. If the length is 0, RETVAL contains the total number of mounted file systems; otherwise, RETVAL contains the number of entries returned.

variable

The name of the buffer where the information about the mount entries is to be stored. Clear the buffer on the first call and do not alter it between calls.

Usage notes

1. Before a program calls **getmnt** for the first time, the variable should be dropped, set to blanks, or set to nulls.
2. If more than one call is made to **getmnt**, use the same variable on each call, because part of the information returned in the variable tells the file system where to continue retrieving its information.
3. **getmnt** normally returns information about as many file systems as are mounted, or as many as fit in the passed variable. The number of entries contained in the variable is returned. The caller must have a variable large enough to receive information about at least a single mount entry with each call. If a zero-length variable is passed, no information is returned, but the return value contains the total number of mounted file systems. This value could then be used to get enough storage to retrieve information on all these file systems in one additional call.

getment

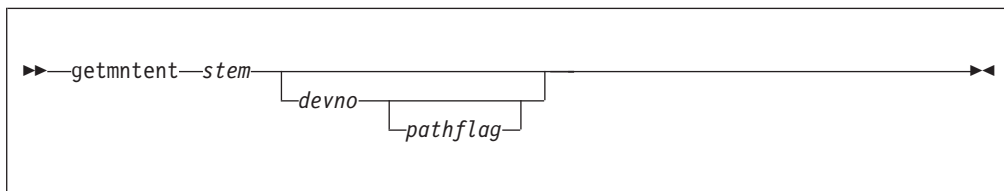
4. You could also retrieve all mount entries by setting up a loop that continues to call **getment** until a return value of either -1 (in an error) or 0 (no more entries found) is returned.

Example

In the following example, assume that *buf* was assigned a value earlier in the exec. This example returns the number of mounted file systems in RETVAL:

```
"getment 0 buf"
```

getmntent



Function

getmntent invokes the w_getmntent callable service to get information about the mounted file systems, or a specific mounted file system, and return the formatted information in a stem variable.

Parameters

stem

The name of a stem variable used to return the mount table information. The stem has two dimensions: the field name, followed by a period and the number of the mount table entry. For example, you can access the file system name for the first entry in the mount table as *stem.mnte_fsname.1*.

For the field name, you can specify a numeric value (see Appendix A, “REXX predefined variables,” on page 241) or the predefined variable beginning with MNT_ used to derive the appropriate numeric value. For example, you could use *stem.mnte_fsname.1* or *stem.6.1* to access the file system name for the first entry:

Variable	Description
MNTE_AGGNAME	The name of the zFS aggregate data set.
MNTE_BYTESREADHW	The number of bytes read (high-word value).
MNTE_BYTESREADLW	The number of bytes read (low-word value).
MNTE_BYTESWRITTENHW	The number of bytes written (high-word value).
MNTE_BYTESWRITTENLW	The number of bytes written (low-word value).
MNTE_DD	The ddname specified on the mount.
MNTE_DEV	The device ID of the file system.
MNTE_DIRIBC	The number of directory I/O blocks.
MNTE_FILETAG	The file tag.
MNTE_FROMSYS	The file systems are to be moved from here.
MNTE_FSNAME	The file system name that was specified on the mount. This statement applies only to HFS and zFS. The file system name is the name of the MVS data set that contains the HFS or zFS file system.
MNTE_FSTYPE	The file system type.

Variable	Description
MNTE_MODE	The file system type mount method. MNTE_MODE contains a decimal number that corresponds to the value as defined in the field mntentfsmode in the BPXYMNTE mapping macro. The REXX predefined variables for this field, as well as other fields on this interface, are a subset of what might be returned by the getmntent service. You can specify a numeric value (see Appendix A, “REXX predefined variables,” on page 241) or one of the following predefined variables used to derive the appropriate numeric value. For example, there is no predefined rexx variable corresponding to the bit position defined by mntentfssynchronly in the mntentfsmode.
MNT_MODE_RDWR	File system mounted read-write.
MNT_MODE_RDONLY	File system mounted read-only.
MNT_MODE_AUNMOUNT	The file system can be unmounted if the system's owner crashes.
MNT_MODE_CLIENT	The file system is a client.
MNT_MODE_EXPORT	The file system exported by DFS.
MNT_MODE_NOAUTOMOVE	Automove is not allowed.
MNT_MODE_NOSEC	No security checks are enforced.
MNT_MODE_NOSETID	SetUID is not permitted for files in this filesystem.
MNT_MODE_SECACL	ACLs are supported by the security product
MNTE_PARDEV	The ST_DEV of the parent file system.
MNTE_PARM	The parameter specified with mount() .
MNTE_PATH	The mountpoint pathname.
MNTE_PFSSTATUSNORMAL	Normal status string returned by the physical file system.
MNTE_PFSSTATUSEXCP	Exception status string returned by the physical file system.
MNTE_ROSECLABEL	Default security label for objects in a read-only file system.
MNTE_QJOBNAME	The job name of the quiesce requestor.
MNTE_QPID	The PID of the quiesce requestor.
MNTE_QSYSNAME	The name of the quiesce system name.
MNTE_READCT	The number of reads from fileys.
MNTE_READIBC	The number of read I/O blocks.
MNTE_RFLAGS	The request flags.
MNTE_ROOTINO	The inode of the mountpoint.
MNTE_ROSECLABEL	The read-only SECLABEL.

getmntent

Variable	Description
MNTE_STATUS	The status of the file system. To specify the information, you can specify a numeric value (see Appendix A, "REXX predefined variables," on page 241) or one of the following predefined variables used to derive the appropriate numeric value: MNT_ASYNCHMOUNT Asynchronous mount in progress for this file system. MNT_FILEACTIVE File system is active. MNT_FILEDEAD File system is dead. MNT_FILEDRAIN File system is being unmounted with the drain option. MNT_FILEFORCE File system is being unmounted with the force option. MNT_FILEIMMED File system is being unmounted with the immediate option. MNT_FILENORM File system is being unmounted with the normal option. MNT_FILERESET File system is being reset. MNT_IMMEDTRIED File system unmount with the immediate option failed. MNT_MOUNTINPROGRESS Mount in progress for this file system MNT_QUIESCED File system is quiesced.
MNTE_STATUS2	The status of the file system.
MNTE_SUCCESS	Successful moves.
MNTE_SYSLIST	A list of system names.
MNTE_SYSNAME	The name of the owning system.
MNTE_TYPE	The file system type.
MNTE_UID	The effective UID of the nonprivileged user who mounted this file system. MNTE_UID is always 0 for the file system mounted by the privileged user.
MNTE_WRITECT	The number of writes done.
MNTE_WRITEIBC	The number of write I/O blocks.

devno

An optional parameter, this is the device number for a specific file system for which you want the mount information. Specifying 0 is the equivalent of not specifying a device number.

pathflag

An optional parameter, this specifies the returned path names.

0 Returns the current path name.

1 Returns up to 64 bytes of the path name at the time of the mount.

Example

Show all mounted file systems:


```

address syscall 'getmntent mounts.'
do i=1 to mounts.0
  say mounts.mnte_fsname.i
end

```

getpgrp



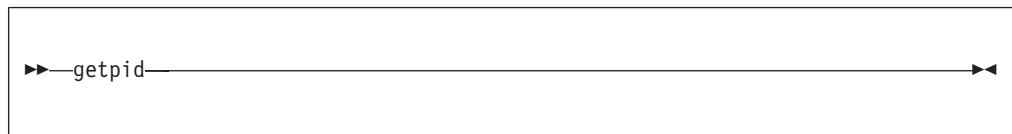
Function

getpgrp invokes the getpgrp callable service to get the process group ID (PGID) of the calling process.

Usage notes

1. The PGID for the calling process is returned in RETVAL.
2. If the service fails, the process abends.

getpid



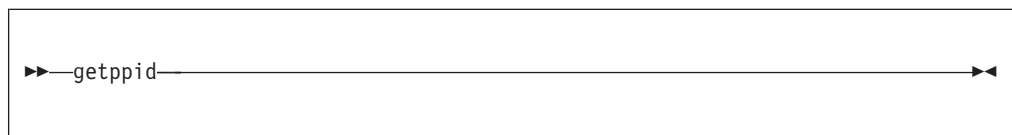
Function

getpid invokes the getpid callable service to get the process ID (PID) of the calling process.

Usage notes

1. The PID for the calling process is returned in RETVAL.
2. If the service fails, the process abends.

getppid



Function

getppid invokes the getppid callable service to get the parent process ID (PPID) of the calling process.

Usage notes

1. The parent PID for the calling process is returned in RETVAL.
2. If the service fails, the process abends.

getpsent



Function

getpsent invokes the `w_getpsent` callable service to provide data describing the status of all the processes to which you are authorized. The data is formatted in a stem. The `PS_` variables, or their numeric equivalents, are used to access the fields.

Parameters

stem

Upon return, *stem.0* contains the number of processes for which information is returned. *stem.1* through *stem.n* (where *n* is the number of entries returned) each contain process information for the *n*th process.

You can use the predefined variables that begin with `PS_` or their equivalent numeric values (see Appendix A, “REXX predefined variables,” on page 241) to access the information. For example, *stem.1.ps_pid* is the process ID for the first process returned.

Variable	Description
<code>PS_CMD</code>	Command
<code>PS_CONTTY</code>	Controlling tty
<code>PS_EGID</code>	Effective group ID
<code>PS_EUID</code>	Effective user ID
<code>PS_FGPID</code>	Foreground process group ID
<code>PS_MAXVNODES</code>	Maximum number of vnode tokens allowed
<code>PS_PATH</code>	Pathname
<code>PS_PGPID</code>	Process Group ID
<code>PS_PID</code>	Process ID
<code>PS_PPID</code>	Parent Process ID
<code>PS_RGID</code>	Real group ID
<code>PS_RUID</code>	Real user ID
<code>PS_SERVERFLAGS</code>	Server flags
<code>PS_SERVERNAME</code>	Server name supplied on registration
<code>PS_SERVERTYPE</code>	Server type (File=1; Lock=2)
<code>PS_SGID</code>	Saved set group ID
<code>PS_SID</code>	Session ID (leader)
<code>PS_SIZE</code>	Total size
<code>PS_STARTTIME</code>	Starting time, in POSIX format (seconds since the Epoch, 00:00:00 on 1 January 1970)
<code>PS_STAT</code>	Process status

Variable	Description
PS_STATE	Process state This value can be expressed as one of the following predefined variables or as an alphabetic value (see Appendix A, "REXX predefined variables," on page 241):
PS_CHILD	Waiting for a child process
PS_FORK	fork() a new process
PS_FREEZE	QUIESCEFREEZE
PS_MSGRCV	IPC MSGRCV WAIT
PS_MSGSND	IPC MSGSND WAIT
PS_PAUSE	MVSPAUSE
PS_QUIESCE	Quiesce termination wait
PS_RUN	Running, not in kernel wait
PS_SEMWT	IPC SEMOP WAIT
PS_SLEEP	sleep() issued
PS_WAITC	Communication kernel wait
PS_WAITF	File system kernel wait
PS_WAITO	Other kernel wait
PS_ZOMBIE	Process cancelled
PS_ZOMBIE2	Process terminated yet still the session or process group leader
PS_SUID	Saved set user ID
PS_SYSTIME	System CPU time, a value of the type clock_t, which needs to be divided by sysconf(_SC_CLK_TCK) to convert it to seconds. For z/OS UNIX, this value is expressed in hundredths of a second.
PS_USERTIME	User CPU time, a value of the type clock_t, which needs to be divided by sysconf(_SC_CLK_TCK) to convert it to seconds. For z/OS UNIX, this value is expressed in hundredths of a second.
PS_VNODECOUNT	The current number of vnode tokens

Usage notes

1. Information is returned for only those processes for which RACF allows the user access based on effective user ID, real user ID, or saved set user ID.
2. PS_STARTTIME is in seconds since the Epoch (00:00:00 on 1 January 1970).
3. PS_USERTIME and PS_SYSTIME are task-elapsed times in 1/100ths of seconds.
4. PS_SYSTIME reports the system CPU time consumed for the address space that the process is running in. When only one process is running in the address space, this time represents the accumulated system CPU time for that process. However, when more than one process is running in an address space, the information that is returned is actually the accumulated system CPU time consumed by all of the work running in the address space.

Example

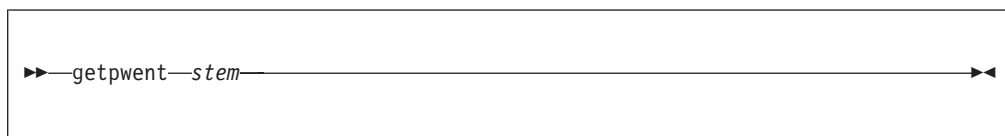
This exec will produce output similar to the `ps -A` shell command, displaying information on all accessible processes:

```

/* rexx */
address syscall
say right('PID',12) left('TTY',10) '   TIME' 'COMMAND'
ps.0=0
'getpsent ps.'                               /* get process data          */
do i=1 to ps.0                                 /* process each entry returned */
  t=(ps.i.ps_usertime + 50) % 100             /* change time to seconds    */
  'gmtime (t) gm.'                             /* convert to usable format   */
  if gm.tm_hour=0 then                          /* set hours: samp ignores day */
    h='   '
  else
    h=right(gm.tm_hour,2,0)':'
  m=right(gm.tm_min,2,0)':'                    /* set minutes                */
  parse value reverse(ps.i.ps_contty),
    with tty '/'                               /* get tty filename           */
  tty=reverse(tty)
  say right(ps.i.ps_pid,12),                    /* display process id         */
  say right(ps.i.ps_pid,12),                    /* display process id         */
    left(tty,10),                               /* display controlling tty    */
    h || m || right(gm.tm_sec,2,0),             /* display process time       */
    ps.i.ps_cmd                                  /* display command            */
end
return 0

```

getpwent



Function

`getpwent` invokes the `getpwent` callable service to retrieve a user database entry.

Parameters

stem

The name of a stem variable used to return the information. Upon return, `stem.0` contains the number of variables returned. You can use a numeric value (see Appendix A, “REXX predefined variables,” on page 241) or the predefined variables beginning with `PW_` to access the values:

Variable	Description
<code>PW_DIR</code>	The initial working directory
<code>PW_GID</code>	The group ID
<code>PW_NAME</code>	The TSO/E user ID
<code>PW_SHELL</code>	The name of the initial user program.
<code>PW_UID</code>	The user ID (UID) as defined to RACF.

Usage notes

1. The service is intended to be used to search the user database sequentially. The first call to this service from a given task returns the first user entry in the user

database. Subsequent calls from the same task return the next user entry found, until no more entries exist, at which time a RETVAL of 0 is returned.

2. The setpwent service can be used to reset this sequential search.

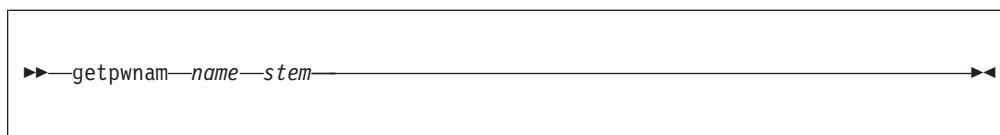
Example

To list all users in the user database:

```
"getpwent pw."
```

For a complete example, see "List all users and groups" on page 162.

getpwnam



Function

getpwnam invokes the getpwnam callable service to get information about a user, identified by user name.

Parameters

name

The user name as defined to the system.

stem

The name of a stem variable used to return the information. Upon return, *stem.0* contains the number of variables returned. To access the information, you can specify a numeric value (see Appendix A, "REXX predefined variables," on page 241) or the predefined variable beginning with PW_ used to derive the appropriate numeric value. For example, to access the name of the user's initial working directory, you can specify *stem.4* or *stem.pw_dir*:

Variable	Description
PW_DIR	The initial working directory
PW_GID	The group ID
PW_NAME	The TSO/E user ID
PW_SHELL	The name of the initial user program.
PW_UID	The user ID (UID) as defined to RACF.

Usage notes

1. A RETVAL greater than zero indicates success. A RETVAL of -1 or 0 indicates failure. A RETVAL of 0 has an ERRNOJR, but no ERRNO.
2. If an entry for the specified *name* is not found in the user database, the RETVAL is 0.

Example

To get information about the user JANET:

```
"getpwnam JANET pw."
```

getpwuid

```
▶▶—getpwuid—uid—stem—◀◀
```

Function

getpwuid invokes the getpwuid callable service to get information about a user, identified by UID.

Parameters

uid

A numeric value that is the user's UID as defined to the system.

stem

The name of a stem variable used to return the information. Upon return, *stem.0* contains the number of variables returned. You can use a numeric value (see Appendix A, "REXX predefined variables," on page 241) or the predefined variables beginning with PW_ to access the values:

Variable	Description
PW_DIR	The initial working directory
PW_GID	The group ID
PW_NAME	The TSO/E user ID
PW_SHELL	The name of the initial user program.
PW_UID	The user ID (UID) as defined to RACF.

Usage notes

A RETVAL greater than zero indicates success. A RETVAL of -1 or 0 indicates failure. A RETVAL of 0 has an ERRNOJR, but no ERRNO.

Example

To get information about the user with UID 42:

```
"getpwuid 42 pw."
```

getrlimit

```
▶▶—getrlimit—resource—stem—◀◀
```

Function

getrlimit invokes the getrlimit callable service to get the maximum and current resource limits for the calling process.

Parameters

resource

The resource whose limit is being requested. Resources that have limits with values greater than RLIM_INFINITY will return values of RLIM_INFINITY.

You can use the predefined variables beginning with RLIMIT_ or their equivalent numeric values to access the limits. (See Appendix A, "REXX predefined variables," on page 241 for the numeric values.)

Variable	Description
RLIMIT_AS	Maximum address space size for a process
RLIMIT_CORE	Maximum size (in bytes) of a core dump created by a process
RLIMIT_CPU	Maximum amount of CPU time (in seconds) used by a process
RLIMIT_FSIZE	Maximum files size (in bytes) created by a process
RLIMIT_NOFILE	Maximum number of open file descriptors for a process

stem

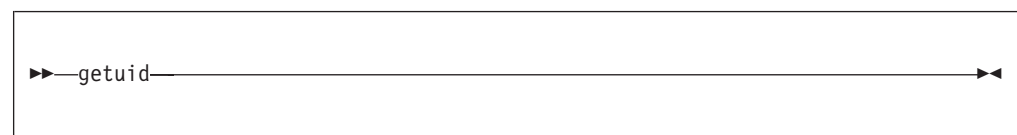
The name of the stem variable used to return the limit. *stem.1* is the first word and *stem.2* is the second word. The first word contains the current limit; the second word contains the maximum limit. The values for each word are dependent on the *resource* specified.

Example

To print the maximum and current limit for number of open files allowed:

```
"getrlimit" rlimit_nofile r.
say 'maximum open limit is' r.2
say 'current open limit is' r.1
```

getuid



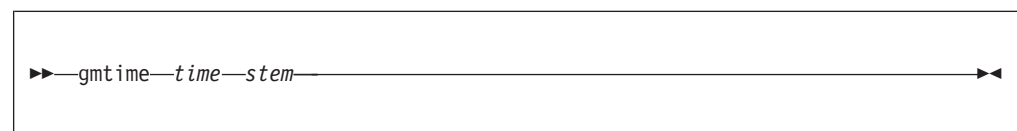
Function

getuid invokes the getuid callable service to get the real user ID of the calling process.

Usage notes

Upon return, RETVAL contains the UID. If the service fails, the process abends.

gmtime



gmtime

Function

gmtime converts time expressed in seconds since Epoch into month, day, and year time format.

Parameters

time

A numeric value, the time expressed as "POSIX time", the number of seconds since the Epoch (00:00:00 on 1 January 1970).

stem

The name of a stem variable used to return the information. Upon return, *stem.0* contains the number of variables returned. To access the status values, you can specify a numeric value (see Appendix A, "REXX predefined variables," on page 241) or the predefined variable beginning with *TM_* used to derive the appropriate numeric value. For example, to access the year, you can specify *stem.6* or *stem.tm_year*:

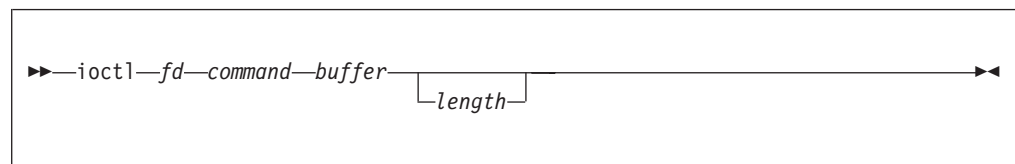
Variable	Description
TM_HOUR	The hour of the day.
TM_ISDST	The daylight saving time flag. This flag is always zero (-1) for Greenwich Mean Time (GMT).
TM_MDAY	The day of the month, 1 to 31.
TM_MIN	The minutes after the hour, 0 to 59.
TM_MON	The months since January, 0 to 11.
TM_SEC	The seconds after the minute, 0 to 59.
TM_WDAY	The days since Sunday, 0 to 6.
TM_YDAY	The days since January 1, 0 to 365.
TM_YEAR	The year.

Example

The *st_ctime* in the following example was set with a stat call:

```
"gmtime" st.st_ctime "tm."
```

ioctl



Function

ioctl invokes the *w_ioctl* service to issue *ioctl* commands to a file.

Restriction: Hierarchical file system files and pipes do not support *ioctl* commands.

Parameters

fd A file descriptor for an open file.

Restriction: REXX does not support *ioctl* commands for sockets.

command

The numeric value for an ioctl command. The commands that can be specified vary by device and are defined by the device driver.

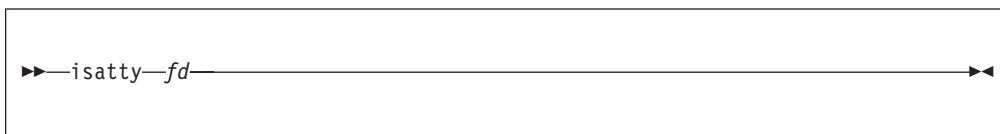
buffer

The name of a buffer containing the argument to be passed to the device driver. The argument is limited to 1024 bytes.

length

An optional numeric value indicating the length of the argument. If length is not specified, the length of the buffer is used.

isatty

**Function**

`isatty` invokes the `isatty` callable service to determine if a file is a terminal.

Parameters

`fd` The file descriptor (a number) for the file.

Usage notes

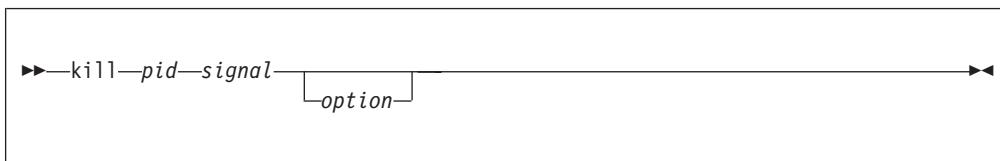
Upon return, `RETV` contains either 0 (not a tty) or 1 (a tty).

Example

To test if file descriptor 0 is a terminal:

```
"isatty 0"
```

kill

**Function**

`kill` invokes the `kill` callable service to send a signal to a process or process group.

Parameters**pid**

A number, the process ID of the process or process group to which the caller wants to send a signal.

signal

The signal to be sent. You can set the value by using a numeric value (see

Appendix A, “REXX predefined variables,” on page 241 or the predefined variable beginning with SIG used to derive the numeric value:

Variable	Description
SIGABND	Abend
SIGABRT	Abnormal termination
SIGALRM	Timeout
SIGBUS	Bus error
SIGCHLD	Child process terminated or stopped
SIGCONT	Continue if stopped
SIGFPE	Erroneous arithmetic operation, such as division by zero or an operation resulting in an overflow
SIGHUP	Hangup detected on the controlling terminal
SIGILL	Termination that cannot be caught or ignored
SIGINT	Interactive attention
SIGIO	Completion of input or output
SIGIOERR	Error on input/output; used by the C runtime library
SIGKILL	Termination that cannot be caught or ignored
SIGPIPE	Write on a pipe with no readers
SIGPOLL	Pollable event
SIGPROF	Profiling timer expired
SIGQUIT	Interactive termination
SIGSEGV	Detection of an incorrect memory reference
SIGSTOP	Stop that cannot be caught or ignored
SIGSYS	Bad system call
SIGTERM	Termination
SIGTRAP	Trap used by the ptrace callable service
SIGTSTP	Interactive stop
SIGTTIN	Read from a controlling terminal attempted by a member of a background process group
SIGTTOU	Write from a controlling terminal attempted by a member of a background process group
SIGURG	High bandwidth data is available at a socket
SIGUSR1	Reserved as application-defined signal 1
SIGUSR2	Reserved as application-defined signal 2
SIGVTALRM	Virtual timer expired
SIGXCPU	CPU time limit exceeded
SIGXFSZ	File size limit exceeded

option

The name of a variable that contains exactly four bytes mapped by the BPXYPPSD macro.

Usage notes

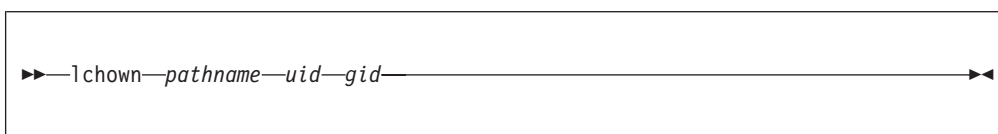
1. A caller can send a signal if the real or effective user ID of the caller is the same as the real or saved set user ID of the intended recipient. A caller can also send signals if it has appropriate privileges.
2. Regardless of user ID, a caller can always send a **SIGCONT** signal to a process that is a member of the same session as the sender.
3. A caller can also send a signal to itself. If the signal is not blocked, at least one pending unblocked signal is delivered to the sender before the service returns control. Provided that no other unblocked signals are pending, the signal delivered is the signal sent.

See “Using the REXX signal services” on page 10 for information on using signal services.

Example

In the following example, assume that *pid* was assigned a value earlier in the exec:
"kill" pid sighup

lchown



Function

lchown invokes the lchown callable service to change the owner or group for a file, directory, or symbolic link.

Parameters

pathname

The pathname for a file, directory, or symbolic link.

uid

The numeric UID for the new owner or the present UID, or -1 if there is no change to the UID.

GID

The numeric GID for the new group or the present GID, or -1 if there is no change to the GID.

Usage notes

1. If lchown's target is a symbolic link, it modifies the ownership of the actual symbolic link file instead of the ownership of the file pointed to by the symbolic link.
2. The lchown service changes the owner UID and owner GID of a file. Only a superuser can change the owner UID of a file.
3. The owner GID of a file can be changed by a superuser, or if a caller meets all of these conditions:
 - The effective UID of the caller matches the file's owner UID.
 - The *uid* value specified in the change request matches the file's owner UID.

lchown

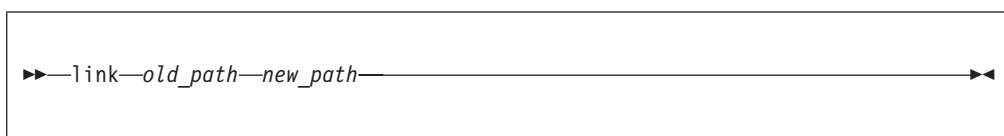
- The *gid* value specified in the change request is the effective GID, or one of the supplementary GIDs, of the caller.
4. The set-user-ID-on-execution and set-group-ID-on-execution permissions of the file mode are automatically turned off.
 5. If the change request is successful, the change time for the file is updated.
 6. Values for both *uid* and *gid* must be specified as they are to be set. If you want to change only one of these values, the other must be set to its present value to remain unchanged.

Example

In the following example, assume that *pathname*, *uid*, and *gid* were assigned a value earlier in the exec:

```
"lchown (pathname) (uid) (gid)"
```

link



Function

link invokes the link callable service to create a hard link to a file (not a directory); that is, it creates a new name for an existing file. The new name does not replace the old one.

Parameters

old_path

A path name, the current name for the file.

new_path

A path name, the new name for the file.

Usage notes

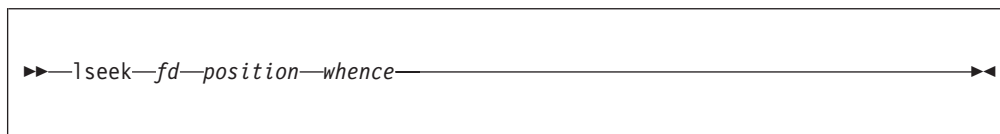
1. The link service creates a link named *new_path* to an existing file named *old_path*. This provides an alternate path name for the existing file, so that the file can be accessed by the old name or the new name. The link can be stored in the same directory as the original file, or in a different directory.
2. The link and the file must be in the same file system.
3. If the link is created successfully, the service increments the link count of the file. The link count shows how many links exist for a file. If the link is not created successfully, the link count is not incremented.
4. Links are allowed only to files, not to directories.
5. If the link is created successfully, the change time of the linked-to file is updated and the change and modification times of the directory that holds the link are updated.

Example

To create the hard link `/usr/bin/grep` to the file `/bin/grep`:

```
"link /bin/grep /usr/bin/grep"
```

lseek



Function

lseek invokes the `lseek` callable service to change the file offset of a file to a new position. The file offset is the position in a file from which data is next read or to which data is next written.

Refer to *z/OS UNIX System Services Programming: Assembler Callable Services Reference* for restrictions when running in a conversion environment.

Parameters

fd The file descriptor (a number) for the file whose offset you want to change. The file descriptor is returned when the file is opened.

position

A number indicating the number of bytes by which you want to change the offset. If the number is unsigned, the offset is moved forward that number of bytes; if the number is preceded by a - (minus sign), the offset is moved backward that number of bytes.

whence

A numeric value that indicates the point from which the offset is calculated. You can specify a numeric value (see Appendix A, "REXX predefined variables," on page 241) or the predefined variable beginning with `SEEK_` used to derive the appropriate numeric value:

Variable	Description
<code>SEEK_CUR</code>	Set the file offset to current offset plus the specified offset.
<code>SEEK_END</code>	Set the file offset to EOF plus the specified offset.
<code>SEEK_SET</code>	Set the file offset to the specified offset.

Usage notes

1. *position* gives the length and direction of the offset change. *whence* states where the change is to start. For example, assume that a file is 2000 bytes long, and that the current file offset is 1000:

Position specified	Whence	New file offset
80	<code>SEEK_CUR</code>	1080
1200	<code>SEEK_SET</code>	1200
-80	<code>SEEK_END</code>	1920
132	<code>SEEK_END</code>	2132

lseek

2. The file offset can be moved beyond the end of the file. If data is written at the new file offset, there will be a gap between the old end of the file and the start of the new data. A request to read data from anywhere within that gap completes successfully, and returns bytes with the value of zero in the buffer and the actual number of bytes read.

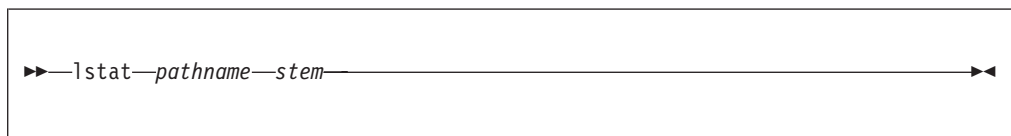
Seeking alone, however, does not extend the file. Only if data is written at the new offset does the length of the file change.

Example

To change the offset of file descriptor *fd* (assuming that it was assigned a value earlier in the exec) to the beginning of the file (offset 0):

```
"lseek" fd 0 seek_set
```

lstat



Function

lstat invokes the **lstat** callable service to obtain status information about a file.

Parameters

pathname

A pathname for the file.

stem

The name of a stem variable used to return the information. Upon return, *stem.0* contains the number of variables returned. To obtain the desired status information, you can use a numeric value (see Appendix A, “REXX predefined variables,” on page 241) or the predefined variables beginning with **ST_** used to derive the numeric value. See “**fstat**” on page 65 for a list of those variables.

Usage notes

1. If the pathname specified is a symbolic link, the status information returned relates to the symbolic link, rather than to the file to which the symbolic link refers.
2. All time values returned are in POSIX format. Time is in seconds since 00:00:00 GMT, January 1, 1970. You can use **gmtime** to convert it to other forms.

Example

In the following example, assume that *pathname* was assigned a value earlier in the exec:

```
"lstat (pathname) st."
```

mkdir

Format

```
▶▶ mkdir pathname mode ▶▶
```

Purpose

mkdir invokes the `mkdir` callable service to create a new, empty directory.

Parameters

pathname

A path name for the directory.

mode

A three- or four-digit number, corresponding to the access permission bits.

Each digit must be in the range 0–7, and at least three digits must be specified.

For more information on permissions, see Appendix B, “Setting permissions for files and directories,” on page 253.

Usage

1. The file permission bits specified in *mode* are modified by the file creation mask of the calling process (see “`umask`” on page 149), and are then used to set the file permission bits of the new directory.
2. The new directory's owner ID is set to the effective user ID (UID) of the calling process.
3. The `mkdir` service sets the access, change, and modification times for the new directory. It also sets the change and modification times for the directory that contains the new directory.

Examples

In the following example, assume that *pathname* and *mode* were assigned a value earlier in the exec:

```
"mkdir (pathname)" mode
```

mkfifo

Format

```
▶▶ mkfifo pathname mode ▶▶
```

Purpose

mkfifo invokes the `mknod` callable service to create a new FIFO special file.

Parameters

pathname

A path name for the file.

mode

A three- or four-digit number, corresponding to the access permission bits. Each digit must be in the range 0–7, and at least three digits must be specified. For more information on permissions, see Appendix B, “Setting permissions for files and directories,” on page 253.

Usage

1. The file permission bits specified in *mode* are modified by the process's file creation mask (see “umask” on page 149), and then used to set the file permission bits of the file being created.
2. The file's owner ID is set to the process's effective user ID (UID). The group ID is set to the group ID (GID) of the directory containing the file.
3. The mknod service sets the access, change, and modification times for the new file. It also sets the change and modification times for the directory that contains the new file.

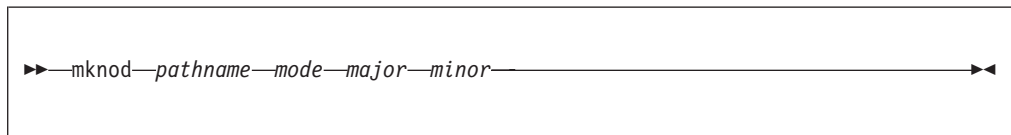
Examples

In the following example, assume that *pathname* was assigned a value earlier in the exec. The mode 777 grants read-write-execute permission to everyone.

```
"mkfifo (pathname) 777"
```

mknod

Format



Purpose

mknod invokes the mknod callable service to create a new character special file. You must be a superuser to use this function.

Parameters

pathname

A path name for the file.

mode

A three- or four-digit number, corresponding to the access permission bits. Each digit must be in the range 0–7, and at least three digits must be specified. For more information on permissions, see Appendix B, “Setting permissions for files and directories,” on page 253.

major

The device major number corresponds to a device driver supporting a class of devices (for example, interactive terminals). For information on specifying the device major number, see Creating special files in *z/OS UNIX System Services Planning*.

minor

The number that corresponds to a specific device within the class of devices referred to by the device major number. For information on specifying the device minor number, see *Creating special files in z/OS UNIX System Services Planning*.

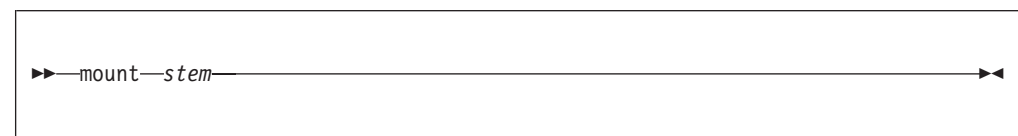
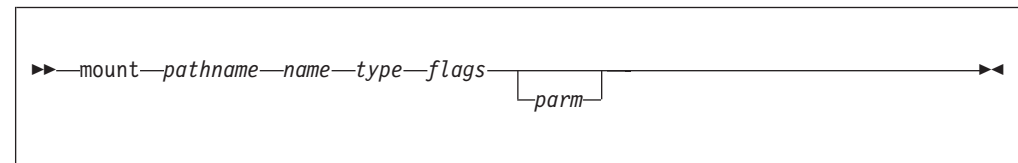
Usage

1. The file permission bits specified in *mode* are modified by the process's file creation mask (see "umask" on page 149), and then used to set the file permission bits of the file being created.
2. The file's owner ID is set to the process's effective user ID (UID). The group ID is set to the group ID (GID) of the directory containing the file.
3. The mknod service sets the access, change, and modification times for the new file. It also sets the change and modification times for the directory that contains the new file.

Examples

To create `/dev/null` with read-write-execute permission for everyone:

```
ISearchByExample"mknod /dev/null 777 4 0"
```

mount**Format****Purpose**

mount invokes the mount callable service to mount a file system, making the files in it available for use.

Requirement: The caller must have mount authorities to mount a file system. See the section on mount authority in *z/OS UNIX System Services Planning*.

When used in a sysplex, **mount** can also be used to change some of the mounted file system attributes, including the system that owns that mount.

Parameters**pathname**

The path name for the mount point.

mount

name

The name of the file system to be mounted. You must specify HFS data set names as fully qualified names in uppercase letters. Do not enclose the data set name in single quotes.

type

The type of file system, as defined by the FILESYSTYPE parameter on the BPXPRMxx parmlib member. Specify this as it is specified on the parmlib member.

flags

A value indicating how the file system is to be mounted. To specify the information, you can specify a numeric value (see Appendix A, "REXX predefined variables," on page 241) or one or more of the predefined variables beginning with MTM_. If more than one variable is specified, they are added together like open flags. (RDONLY and RDWR cannot be specified together.) The predefined variables used to derive the appropriate numeric value are:

MTM_NOSECURITY

Mount with no security.

MTM_NOSUID

The SETUID and SETGID mode bits on any executable in this file system are to be ignored when the program is run.

MTM_RDONLY

Mount read-only.

MTM_RDWR

Mount read-write.

MTM_SYNCHONLY

Mount must be completed synchronously; that is, **mount()** must not return +1.

parm

The name of a variable that contains a parameter string to be passed to the physical file system. The format and content of the string are specified by the physical file system that is to perform the logical mount.

For an HFS file system, *parm* is not used.

stem

The name of a stem variable which contains the mount variables. To set the mount variables, you can use a numeric value (see Appendix A, "REXX predefined variables," on page 241) or the predefined variables beginning with MNTE_ used to derive the numeric value. Unused stem variables should be set to the null string.

The following variables are used for mount requests:

Variable	Description
MNTE_FILETAG	4-byte file tag, the contents of which are mapped by the ST_FILETAG structure in the BPXYSTAT mapping macro. BPXYSTAT is described in BPXYSTAT — Map the Response Structure for stat in <i>z/OS UNIX System Services Programming: Assembler Callable Services Reference</i> .
MNTE_FSNAME	The name of the HFS data set containing the file system.
MNTE_FSTYPE	The file system type.

Variable	Description
MNTE_MODE	The file system type mount method. You can specify a numeric value (see Appendix A, "REXX predefined variables," on page 241) or one of the following predefined variables used to derive the appropriate numeric value: MNT_MODE_RDWR File system mounted read-write. MNT_MODE_RDONLY File system mounted read-only. MNT_MODE_AUNMOUNT The file system can be unmounted if the system's owner crashes. MNT_MODE_NOAUTOMOVE Automove is not allowed. MNT_MODE_NOSEC No security checks are enforced. MNT_MODE_NOSETID SetUID is not permitted for files in this filesystem.
MNTE_PARM	The parameter specified with mount() .
MNTE_PATH	The mountpoint path name.
MNTE_SYSLIST	A list of system names. For more information about MNTE_SYSLIST, see BPXYMNTE in <i>z/OS UNIX System Services Programming: Assembler Callable Services Reference</i> .
MNTE_SYSNAME	The name of the file system to be mounted on.

The following variables are used for changing mount attributes:

Variable	Description
MNTE_FSNAME	The name of the HFS data set containing the file system.
MNTE_MODE	The file system type mount method. You can specify a numeric value (see Appendix A, "REXX predefined variables," on page 241) or one of the following predefined variables used to derive the appropriate numeric value: MNT_MODE_RDWR File system mounted read-write. MNT_MODE_RDONLY File system mounted read-only. MNT_MODE_AUNMOUNT The file system can be unmounted if the system's owner crashes. MNT_MODE_NOAUTOMOVE Automove is not allowed.
MNTE_RFLAGS	Request flags (set to 3 to change the automove attribute, 1 to change mount owner).
MNTE_SYSNAME	The name of the system to be mounted on.
MNTE_SYSLIST	A list of system names. For more information about MNTE_SYSLIST, see BPXYMNTE in <i>z/OS UNIX System Services Programming: Assembler Callable Services Reference</i> .

Usage

1. The mount service effectively creates a virtual file system. After a file system is mounted, references to the path name that is mounted refer to the root directory on the mounted file system.
2. A file system can be mounted at only one point.
3. Parameter specifics for the HFS physical file system:
 - The *name* value must be uppercase and must be the name of the data set.

mount

- The *parm* parameter is not used.
4. You can use a wildcard (*) as the last item (or only item) of a system list (MNTE_SYSLIST). A wildcard is allowed only with an INCLUDE list, not with an EXCLUDE list.

Examples

1. To mount the HFS data set HFS.BIN.V1R1M0 on the mount point /v1r1m0 as a read-only file system:

```
"mount /v1r1m0 HFS.BIN.V1R1M0 HFS" mtm_rdnly
```

2. To mount an HFS read/write with this system as the owner:

```
m.=" /* set all of the stem variables to the null string */
m.mnte_mode=mnt_mode_rdwr
m.mnte_fsname='OMVS.HFS.U.WJS'
m.mnte_fstype='HFS'
m.mnte_path='/u/wjs'
address syscall 'mount m.'
```

3. To mount an HFS file system read/write with system SYS2 as the owner and with NOAUTOMOVE:

```
m.="
m.mnte_mode=mnt_mode_rdwr+mnt_mode_noautomove
m.mnte_fsname='OMVS.HFS.U.WJS'
m.mnte_fstype='HFS'
m.mnte_path='/u/wjs'
m.mnte_sysname='SYS2'
address syscall 'mount m.'
```

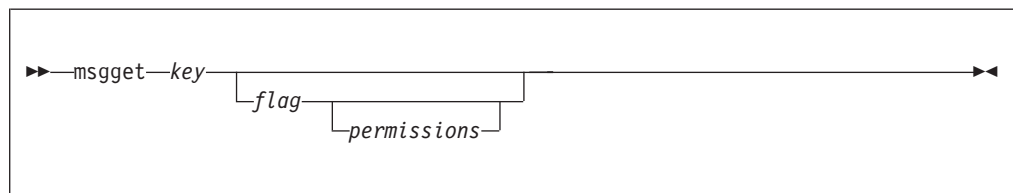
- 4.

```
mnt.='''
mnt.mnte_fsname = 'FS1.HFS'
mnt.mnte_path   = '/fs1'
mnt.mnte_mode   = mnt_mode_rdwr
mnt.mnte_fstype = 'HFS'

/* build an INCLUDE system list */
syslist      = 'SY1 SY2 SY3'
num_systems = d2c(words(syslist),2)
type_syslist = '0000'x /* or '0001'x for exclude list */
mnt.mnte_syslist = num_systems || type_syslist

do i = 1 to words(syslist)
  mnt.mnte_syslist = mnt.mnte_syslist || left(word(syslist,i),8)
end
address syscall 'mount mnt.'
say 'mount RRR =' retval errno errnoj
```

msgget



Function

msgget locates or creates a message queue.

Parameters

key

The message queue key. *key* must be 4 bytes.

flag

The type of connection to the message queue. It can be 0 to get the existing queue, or it can be a sum of any of the following:

- 1 ipc_create. This is the default.
- 2 ipc_excl
- 4 ipc_rcvtypepid
- 8 ipc_sndtypepid

permissions

The permission settings in octal. The default is 600.

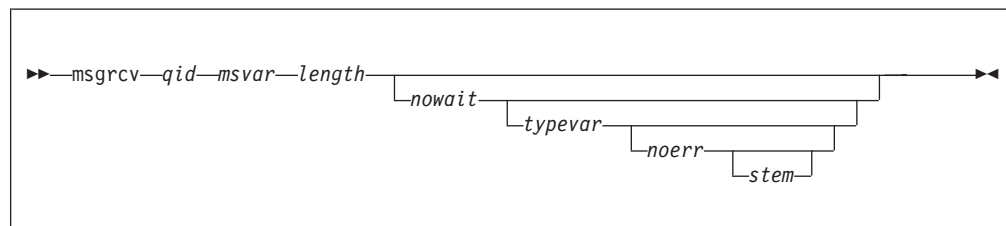
Example

```
'msgget WJSQ'  
qid=retval
```

Usage notes

Upon return, RETVAL contains -1 if the service fails. Otherwise, the queue ID is returned.

msgrcv



Function

`msgrcv` receives messages from message queues.

Parameters

qid

The queue ID.

msgvar

The name of a variable that contains the received message.

length

The maximum length of the message to be received. This parameter is flagged as an error if storage cannot be obtained for a buffer of this size.

nowait

Specifies whether the caller is to wait for a message of the required type.

- 0 Waits for a message of the requested type. This is the default.
- 1 An error is immediately returned if a message of the requested type is not on the queue.

msgrcv

typevar

The name of a variable that contains the type of message to be received. On output, the variable contains the type for the message received. The type is a 4-byte binary value with a default of '00000000'x, which means that any message type is received.

noerr

The action to be taken if the message is longer than the receive length.

0 An error is returned. This is the default.

1 The message is truncated.

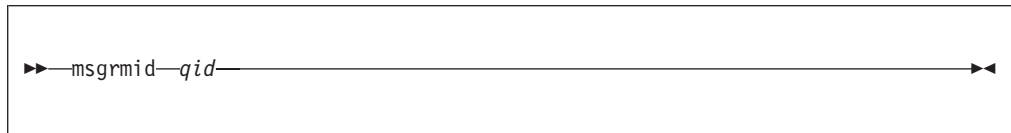
stem

The name of a stem variable that is used to return information on the message. Upon return, stem.0 contains the number of variables returned. To access the variables, use a numeric value (see Appendix A, "REXX predefined variables," on page 241) or any of the predefined variables: MSGBUF, MTIME, MSGBUF_UID, MSGBUF_GID, or MSGBUF_PID.

Example

```
'msgrcv (qid) msg 500'
```

msgrmid



Function

`msgrmid` removes the message queue.

Parameters

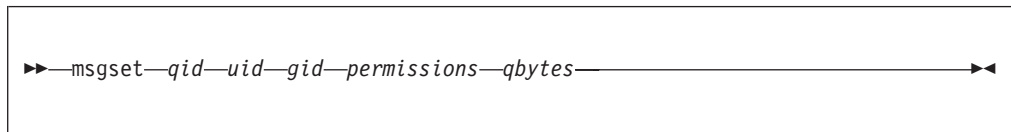
qid

The queue ID.

Example

```
'msgrmid (qid)'
```

msgset



Function

`msgset` sets the message queue attributes.

Parameters

qid

The queue ID.

uid

Sets the new UID.

gid

Sets the new GID.

permissions

Sets the new permissions in octal format.

qbytes

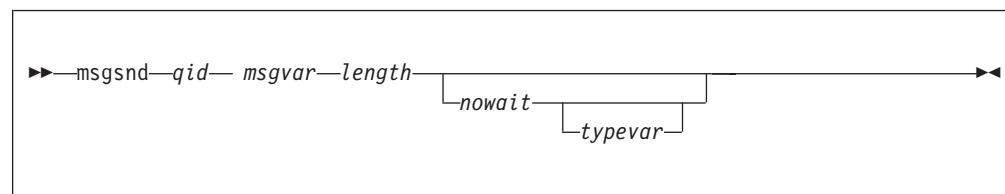
Sets the maximum bytes that are allowed on the queue.

All values must be set. Use **msgstat** to obtain the current values if you do not want to change a value.

Example

```
'msgstat (qid) mst.'
```

```
'msgset' qid mst.msg_uid mst.msg_gid 660 mst.msg_qbytes
```

msgsnd**Function**

msgsnd sends messages to message queues.

Parameters**qid**

The queue ID.

msgvar

The name of a variable that contains the message to be sent.

length

The maximum length of the message to be sent. This parameter is flagged as an error if storage cannot be obtained for a buffer of this size.

nowait

Specifies whether the caller is to wait until the message can be placed on the message queue.

0 Wait for the message to be placed on the message queue. This is the default.

1 An error is returned if the message cannot be placed on the queue immediately.

typevar

The name of a variable that contains the type of message to be sent. The type is a 4-byte binary value with a default of '00000001'x.

Example

```
'msgsnd (qid) msg' length(msg)
```

msgstat

```
▶▶ msgstat qid stem ▶▶
```

Function

msgstat obtains status information for a message queue.

Parameters

qid

The queue ID.

stem

The name of a stem variable that is used to return the status information.

Upon return, stem.0 contains the number of variables that are returned. To access the status values, use a numeric value (see Appendix A, "REXX predefined variables," on page 241) or any of the predefined variables:

MSG_UID	MSG_QBYTES
MSG_GID	MSG_LSPID
MSG_CUID	MSG_LRPID
MSG_CGID	MSG_STIME
MSG_TYPE	MSG_RTIME
MSG_MODE	MSG_CTIME
MSG_QNUM	

Example

```
'msgstat (qid) mst.'
```

open

```
▶▶ open pathname o_flags mode ▶▶
```

Function

open invokes the open callable service to access a file and create a file descriptor for it. The file descriptor is returned in RETVAL.

Parameters

pathname

A pathname for the file.

o_flags

One or more numeric values that describe how the file is to be opened. You can specify a numeric value (see Appendix A, "REXX predefined variables," on page 241

page 241) or any of the predefined variables that begin with `O_` used to derive the appropriate numeric value. For example, the numeric values `128+3` or `131` or the predefined variables `o_creat+o_rdwr` could be used to specify how the file is to be opened:

Variable	Description
<code>O_APPEND</code>	Set the offset to EOF before each write.
<code>O_CREAT</code>	Create the file if it does not exist.
<code>O_EXCL</code>	Fail if the file does exist and <code>O_CREAT</code> is set.
<code>O_NOCTTY</code>	Do not make this file a controlling terminal for the calling process.
<code>O_NONBLOCK</code>	Do not block an open, a read, or a write on the file (do not wait for terminal input).
<code>O_RDONLY</code>	Open for read-only.
<code>O_RDWR</code>	Open for read and write.
<code>O_SYNC</code>	Force synchronous updates.
<code>O_TRUNC</code>	Write, starting at the beginning of the file.
<code>O_WRONLY</code>	Open for write-only.

mode

A three- or four-digit number, corresponding to the access permission bits. If this optional parameter is not supplied, the mode is defaulted to `000`, which is useful for opening an existing file. For an explanation of how mode is handled if you are creating a file, see the usage notes below.

Each digit must be in the range `0–7`, and at least three digits must be specified. For more information on permissions, see Appendix B, “Setting permissions for files and directories,” on page 253.

Usage notes

When a file is created with the `O_CREAT` or `O_EXCL` options, the file permission bits as specified in the mode parameter are modified by the process's file creation mask (see “`umask`” on page 149), and then used to set the file permission bits of the file being created.

O_EXCL option: If the `O_EXCL` bit is set and the create bit is not set, the `O_EXCL` bit is ignored.

O_TRUNC option: Turning on the `O_TRUNC` bit opens the file as though it had been created earlier but never written into. The mode and owner of the file do not change (although the change time and modification time do); but the file's contents are discarded. The file offset, which indicates where the next write is to occur, points to the first byte of the file.

O_NONBLOCK option: A FIFO special file is a shared file from which the first data written is the first data read. The `O_NONBLOCK` option is a way of coordinating write and read requests between processes sharing a FIFO special file. It works this way, provided that no other conditions interfere with opening the file successfully:

- If a file is opened read-only and `O_NONBLOCK` is specified, the open request succeeds. Control returns to the caller immediately.
- If a file is opened write-only and `O_NONBLOCK` is specified, the open request completes successfully, provided that another process has the file open for reading. If another process does not have the file open for reading, the request ends with `RETVAL` set to `-1`.

open

- If a file is opened read-only and O_NONBLOCK is omitted, the request is blocked (control is not returned to the caller) until another process opens the file for writing.
- If a file is opened write-only and O_NONBLOCK is omitted, the request is blocked (control is not returned to the caller) until another process opens the file for reading.
- If the O_SYNC update option is used, the program is assured that all data updates have been written to permanent storage.

Example

To open or create the file `/u/linda/my.exec` with read-write-execute permission for the owner, and to read and write starting at the beginning of the file:

```
"open /u/linda/my.exec" o_rdwr+o_trunc+o_creat 700
```

opendir



Function

`opendir` invokes the `opendir` callable service to open a directory stream so that it can be read by `rddir`. The file descriptor is returned in `RETVAl`.

Parameters

`pathname`

A pathname for the directory.

Usage notes

1. You can use `opendir` and `closedir` together with the `rddir` syscall command, but not with the `readdir` command. The `rddir` command reads a directory in the `readdir` callable service format. Alternatively, you can simply use the `readdir` syscall command to read an entire directory and format it in a stem.
2. The `opendir` service opens a directory so that the first `rddir` service (see “`rddir`” on page 110) starts reading at the first entry in the directory.
3. `RETVAl` is a file descriptor for a directory only. It can be used only as input to services that expect a directory file descriptor. These services are `closedir`, `rewinddir`, and `rddir`.

Example

To open the directory `/u/edman`:

```
"opendir /u/edman"
```

pathconf

```
▶▶—pathconf—pathname—name—◀◀
```

Function

pathconf invokes the pathconf callable service to determine the current values of a configurable limit or option (variable) that is associated with a file or directory. The limit or option is returned in RETVAL.

Parameters

pathname

A pathname for a file or directory.

name

A numeric value that indicates which limit is returned. You can specify a numeric value (see Appendix A, “REXX predefined variables,” on page 241) or the predefined variable beginning with PC_ used to derive the appropriate numeric value.

Variable	Description
PC_ACL	Test if access control lists (ACLs) are supported.
PC_ACL_MAX	Maximum number of entries allowed in an ACL.
PC_LINK_MAX	Maximum value of a file's link count.
PC_MAX_CANON	Maximum number of bytes in a terminal canonical input line.
PC_MAX_INPUT	Minimum number of bytes for which space will be available in a terminal input queue; therefore, the maximum number of bytes a portable application may require to be typed as input before reading them.
PC_NAME_MAX	Maximum number of bytes in a filename (not a string length; count excludes a terminating null).
PC_PATH_MAX	Maximum number of bytes in a pathname (not a string length; count excludes a terminating null).
PC_PIPE_BUF	Maximum number of bytes that can be written atomically when writing to a pipe.
PC_POSIX_CHOWN_RESTRICTED	Change ownership (“chown” on page 39) function is restricted to a process with appropriate privileges, and to changing the group ID (GID) of a file only to the effective group ID of the process or to one of its supplementary group IDs.
PC_POSIX_NO_TRUNC	Pathname components longer than 255 bytes generate an error.
PC_POSIX_VDISABLE	Terminal attributes maintained by the system can be disabled using this character value.

Usage notes

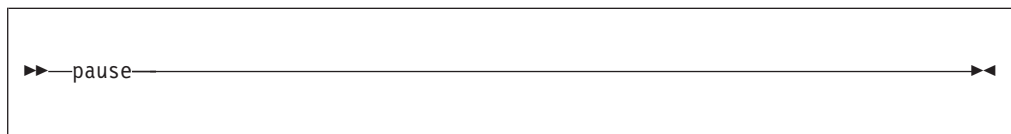
1. If *name* refers to MAX_CANON, MAX_INPUT, or _POSIX_VDISABLE, the following applies:
 - If *pathname* does not refer to a terminal file, the service returns -1 in RETVAL and sets ERRNO to EINVAL.
2. If *name* refers to NAME_MAX, PATH_MAX, or _POSIX_NO_TRUNC, the following applies:
 - If *pathname* does not refer to a directory, the service still returns the requested information using the parent directory of the specified file.
3. If *name* refers to PC_PIPE_BUF, the following applies:
 - If *pathname* refers to a pipe or a FIFO, the value returned applies to the referred-to object itself. If *pathname* refers to a directory, the value returned applies to any FIFOs that exist or can be created within the directory. If *pathname* refers to any other type of file, the pathconf service returns -1 in RETVAL and sets the ERRNO to EINVAL.
4. If *name* refers to PC_LINK_MAX, the following applies:
 - If *pathname* refers to a directory, the value returned applies to the directory.

Example

To determine the maximum number of bytes allowed in a pathname in the root directory:

```
"pathconf /" pc_name_max
```

pause



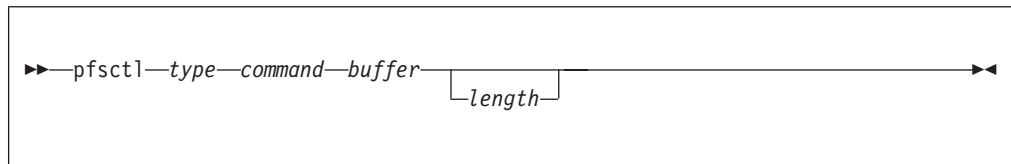
Function

pause invokes the pause callable service to suspend execution of the calling thread until delivery of a signal that either executes a signal-catching function or ends the thread. See “Using the REXX signal services” on page 10 for more information.

Usage notes

1. A thread that calls **pause** does not resume processing until a signal is delivered with an action to either process a signal-handling function or end the thread. Some signals can be blocked by the thread's signal mask; see “sigprocmask” on page 134 for details.
2. If an incoming unblocked signal ends the thread, **pause** never returns to the caller.
3. A return code is set when any failures are encountered that prevent this function from completing successfully.

pfsctl



Function

pfsctl invokes the pfsctl service to issue physical file system (PFS) control commands to a PFS. The meaning of the command and argument are specific to and defined by the PFS.

Parameters

type

The type of file system, as defined by the FILESYSTYPE parameter on the BPXPRMxx parmlib member. Specify this as it is specified on the parmlib member.

command

The name of a PFS control command.

buffer

The name of a buffer containing the argument to be passed to the PFS. The argument is limited to 4095 bytes.

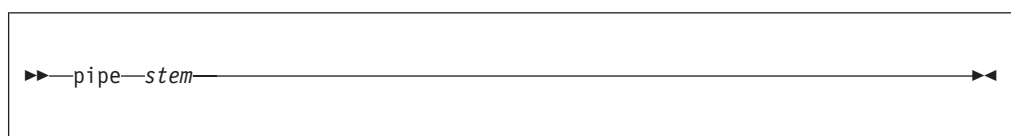
length

An optional numeric value, indicating the length of the argument. If length is not specified, the length of the buffer is used. The maximum length allowed is 4095 bytes.

Usage notes

1. This service is provided for communication between a program running in a user process and a physical file system.
It is similar to **ioctl**, but the command is directed to the physical file system itself rather than to, or for, a particular file or device.
2. As an example of how you could use this function in writing a physical file system, consider the requirement to display status and performance statistics about the physical file system. You can collect this information in the physical file system, but you need a way to display it to the user. For more information about the use of **pfsctl**, see *z/OS DFSMS Using Data Sets*.
With **pfsctl**, your status utility program can easily fetch the information it needs from the physical file system.

pipe



pipe

Function

pipe invokes the pipe callable service to create a pipe; or an I/O channel that a process can use to communicate with another process, another thread (in this same process or another process), or, in some cases, with itself. Data can be written into one end of the pipe and read from the other end.

Parameters

stem

Upon return, *stem.0* contains the number of variables returned. Two stem variables are returned:

stem.1 The file descriptor for the end of the pipe that you read from

stem.2 The file descriptor for the end of the pipe that you write to

Usage notes

When the pipe call creates a pipe, the O_NONBLOCK and FD_CLOEXEC flags are turned off on both ends of the pipe. You can turn these flags on or off by invoking:

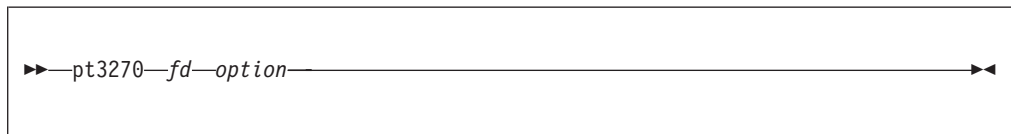
- “f_setfl” on page 59 for the flag O_NONBLOCK
- “f_setfd” on page 59 for the flag FD_CLOEXEC

Example

To create a pipe:

```
"pipe pfd."
```

pt3270



Function

pt3270 invokes the tcgetattr and tcsetattr callable services to query, set, and reset 3270 passthrough mode.

A REXX program running under a shell started from the OMVS TSO/E command can use this service to send and receive a 3270 data stream or issue TSO/E commands.

Parameters

fd The file descriptor for the file.

option

A number that identifies the service being requested:

Number

Service

1 Query 3270 passthrough support for this file.

Upon return, RETVAL will contain a code for the current state of the file descriptor, or -1 if there is an error.

- 0 or -1** This file cannot support 3270 passthrough mode.
- 1** This file can support 3270 passthrough mode.
- 3** This file is currently in 3270 passthrough mode.
- 2** Set 3270 passthrough support for this file. If this is attempted on a file that does not support 3270 passthrough mode, upon return RETVAL contains -1 and ERRNO contains the value for ENOSYS.
- 3** Reset 3270 passthrough support for this file.

Example

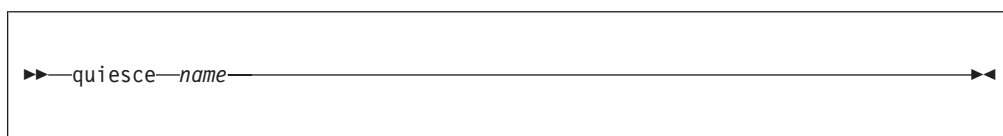
The following is an example of a REXX program that can accept a TSO/E command as its argument and issue the command through OMVS using 3270 passthrough mode. This REXX program would be located in the HFS and run as a command from the shell.

```

/* rexx */
parse arg cmd
if cmd=' then return
address syscall
'pt3270 1 2'                               /* set passthrough mode on stdout */
if retval=-1 then
  do
    say 'Cannot set passthrough mode' retval errno errnojr
    return
  end
buf='ff51000000010001'x || ,              /* OMVS passthrough command */
  d2c(length(cmd),4) || cmd
"write 1 buf"                               /* send command to OMVS */
"read 1 ibuf 1000"                          /* discard the response */
'pt3270 1 3'                               /* reset passthrough mode */
return 0

```

quiesce



Function

quiesce invokes the quiesce callable service to quiesce a file system, making the files in it unavailable for use. You must be a superuser to quiesce a file system.

Parameters

name

The name of the file system to be quiesced, specified as the name of an HFS data set. You must specify HFS data set names as fully qualified names in uppercase letters. Do not enclose the data set name in single quotes.

Usage notes

1. After a quiesce service request, the file system is unavailable for use until a subsequent unquiesce service request is received.

quiesce

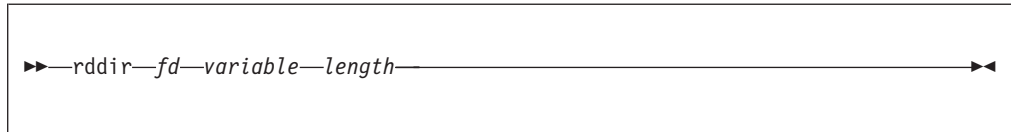
2. Users accessing files in a quiesced HFS file system are suspended until an unquiesce request for the file system is processed. Other file systems may send an EAGAIN instead of suspending the user.
3. If a file system that is not mounted is quiesced, that file system cannot be mounted until the file system is unquiesced. This ensures that no one can use the file system while it is quiesced.

Example

To quiesce an HFS data set named HFS.USR.SCHOEN:

```
"quiesce HFS.USR.SCHOEN"
```

rddir



Function

rddir invokes the readdir callable service to read multiple name entries from a directory and format the information in the readdir callable service format. To format this type of information in a stem, see “readdir” on page 113. The number of entries read is returned in RETVAL.

Parameters

fd The file descriptor (a number) for the directory to be read.

variable

The name of the buffer into which the directory entries are to be read.

length

The size of the buffer. After the read completes, the length of *variable* is the size of the buffer. The number of entries is returned in RETVAL.

Usage notes

1. You can use this command only with file descriptors opened using the opendir syscall command. The **rddir** syscall command reads a directory in the readdir callable service format. You can use **opendir**, **rewinddir**, and **closedir** together with the **rddir** syscall command, but not with the **readdir** syscall command. Alternatively, you can use the **readdir** syscall command to read an entire directory and format it in a stem.
2. The buffer contains a variable number of variable-length directory entries. Only full entries are placed in the buffer, up to the buffer size specified, and the number of entries is returned.
3. Each directory entry returned has the following format:

2-byte Entry_length

The total entry length, including itself.

2-byte Name_length

Length of the following Member_name subfield.

Member_name

A character field of length Name_length. This name is not null-terminated.

File system specific data

If name_length + 4 = entry_length, this subfield is not present.

The entries are packed together, and the length fields are not aligned on any particular boundary.

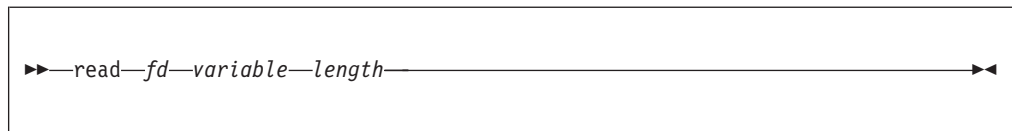
4. The buffer returned by one call to the readdir service must be used again on the next call to the readdir service to continue reading entries from where you left off. The buffer must not be altered between calls, unless the directory has been rewound.
5. The end of the directory is indicated in either of two ways:
 - A RETVAL of 0 entries is returned.
 - Some physical file systems may return a null name entry as the last entry in the caller's buffer. A null name entry has an Entry_length of 4 and a Name_length of 0.

The caller of the readdir service should check for both conditions.

Example

To read the entries from the directory with file descriptor 4 into the buffer named *buf*, which is 300 bytes long:

```
"rddir 4 buf 300"
```

read**Function**

read invokes the read callable service to read a specified number of bytes from a file into a buffer that you provide. The number of bytes read is returned in RETVAL.

Refer to *z/OS UNIX System Services Programming: Assembler Callable Services Reference* for restrictions when running in a conversion environment.

Parameters

fd The file descriptor (a number) for the file to be read.

variable

The name of the buffer into which the data is to be read.

length

The maximum number of characters to read. After the read completes, the length of *variable* is the number of bytes read. This value is also returned in RETVAL.

Usage notes

Length:

The value of *length* is not checked against any system limit.

Access time:

A successful read updates the access time of the file read.

Origin of bytes read:

If the file specified by *fd* is a regular file, or any other type of file where a seek operation is possible, bytes are read from the file offset associated with the file descriptor. A successful read increments the file offset by the number of bytes read.

For files where no seek operation is possible, there is no file offset associated with the file descriptor. Reading begins at the current position in the file.

Number of bytes read:

When a read request completes, the RETVAL field shows the number of bytes actually read—a number less than or equal to the number specified as *length*. The following are some reasons why the number of bytes read might be less than the number of bytes requested:

- Fewer than the requested number of bytes remained in the file; the end of file was reached before *length* bytes were read.
- The service was interrupted by a signal after some but not all of the requested bytes were read. (If no bytes were read, the return value is set to -1 and an error is reported.)
- The file is a pipe, FIFO, or special file and fewer bytes than *length* specified were available for reading.

There are several reasons why a read request might complete successfully with no bytes read (that is, with RETVAL set to 0). For example, zero bytes are read in these cases:

- The call specified a *length* of zero.
- The starting position for the read was at or beyond the end of the file.
- The file being read is a FIFO file or a pipe, and no process has the pipe open for writing.
- The file being read is a slave pseudoterminal and a zero-length canonical file was written to the master.

Nonblocking:

If a process has a pipe open for reading with nonblocking specified, a request to read from the file ends with a return value of -1 a return code of 0, and ERRNO of EAGAIN. But if nonblocking was not specified, the read request is blocked (does not return) until some data is written or the pipe is closed by all other processes that have the pipe open for writing.

Both master and slave pseudoterminals operate this way, too, except that how they act depends on how they were opened. If the master or the slave is opened blocking, the reads are blocked if there is no data. If it is opened nonblocking, EAGAIN is returned if there is no data.

SIGTTOU processing:

The read service causes signal SIGTTIN to be sent under the following conditions:

- The process is attempting to read from its controlling terminal, and
- The process is running in a background process group, and
- The SIGTTIN signal is not blocked or ignored, and

- The process group of the process is not orphaned.

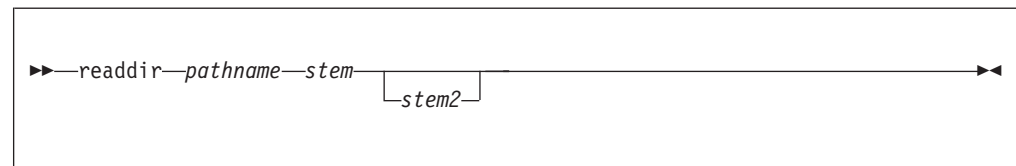
If these conditions are met, SIGTTIN is sent. If SIGTTIN has a handler, the handler gets control and the read ends with the return code set to EINTRO. If SIGTTIN is set to default, the process stops in the read and continues when the process is moved to the foreground.

Example

In the following example, assume that *fd* was assigned a value earlier in the exec. This reads 1000 characters from the file *fd* into the buffer *buf*:

```
"read (fd) buf 1000"
```

readdir



Function

readdir invokes the `opendir`, `readdir`, and `closedir` callable services to read multiple name entries from a directory and format the information in a stem.

Parameters

pathname

A pathname for a directory.

stem

Upon return, *stem.0* contains the number of directory entries returned. *stem.1* through *stem.n* (where *n* is the number of entries returned) each contain a directory entry.

stem2

If the optional *stem2* is provided, *stem2.1* through *stem2.n* (where *n* is the number of structures returned) each contain the stat structure for a directory entry.

You can use the predefined variables that begin with `ST_` or their equivalent numeric values (see Appendix A, “REXX predefined variables,” on page 241) to access the stat values. For example, *stem2.1.st_size* or *stem2.1.8* accesses the file size for the first directory entry. See “`fstat`” on page 65 for a list of the `ST_` variables.

Usage notes

The **rddir** command reads a directory in the `readdir` callable service format. You can use **opendir** and **closedir** together with the **rddir** syscall command, but not with the **readdir** syscall command. The **readdir** syscall command reads a directory and formats it in a stem.

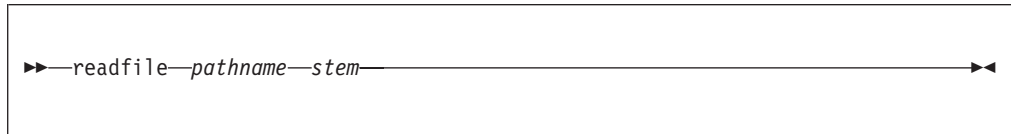
readdir

Example

To read the root directory and return information about the directory entries and a stat structure for each directory entry:

```
"readdir / root. rootst."
```

readfile



Function

readfile invokes the open, read, and close callable services to read from a text file and format it in a stem.

Parameters

pathname

A pathname for the file to be read.

stem

Upon return, *stem.0* contains the number of lines read. *stem.1* through *stem.n* (where *n* is the number of lines) each contain a line read.

Usage notes

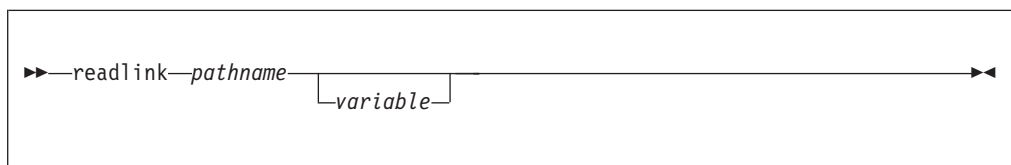
1. The maximum allowable length of a line in the file is 1024 characters. **RC=4** is returned when a line is greater than 1024 characters long or is not delimited with a newline character. The contents of the stem is unpredictable.
2. The newline characters that delimit the lines in a text file are stripped before the lines are saved in the stem.
3. When the return code indicates success, then Retval will be equal to or greater than zero. It will also indicate the corresponding file description number.

Example

In the following example, assume that *pathname* was assigned a value earlier in the exec:

```
"readfile (pathname) file."
```

readlink



Function

readlink invokes the readlink callable service to read the contents of a symbolic link. A symbolic link is a file that contains the pathname for another file. The length, in bytes, of the contents of the link is returned in RETVAL. If a *variable* is specified, the pathname is read into it.

Parameters

pathname

A pathname for the symbolic link.

variable

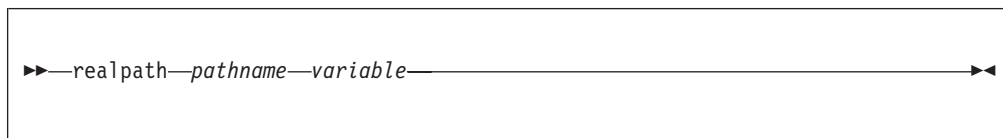
The name of the variable to hold the contents of the symbolic link. After the link is read, the length of the variable is the length of the symbolic link.

Example

In the following example, assume that *sl* and *linkbuf* were assigned a value earlier in the exec:

```
"readlink (sl) linkbuf"
```

realpath



Function

realpath invokes the realpath callable service to resolve a pathname to a full pathname without any symbolic links.

Parameters

pathname

The pathname to resolve.

variable

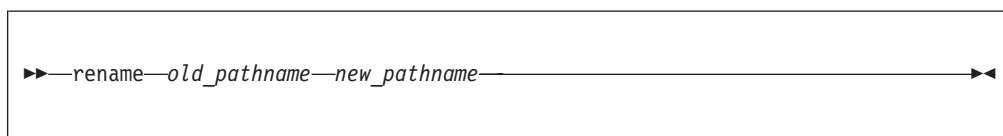
The name of the variable to contain the resolved pathname that is returned.

Example

The following example retrieves the real pathname for the working directory:

```
"realpath . mycwd"
```

rename



rename

Function

rename invokes the rename callable service to change the name of a file or directory.

Parameters

old_pathname

An existing pathname.

new_pathname

A new pathname.

Usage notes

The rename service changes the name of a file or directory from *old_pathname* to *new_pathname*. When renaming finishes successfully, the change and modification times for the parent directories of *old_pathname* and *new_pathname* are updated.

The calling process needs write permission for the directory containing *old_pathname* and the directory containing *new_pathname*. The caller does not need write permission for the files themselves.

Renaming files: If *old_pathname* and *new_pathname* are links referring to the same file, rename returns successfully.

If *old_pathname* is the name of a file, *new_pathname* must also name a file, not a directory. If *new_pathname* is an existing file, it is unlinked. Then the file specified as *old_pathname* is given *new_pathname*. The pathname *new_pathname* always stays in existence; at the beginning of the operation, *new_pathname* refers to its original file, and at the end, it refers to the file that used to be *old_pathname*.

Renaming directories: If *old_pathname* is the name of a directory, *new_pathname* must also name a directory, not a file. If *new_pathname* is an existing directory, it must be empty, containing no files or subdirectories. If empty, it is removed, as described in "rmdir" on page 117.

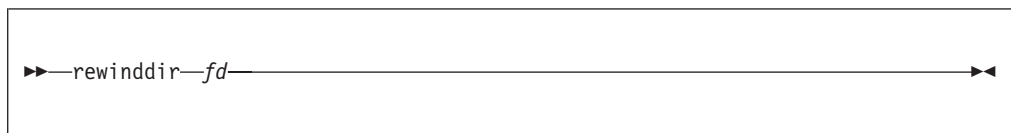
new_pathname cannot be a directory under *old_pathname*; that is, the old directory cannot be part of the pathname prefix of the new one.

Example

In the following example, assume that *old* and *new* were assigned values earlier in the exec:

```
"rename (old) (new)"
```

rewinddir



Function

rewinddir invokes the `rewinddir` callable service to rewind, or reset, to the beginning of an open directory. The next call to **rddir** reads the first entry in the directory.

Parameters

fd The file descriptor (a number) returned from an **opendir** syscall command. This is the file descriptor for the directory to be reset.

Usage notes

You can use this command only with file descriptors opened using the **opendir** syscall command. The **rddir** syscall command reads a directory in the `readdir` callable service format. You can use **opendir**, **rewinddir**, and **closedir** together with the **rddir** syscall command, but not with the **readdir** syscall command. Alternatively, you can use **readdir** syscall command to read an entire directory and format it in a stem.

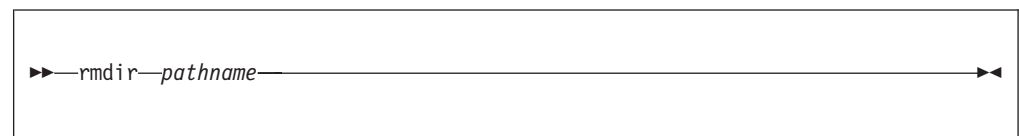
If the contents of the directory you specify have changed since the directory was opened, a call to the `rewinddir` service will reset the pointer into the directory to the beginning so that a subsequent call to the `readdir` service will read the new contents.

Example

To rewind the directory associated with file descriptor 4:

```
"rewinddir 4"
```

rmdir



Function

rmdir invokes the `rmdir` callable service to remove a directory. The directory must be empty.

Parameters

pathname
A pathname for the directory.

Usage notes

1. The directory must be empty.
2. If the directory is successfully removed, the change and modification times for the parent directory are updated.
3. If the link count of the directory becomes zero and no process has the directory open, the directory itself is deleted. The space occupied by the directory is freed for new use.

rmdir

4. If any process has the directory open when the last link is removed, the directory itself is not removed until the last process closes the directory. New files cannot be created under a directory after the last link is removed, even if the directory is still open.

Example

To remove the directory `/u/ehk0`:

```
"rmdir /u/ehk0"
```

setegid



Function

setegid invokes the setegid callable service to set the effective group ID (GID) of the calling process.

Parameters

gid

The numeric GID that the calling process is to assume.

Usage notes

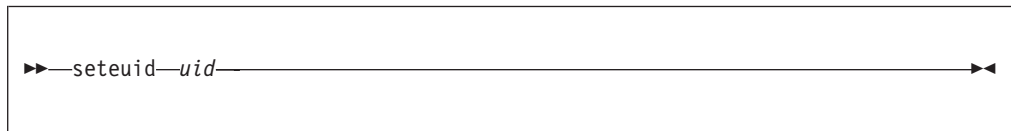
1. If *gid* is equal to the real group ID or the saved set group ID of the process, the effective group ID is set to *gid*.
2. If *gid* is not the same as the real group ID, and the calling process has the appropriate privileges, the effective group ID is set to *gid*.
3. The setegid service does not change any supplementary group IDs of the calling process.

Example

In the following example, assume that *gid* was assigned a value earlier in the exec:

```
"setegid" gid
```

seteuid



Function

seteuid invokes the seteuid callable service to set the effective user ID (UID) of the calling process.

Parameters

uid

The numeric UID that the calling process is to assume.

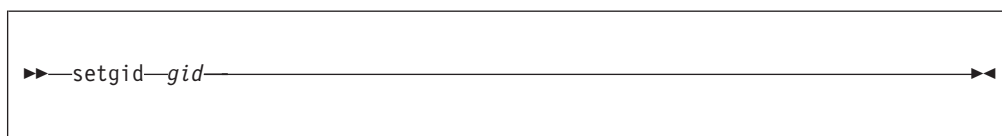
Usage notes

1. A user can switch to superuser authority (with an effective UID of 0) if the user is permitted to the BPX.SUPERUSER FACILITY class profile within RACF.
2. If *uid* is the same as the process's real or saved set UID, or the user has the appropriate privilege, the seteuid service sets the effective UID to be the same as *uid*.
3. The seteuid() function invokes SAF services to change the MVS identity of the address space. The MVS identity that is used is determined as follows:
 - a. If an MVS user ID is already known by the kernel from a previous call to a kernel function (for example, getpwnam()) and the UID for this user ID matches the UID specified on the seteuid() call, then this user ID is used.
 - b. For nonzero target UIDs, if there is no saved user ID or the UID for the saved user ID does not match the UID requested on the seteuid() call, the seteuid() function queries the security database (for example, using getpwnam) to retrieve a user ID. The retrieved user ID is then used.
 - c. If the target UID=0 and a user ID is not known, the seteuid() function always sets the MVS user ID to BPXROOT or the value specified on the SUPERUSER parm in sysparms. BPXROOT is set up during system initialization as a superuser with a UID=0. The BPXROOT user ID is not defined to the BPX.DAEMON FACILITY class profile. This special processing is necessary to prevent a superuser from gaining daemon authority.
 - d. A nondaemon superuser that attempts to set a user ID to a daemon superuser UID fails with an EPERM. When the MVS identity is changed, the auxiliary list of groups is also set to the list of groups for the new user ID. If the seteuid() function is issued from multiple tasks within one address space, use synchronization to ensure that the seteuid() functions are not performed concurrently. The execution of seteuid() function concurrently within one address space can yield unpredictable results.

Example

In the following example, assume that *uid* was assigned a value earlier in the exec:
`"seteuid (uid)"`

setgid



Function

setgid invokes the setgid callable service to set the real, effective, and saved set group IDs (GIDs) for the calling process.

setgid

Parameters

gid

The numeric GID that the calling process is to assume.

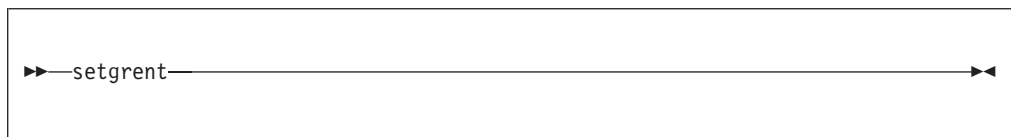
Usage notes

1. If *gid* is equal to the real group ID or the saved set group ID of the process, the effective group ID is set to *gid*.
2. If *gid* is not the same as the real group ID, and the calling process has the appropriate privileges, then the real, saved set, and effective group IDs are set to *gid*.
3. The setgid service does not change any supplementary group IDs of the calling process.

Example

In the following example, assume that *gid* was assigned a value earlier in the exec:
"setgid (gid)"

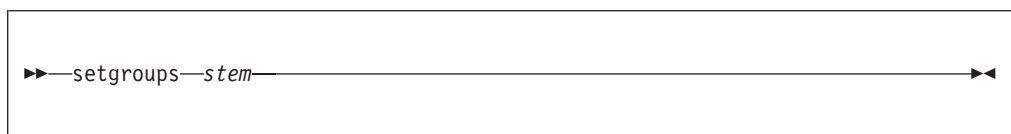
setgrent



Function

setgrent invokes the setgrent callable service to rewind, or reset to the beginning, the group database, allowing repeated searches. For more information, see "getgrent" on page 71.

setgroups



Function

setgroups invokes the setgroups callable service to set the supplemental group list for the process.

Parameters

stem

The name of a stem variable used to set the group list. Upon return, *stem.0* contains the number of variables containing a group id. *stem.1* to *stem.n* (where *n* is the number of variables) each contain one group id number.

Usage notes

A RETVAL of -1 indicates failure.

Example

In the following example, assume that the stem *gr.* was setup earlier in the exec:
`"setgroups gr."`

setpgid



```
▶▶ setpgid pid pgid ◀◀
```

Function

setpgid invokes the setpgid callable service to place a process in a process group. To identify the group, you specify a process group ID. You can assign a process to a different group, or you can start a new group with that process as its leader.

Parameters

pid

The numeric process ID (PID) of the process to be placed in a process group. If the ID is specified as 0, the system uses the process ID of the calling process.

pgid

The ID of the process group. If the ID is specified as 0, the system uses the process group ID indicated by *pid*.

Usage notes

1. The process group ID to be assigned to the group must be within the calling process's session.
2. The subject process (the process identified by *pid*) must be a child of the process that issues the service, and it must be in the same session; but it cannot be the session leader. It can be the caller.

Example

In the following example, assume that *pid* and *pgid* were assigned values earlier in the exec:

```
"setpgid" pid pgid
```

setpwent



```
▶▶ setpwent ◀◀
```

setpwent

Function

setpwent invokes the setpwent callable service to effectively rewind the user database to allow repeated searches. For more information, see “getpwent” on page 82.

setregid

```
►►—setregid—rgid—egid—◄◄
```

Function

setregid invokes the setregid callable service to set the real or effective GIDs for the calling process. If a specified value is set to -1, the corresponding real or effective GID of the calling process is left unchanged.

Parameters

rgid

The numeric GID value that becomes the real GID for the calling process.

egid

The numeric GID value that becomes the effective GID for the calling process.

Example

In the following example, assume that *rgid* and *egid* were assigned values earlier in the exec:

```
setregid rgid egid
```

setreuid

```
►►—setreuid—ruid—euid—◄◄
```

Function

setreuid invokes the setreuid callable service to set the real or effective user IDs (UIDs) of the calling process. If a specified value is set to -1, the corresponding real or effective UID of the calling process is left unchanged.

Parameters

ruid

The numeric UID value that becomes the real UID for the calling process.

euid

The numeric UID value that becomes the effective UID for the calling process.

Example

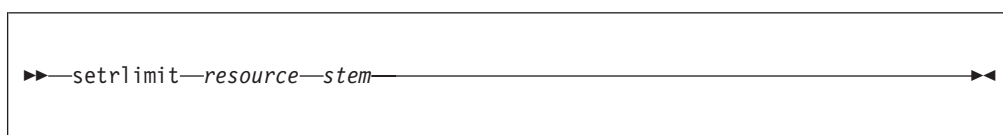
In the following example, assume that *ruid* and *euid* were assigned values earlier in the exec:

```
setreuid ruid euid
```

Usage notes

1. The `seteuid()` function invokes SAF services to change the MVS identity of the address space. The MVS identity that is used is determined as follows:
 - a. If an MVS user ID is already known by the kernel from a previous call to a kernel function (for example, `getpwnam()`) and the UID for this user ID matches the UID specified on the `seteuid()` call, then this user ID is used.
 - b. For nonzero target UIDs, if there is no saved user ID or the UID for the saved user ID does not match the UID requested on the `seteuid()` call, the `seteuid()` function queries the security database (for example, using `getpwnam`) to retrieve a user ID. The retrieved user ID is then used.
 - c. If the target UID=0 and a user ID is not known, the `seteuid()` function always sets the MVS user ID to BPXROOT or the value specified on the SUPERUSER parm in sysparms. BPXROOT is set up during system initialization as a superuser with a UID=0. The BPXROOT user ID is not defined to the BPX.DAEMON FACILITY class profile. This special processing is necessary to prevent a superuser from gaining daemon authority.
 - d. A nondaemon superuser that attempts to set a user ID to a daemon superuser UID fails with an EPERM. When the MVS identity is changed, the auxiliary list of groups is also set to the list of groups for the new user ID. If the `seteuid()` function is issued from multiple tasks within one address space, use synchronization to ensure that the `seteuid()` functions are not performed concurrently. The execution of `seteuid()` function concurrently within one address space can yield unpredictable results.

setrlimit



Function

setrlimit invokes the `setrlimit` callable service to set resource limits for the calling process. A resource limit is a pair of values; one specifies the current limit and the other a maximum limit.

Parameters

resource

The resource whose limit is being set. The maximum resource limit is `RLIM_INFINITY`.

You can use the predefined variables beginning with `RLIMIT_`, or their equivalent numeric values, to specify the resource. (See Appendix A, “REXX predefined variables,” on page 241 for the numeric values.)

setrlimit

Variable	Description	Allowable Range
RLIMIT_AS	Maximum address space size for a process.	10 485 760—2 147 483 647
RLIMIT_CORE	Maximum size (in bytes) of a core dump created by a process.	0—2 147 483 647
RLIMIT_CPU	Maximum amount of CPU time (in seconds) used by a process.	7—2 147 483 647
RLIMIT_FSIZE	Maximum files size (in bytes) created by a process.	0—2 147 483 647
RLIMIT_NOFILE	Maximum number of open file descriptors for a process.	5—131 072

stem

The name of the stem variable used to set the limit. *stem.1* is the first word, which sets the current limit, and *stem.2* is the second word, which sets the maximum limit. The values for each word depend on the *resource* specified. To specify no limit, use RLIM_INFINITY.

Usage notes

1. The current limit may be modified to any value that is less than or equal to the maximum limit. For the RLIMIT_CPU, RLIMIT_NOFILE, and RLIMIT_AS resources, if the setrlimit service is called with a current limit that is lower than the current usage, the setrlimit service fails with an EINVAL errno.
2. The maximum limit may be lowered to any value that is greater than or equal to the current limit.
3. The maximum limit can only be raised by a process that has superuser authority.
4. Both the current limit and maximum limit can be changed via a single call to setrlimit.
5. If the setrlimit service is called with a current limit that is greater than the maximum limit, setrlimit returns an EINVAL errno.
6. The resource limit values are propagated across exec and fork. An exception exists for exec. If a daemon process invokes exec and it invoked setuid before invoking exec, the limit values are set based on the limit values specified in parmlib member BPXPRMxx.
7. For a process that is not the only process within an address space, the RLIMIT_CPU and RLIMIT_AS limits are shared with all the processes within the address space. For RLIMIT_CPU, when the current limit is exceeded, action is taken on the first process within the address space. If the action is termination, all the processes within the address space are terminated.
8. In addition to the RLIMIT_CORE limit values, CORE dump defaults are set by SYSMDUMP defaults. See *z/OS MVS Initialization and Tuning Guide* for information on setting up SYSMDUMP defaults via the IEADMR00 parmlib member.
9. Core dumps are taken in 4160-byte increments. Therefore, RLIMIT_CORE values affect the size of core dumps in 4160-byte increments. For example, if the RLIMIT_CORE current limit value is 4000, core dumps will contain no data. If the RLIMIT_CORE current limit value is 8000, the maximum size of a core dump is 4160 bytes.
10. Limits may have an infinite value of RLIM_INFINITY.

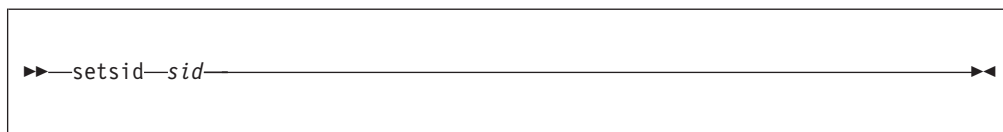
11. If the limit that is specified for RLIMIT_NOFILE is higher than that currently supported by the system, the limit will be reduced to the system maximum when it is used.
12. When setting RLIMIT_NOFILE, the current limit must be set higher than the value of the highest open file descriptor. Attempting to lower the current limit to a value less than or equal to the highest open file descriptor results in an error of EINVAL.
13. When setting RLIMIT_FSIZE, a limit of 0 prevents the creation of new files and the expansion of existing files.

Example

To reduce the maximum number of open files to 100 and the current limit to 50:

```
r.2=100
r.1=50
"setrlimit" rlimit_nofile r.
```

setsid



Function

setsid invokes the setsid callable service to create a new session with the calling process as its session leader. The caller becomes the group leader of a new process group.

Parameters

sid

The process ID of the calling process, which becomes the session or process group ID of the new process group.

Usage notes

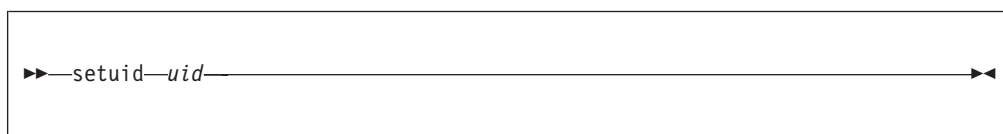
The calling process does not have a controlling terminal.

Example

In the following example, assume that *sid* was assigned a value earlier in the exec:

```
"setsid" sid
```

setuid



Function

setuid invokes the setuid callable service to set the real, effective, and saved set user IDs for the calling process.

Parameters

uid

The numeric UID the process is to assume.

Usage notes

1. A user can switch to superuser authority (with an effective UID of 0) if the user is permitted to the BPX.SUPERUSER FACILITY class profile within RACF.
2. If *uid* is the same as the process's real UID or the saved set UID, the setuid service sets the effective UID to be the same as *uid*.

If *uid* is not the same as the real UID of the process, and the calling process has appropriate privileges, then the real, effective, and saved set UIDs are set to *uid*.

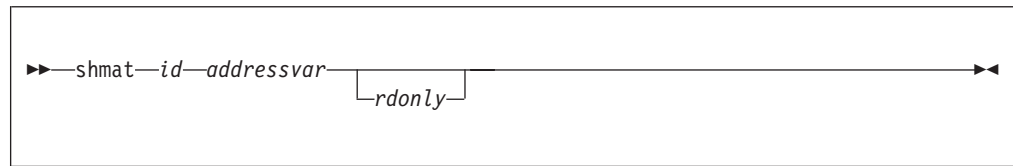
3. The seteuid() function invokes SAF services to change the MVS identity of the address space. The MVS identity that is used is determined as follows:
 - a. If an MVS user ID is already known by the kernel from a previous call to a kernel function (for example, getpwnam()) and the UID for this user ID matches the UID specified on the seteuid() call, then this user ID is used.
 - b. For nonzero target UIDs, if there is no saved user ID or the UID for the saved user ID does not match the UID requested on the seteuid() call, the seteuid() function queries the security database (for example, using getpwnam) to retrieve a user ID. The retrieved user ID is then used.
 - c. If the target UID=0 and a user ID is not known, the seteuid() function always sets the MVS user ID to BPXROOT or the value specified on the SUPERUSER parm in sysparms. BPXROOT is set up during system initialization as a superuser with a UID=0. The BPXROOT user ID is not defined to the BPX.DAEMON FACILITY class profile. This special processing is necessary to prevent a superuser from gaining daemon authority.
 - d. A nondaemon superuser that attempts to set a user ID to a daemon superuser UID fails with an EPERM. When the MVS identity is changed, the auxiliary list of groups is also set to the list of groups for the new user ID. If the seteuid() function is issued from multiple tasks within one address space, use synchronization to ensure that the seteuid() functions are not performed concurrently. The execution of seteuid() function concurrently within one address space can yield unpredictable results.

Example

In the following example, assume that *uid* was assigned a value earlier in the exec:

```
"setuid" uid
```


shmat



Function

`shmat` attaches a shared memory segment.

Parameters

id The ID for the shared memory segment.

addressvar

The name of the variable where the address of the memory segment is returned as a hex value.

rdonly

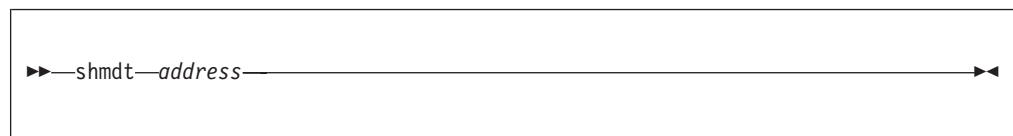
0 Attach as read/write. 0 is the default.

1 Attach as read-only.

Example

```
'shmat (shmid) shmaddr'
```

shmdt



Function

`shmdt` detaches a shared memory segment.

Parameters

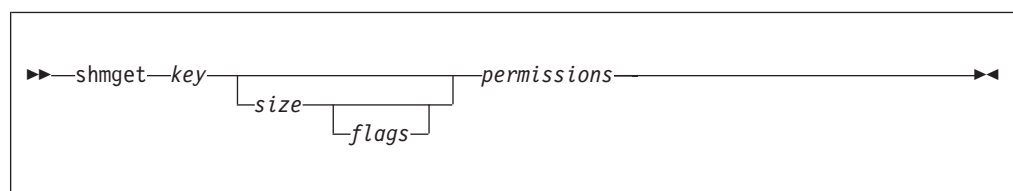
address

The address of the shared memory segment as returned by `shmat`.

Example

```
'shmdt (shmaddr)'
```

shmget



shmget

Function

`shmget` locates or creates a shared memory segment.

Parameters

key

The memory segment key. The value must be 4 bytes.

size

The size of the shared memory segment. The default is 0.

flags

The type of connection. The default is either 0, which obtains the existing queue, or a sum of any of the following flags:

IPC_CREAT	Creates an entry if the key does not exist.
IPC_EXCL	Fails if the key exists.
IPC_MEGA	Allocates in megabytes.
IPC_SHAREAS	Shares within an address space.

permissions

Permission settings in octal (the default is 600). It is dependent on the flags being specified.

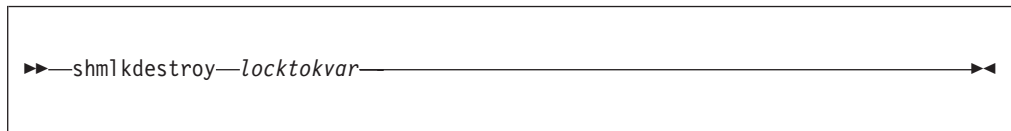
Example

```
'shmget WJSM 4096 (ipc_creat)'
```

Usage notes

Upon return, `RETV` contains `-1` if the service failed. Otherwise, the segment ID is returned.

shmlkdestroy



Function

`shmlkdestroy` removes a shared memory lock.

Parameters

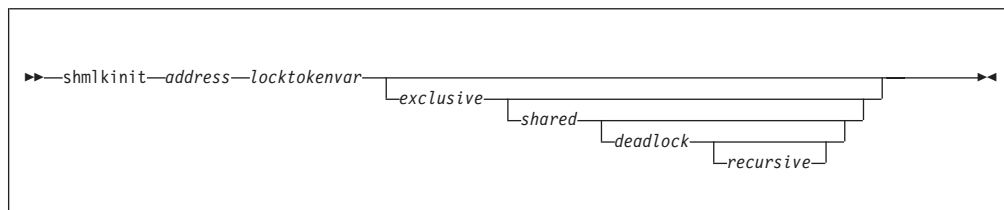
locktokvar

The variable that contains the lock token.

Example

```
'shmlkdestroy lk'
```

shmlkinit



Function

shmlkinit initializes a shared memory lock.

Parameters

address

The address of a lock word in a shared memory segment. The lock word must be in 4-byte hexadecimal format.

locktokvar

The variable where a lock token is returned.

exclusive

0 The lock cannot be obtained with the shared attribute. A lock that is obtained with the shared attribute can be obtained concurrently by other callers requesting a shared lock obtain.

1 Can be obtained with the shared attribute. This is the default.

deadlock

0 Deadlocks are not detected.

1 Deadlocks are detected. This is the default.

recursive

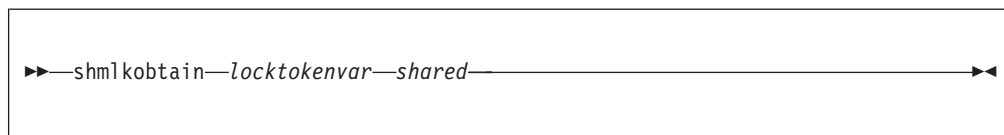
0 Recursive obtains are not allowed. This is the default.

1 Recursive obtains are allowed.

Example

```
'shmlkinit shmaddr lk'
```

shmlkobtain



Function

shmlkobtain obtains a shared memory lock

Parameters

locktokvar

The variable that contains the lock token.

shmlkobtain

shared
0 Obtain exclusive.
1 Obtain shared.

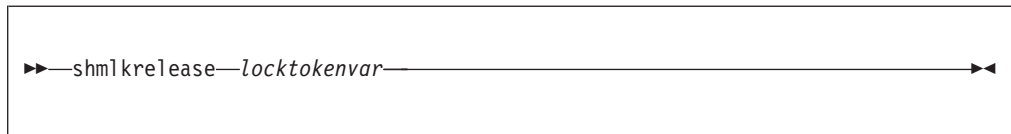
Example

```
'shmlkobtain lk 0'
```

Usage notes

Upon return, RETVAL contains **-1** if the service failed. Otherwise, it contains the count of the number of times the calling thread has the lock held.

shmlkrelease



Function

shmlkrelease releases a shared memory lock.

Parameters

locktokvar

The variable that contains the lock token.

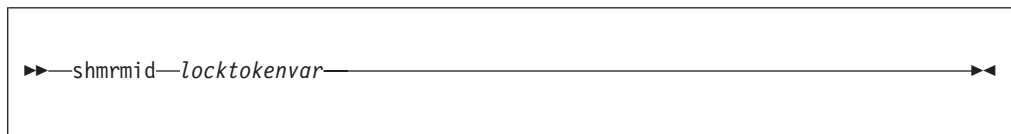
Example

```
'shmlkrelease lk'
```

Usage notes

Upon return, RETVAL contains **-1** if the service failed. Otherwise, it contains the count of the number of times the calling thread has the lock held.

shrmid



Function

shrmid removes a shared memory segment.

Parameters

id The ID for the shared memory segment.

Example

```
'shrmid (shmid)'
```

shmset

```
►► shmset id uid gid permissions ◄◄
```

Function

shmset sets the attributes for a shared memory segment.

Parameters

id The ID for the shared memory segment.

uid
Sets the new UID.

gid
Sets the new GID.

permissions
Sets the new permissions in octal.

All variables must be set. Use **shmget** to get the current values if you do not want to change a value.

Example

```
'shmget (shmid) shm.'  
'shmset' shmid shm.shm_uid shm.shm_gid 660
```

shmstat

```
►► shmstat id stem ◄◄
```

Function

shmstat obtains the status information for a shared memory segment.

Parameters

id The ID for the shared memory segment.

stem
The name of a stem variable that is used to return the status information. Upon return, stem.0 contains the number of variables that are returned. To access the status values, use a numeric value (see Appendix A, “REXX predefined variables,” on page 241) or any of the predefined variables:

SHM_UID	SHM_LPID
SHM_GID	SHM_CPID
SHM_CUID	SHM_ETIME
SHM_CGID	SHMD_DTIME

```

SHM_TYPE      SHM_CTIME
SHM_MODE
SHM_SEGZ

```

Example

```
'shmget (shmid) shm.'
```

sigaction

```

▶▶—sigaction—signal—new_handler—new_flag—old_handler—old_flag————▶▶

```

Function

sigaction invokes the sigaction callable service to examine, change, or both examine and change the action associated with a specific signal for all the threads in the process.

Note: All threads within a process share the signal handlers (a set of additional signals to be masked) and the flags specified by the sigaction callable service.

Parameters

signal

The signal, as specified by a numeric value or a predefined variable beginning with SIG.

Variable	Description
SIGABND	Abend
SIGABRT	Abnormal termination
SIGALRM	Timeout
SIGBUS	Bus error
SIGCHLD	Child process terminated or stopped
SIGCONT	Continue if stopped
SIGFPE	Erroneous arithmetic operation, such as division by zero or an operation resulting in an overflow
SIGHUP	Hangup detected on the controlling terminal
SIGILL	Termination that cannot be caught or ignored
SIGINT	Interactive attention
SIGIO	Completion of input or output
SIGIOERR	Error on input/output; used by the C runtime library
SIGKILL	Termination that cannot be caught or ignored
SIGPIPE	Write on a pipe with no readers
SIGPOLL	Pollable event
SIGPROF	Profiling timer expired
SIGQUIT	Interactive termination

Variable	Description
SIGSEGV	Detection of an incorrect memory reference
SIGSTOP	Stop that cannot be caught or ignored
SIGSYS	Bad system call
SIGTERM	Termination
SIGTRAP	Trap used by the ptrace callable service
SIGTSTP	Interactive stop
SIGTTIN	Read from a controlling terminal attempted by a member of a background process group
SIGTTOU	Write from a controlling terminal attempted by a member of a background process group
SIGURG	High bandwidth data is available at a socket
SIGUSR1	Reserved as application-defined signal 1
SIGUSR2	Reserved as application-defined signal 2
SIGVTALRM	Virtual timer expired
SIGXCPU	CPU time limit exceeded
SIGXFSZ	File size limit exceeded

new_handler

Specifies the new setting for handling the signal. The following predefined variables can be used for *new_handler*:

Variable	Description
SIG_CAT	Set signal handling to catch the signal.
SIG_DFL	Set signal handling to the default action.
SIG_IGN	Set signal handling to ignore the signal.
SIG_QRY	Query the handling for that signal.

new_flag

Used to modify the behavior of the specified signal. Specify one of these:

Variable	Description
0	Use the default behavior.
SA_NOCLDWAIT	Do not create zombie processes when child processes exit.
SA_RESETHAND	Reset the signal action to the default, SIG_DFL, when delivered.
SA_NOCLDSTOP	Do not generate SIGCHLD when the child processes stop.

old_handler

A variable name for the buffer where the system returns the old (current) signal handling action.

old_flag

The name of the variable that will store the old (current) signal action flags.

Usage notes

1. If *new_handler* is set to the action SIG_DFL for a signal that cannot be caught or ignored, the sigaction request is ignored and the return value is set to 0.

sigaction

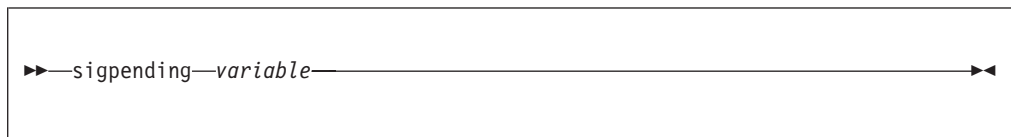
2. Setting a signal action to ignore for a signal that is pending causes the pending signal to be discarded.
3. Setting signal action SIG_IGN or catch for signals **SIGSTOP** or **SIGKILL** is not allowed.
4. Setting signal action SIG_IGN for **SIGCHLD** or **SIGIO** is not allowed.
5. The sigaction caller's thread must be registered for signals. You can register the thread by calling `syscalls('SIGON')`. If the thread is not registered for signals, the sigaction service fails with an ERRNO of EINVAL and ERRNOJR of JRNotSigSetup.

Example

To catch a **SIGALRM** signal:

```
"sigaction" sigalrm sig_cat 0 "prevhdlr prevflag"
```

sigpending



Function

sigpending invokes the sigpending callable service to return the union of the set of signals pending on the thread and the set of signals pending on the process. Pending signals at the process level are moved to the thread that called the sigpending callable service.

Parameters

variable

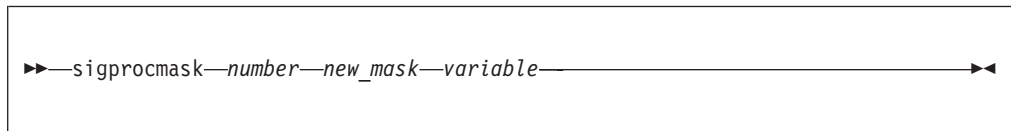
The name of the variable that will store a string of 64 characters with values 0 or 1, representing the 64 bits in a signal mask.

Example

To invoke sigpending:

```
"sigpending sigset"
```

sigprocmask



Function

sigprocmask invokes the sigprocmask callable service to examine or change the calling thread's signal mask.

Parameters

number

To specify the action to be taken on the thread's signal mask, you can specify a numeric value (see Appendix A, "REXX predefined variables," on page 241) or the predefined variable beginning with SIG_ used to derive the appropriate numeric value. Use one of the following predefined variables:

Variable	Description
SIG_BLOCK	Add the signals in <i>new_mask</i> to those to be blocked for this thread.
SIG_SETMASK	Replace the thread's signal mask with <i>new_mask</i> .
SIG_UNBLOCK	Delete the signals in <i>new_mask</i> from those blocked for this thread.

new_mask

The new signal mask, a string of 64 characters with values 0 or 1. The first character represents signal number 1. A string shorter than 64 characters is padded on the right with zeros. Mask bits set on represent signals that are blocked. For more information on signals, see "Using the REXX signal services" on page 10.

variable

The name of the buffer that will store the old signal mask, a string of 64 characters with values 0 or 1, representing the 64 bits in a signal mask. Mask bits set on represent signals that are blocked. A zero indicates that no signal mask was returned.

Usage notes

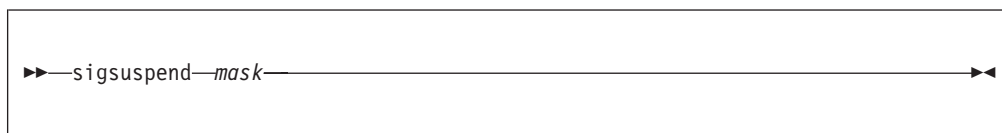
1. The sigprocmask service examines, changes, or both examines and changes the signal mask for the calling thread. This mask is called the thread's signal mask. If there are any pending unblocked signals, either at the process level or at the current thread's level after changing the signal mask, at least one of the signals is delivered to the thread before the sigprocmask service returns.
2. You cannot block the **SIGKILL** and the **SIGSTOP** signals. If you call the sigprocmask service with a request that would block those signals, that part of your request is ignored and no error is indicated.
3. A request to block signals that are not supported is accepted, and a return value of zero is returned.
4. All pending unblocked signals are moved from the process level to the current thread.

Example

In the following example, assume that *newsigset* was assigned a value earlier in the exec:

```
"sigprocmask" sig_setmask newsigset "oldsigset"
```

sigsuspend



Function

sigsuspend invokes the sigsuspend callable service to replace a thread's current signal mask with a new signal mask; then it suspends the caller's thread until delivery of a signal whose action is either to process a signal-catching service or to end the thread.

Parameters

mask

The new signal mask, a string of up to 64 characters with the values 0 or 1. The first character represents signal number 1. A string shorter than 64 characters is padded on the right with zeros. For more information on signals, see "Using the REXX signal services" on page 10.

Usage notes

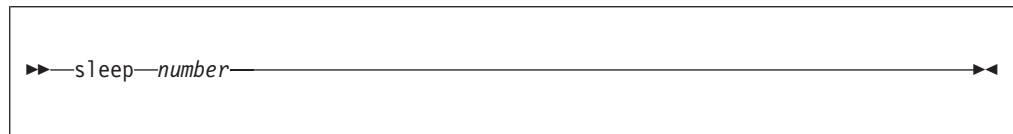
1. The caller's thread starts running again when it receives one of the signals not blocked by the mask set by this call, or a system failure occurs that sets the return code to some value other than EINTR.
2. The signal mask represents a set of signals that will be blocked. Blocked signals do not wake up the suspended service. The signals **SIGSTOP** and **SIGKILL** cannot be blocked or ignored; they are delivered to the program no matter what the signal mask specifies.
3. If the signal action is to end the thread, the sigsuspend service does not return.
4. All pending unblocked signals are moved from the process level to the current thread.

Example

In the following example, assume that *sigmask* was assigned a value earlier in the exec:

```
"sigsuspend" sigmask
```

sleep



Function

sleep invokes the sleep callable service to suspend running of the calling thread (process) until either the number of seconds specified by *number* has elapsed, or a signal is delivered to the calling thread to invoke a signal-catching function or end the thread.

Parameters

number

The number of seconds to suspend the process. For more information on signals, see "Using the REXX signal services" on page 10.

Usage notes

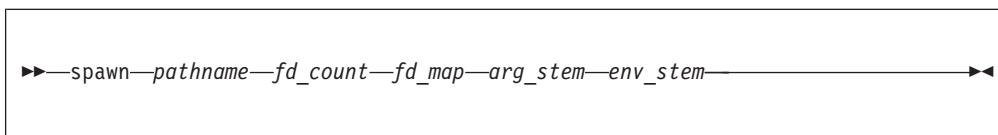
1. The suspension can actually be longer than the requested time, due to the scheduling of other activity by the system.
2. The sleep service suspends the thread running for a specified number of seconds, or until a signal is delivered to the calling thread that invokes a signal-catching function or ends the thread. An unblocked signal received during this time prematurely wakes up the thread. The appropriate signal-handling function is invoked to handle the signal. When that signal-handling function returns, the sleep service returns immediately, even if there is sleep time remaining.
3. The sleep service returns a zero in RETVAL if it has slept for the number of seconds specified. If the time specified by *number* has not elapsed when the sleep service is interrupted because of the delivery of a signal, the sleep service returns the unslept amount of time (the requested time minus the time actually slept when the signal was delivered) in seconds. Any time consumed by signal-catching functions is not reflected in the value returned by the sleep service.
4. The following are usage notes for a **SIGALRM** signal generated by the alarm or kill calls during the execution of the sleep call:
 - If the calling thread has **SIGALRM** blocked prior to calling the sleep service, the sleep service does not return when **SIGALRM** is generated, and the **SIGALRM** signal is left pending when sleep returns.
 - If the calling process has **SIGALRM** ignored when the **SIGALRM** signal is generated, then the sleep service does not return and the **SIGALRM** signal is ignored.
 - If the calling process has **SIGALRM** set to a signal-catching function, that function interrupts the sleep service and receives control. The sleep service returns any unslept amount of time, as it does for any other type of signal.
5. An EC6 abend is generated when the caller's PSW key or RB state prevents signals from being delivered.

Example

In the following example, assume that *timer* was assigned a value earlier in the exec:

```
"sleep (timer)"
```

spawn



Function

spawn invokes the spawn callable service to create a new process, called a *child process*, to run an executable file. It remaps the calling process's file descriptors for the child process.

Parameters

pathname

A path name for the executable file. path names can begin with or without a slash:

- A path name that begins with a slash is an absolute path name, and the search for the file starts at the root directory.
- A path name that does not begin with a slash is a relative path name, and the search for the file starts at the working directory.

fd_count

The number of file descriptors that can be inherited by the child process. In the new process, all file descriptors greater than or equal to *fd_count* are closed.

fd_map

A stem variable. The stem index specifies the child's file descriptor; for example, *stem.0* specifies the child's file descriptor 0. This array selects the file descriptors to be inherited. The value assigned to the variable indicates the parent's file descriptor that will be mapped to the child's file descriptor. For example, if *stem.0* is 4, the child process inherits the parent's file descriptor 4 as its descriptor 0. Any of the stem variables that contains a negative number or a nonnumeric value is closed in the child.

arg_stem

A stem variable. *stem.0* contains the number of arguments you are passing to the program. The first argument should always specify the absolute or relative path name of the program to be executed. If a relative path name is used and PATH is specified, PATH is used to resolve the name; otherwise, the name is processed as relative to the current directory. If a PATH environment variable is not passed, the first argument should specify the absolute path name or a relative path name for the program.

env_stem

A stem variable. *stem.0* contains the number of environment variables that you want the program to be run with. To propagate the current environment, pass `_environment`. Specify each environment variable as `VNAME=value`.

Usage notes

1. The new process (called the *child process*) inherits the following attributes from the process that calls **spawn** (called the *parent process*):
 - Session membership.
 - Real user ID.
 - Real group ID.
 - Supplementary group IDs.
 - Priority.
 - Working directory.
 - Root directory.
 - File creation mask.
 - The process group ID of the parent is inherited by the child.
 - Signals set to be ignored in the parent are set to be ignored in the child.
 - The signal mask is inherited from the parent.
2. The new child process has the following differences from the parent process:
 - The child process has a unique process ID (PID) that does not match any active process group ID.

- The child has a different parent PID (namely, the PID of the process that called **spawn**).
- If the *fd_count* parameter specified a 0 value, the child has its own copy of the parent's file descriptors, except for those files that are marked FCTL_CLOEXEC or FCTL_CLOFORK. The files marked FCTL_CLOEXEC or FCTL_CLOFORK are not inherited by the child. If the *filedesc_count* parameter specifies a value greater than 0, the parent's file descriptors are remapped for the child as specified in the *fd_map* stem with a negative number or non-numeric value.
- The FCTL_CLOEXEC and FCTL_CLOFORK flags are not inherited from the parent file descriptors to the child's.
- The foreground process group of the session remains unchanged.
- The process and system utilization times for the child are set to zero.
- Any file locks previously set by the parent are not inherited by the child.
- The child process has no alarms set (similar to the results of a call to the alarm service with *Wait_time* specified as zero) and has no interval timers set.
- The child has no pending signals.
- The child gets a new process image, which is not a copy of the parent process, to run the executable file.
- Signals set to be caught are reset to their default action.
- Memory mappings established by the parent via the *shm* or *mmap* services are not inherited by the child.
- If the *setuid* bit of the new executable file is set, the effective user ID and saved set-user-ID mode of the process are set to the owner user ID of the new executable file.
- If the *setgid* bit of the new executable file is set, the effective group ID and saved set-group-ID bit of the process are set to the owner user ID of the new executable file.

The last parameter that **spawn** passed to the executable file identifies the caller of the file as the *exec* or *spawn* service.

3. To control whether the spawned child process runs in a separate address space from the parent address space or in the same address space, you can specify the *_BPX_SHAREAS* environment variable. If *_BPX_SHAREAS* is not specified, is set to NO, or contains an unsupported value, the child process to be created will run in a separate address space from the parent process.
_BPX_SHAREAS=YES indicates that the child process to be created is to run in the same address space as the parent. If the program to be run is a set-user-ID or set-group-ID program that will cause the effective user-ID or group-ID of the child process to be different from that of the parent process, the *_BPX_SHAREAS=YES* value is ignored and the child process runs in its own address space.
4. In addition to recognizing the *_BPX_SHAREAS* environment variable, *spawn* recognizes all of the environment variables that are recognized by the *fork* and *exec* callable services.
5. The *fd_count* parameter can be \emptyset , which means that all file descriptors are inherited by the child.
6. The *fd_count* parameter is limited to a maximum value of 1000.
7. When the executable file to be run is a REXX *exec*, the first argument should be the path name of the REXX *exec*. Subsequent arguments for the *exec* can follow this.

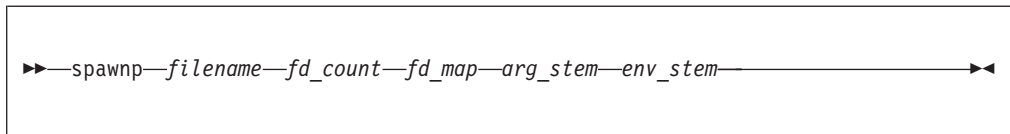
spawn

Example

In the following example, `/bin/ls` is run mapping its STDOUT and STDERR to file descriptors 4 and 5, which were previously opened in the exec, and STDIN is closed:

```
map.0=-1
map.1=4
map.2=5
parm.0=2
parm.1='/bin/ls'
parm.2='/'
'spawn /bin/ls 3 map. parm. _ _environment.'
```

spawnp

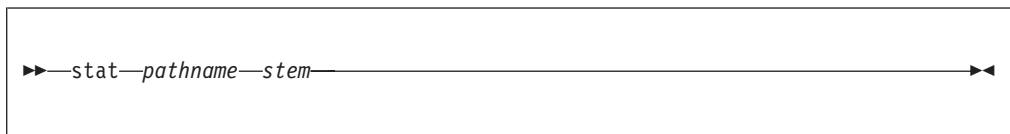


Function

spawnp invokes the spawn callable service and creates a new process, called a *child process*, to run an executable file. **spawnp** functions identically to the spawn function, except that it uses the **PATH** environment variable to resolve relative filenames.

See “spawn” on page 137 for more information.

stat



Function

stat invokes the stat callable service to obtain status about a specified file. You specify the file by its name. If the path name specified refers to a symbolic link, the symbolic link name is resolved to a file and the status information for that file is returned. To obtain status information about a symbolic link, rather than the file it refers to, see “lstat” on page 92.

To use a file descriptor to obtain this information, see “fstat” on page 65.

Parameters

pathname

A path name for the file.

stem

The name of a stem variable used to return the information. Upon return, *stem.0* contains the number of variables returned. You can use the predefined

variables beginning with ST_ or their equivalent numeric values to access the values they represent. (See Appendix A, "REXX predefined variables," on page 241 for the numeric values.)

Variable	Description
ST_AAUDIT	Auditor audit information
ST_ACCESSACL	1 if there is an access ACL (access control list)
ST_ETIME	Time of last access
ST_AUDITID	RACF File ID for auditing
ST_BLKSIZE	File block size
ST_BLOCKS	Blocks allocated
ST_CCSD	Coded character set ID; first 4 characters are the file tag
ST_CRTIME	File creation time
ST_CTIME	Time of last file status change
ST_DEV	Device ID of the file
ST_DMODELACL	1 if there is a directory model access control list (ACL)
ST_EXTLINK	External symbolic link flag, set to 0 or 1
ST_FID	File identifier
ST_FILEFMT	Format of the file. To specify the format, you can specify a numeric value (see Appendix A, "REXX predefined variables," on page 241) or one of the following predefined variables used to derive the appropriate numeric value: S_FFBNARY Binary data S_FFCR Text data delimited by a carriage return character S_FFCRLF Text data delimited by carriage return and line feed characters S_FFCRNL A text file with lines delimited by carriage-return and newline characters. S_FFLF Text data delimited by a line feed character S_FFLFCR Text data delimited by a line feed and carriage return characters S_FFNA Text data with the file format not specified S_FFNL Text data delimited by a newline character S_FFRECORD File data consisting of records with prefixes. The record prefix contains the length of the record that follows.
ST_FMODELACL	1 if there is a file model ACL
ST_GENVALUE	General attribute values
ST_GID	Group ID of the group of the file
ST_INO	File serial number
ST_MAJOR	Major number for a character special file
ST_MINOR	Minor number for a character special file
ST_MODE	File mode, permission bits only
ST_MTIME	Time of last data modification
ST_NLINK	Number of links
ST_RTME	File backup time stamp (reference time)
ST_SETGID	Set Group ID on execution flag, set to 0 or 1
ST_SETUID	Set User ID on execution flag, set to 0 or 1

stat

Variable	Description
ST_SIZE	File size for a regular file, in bytes. If file size exceeds $2^{31}-1$ bytes, size is expressed in megabytes, using an M (for example, 3123M).
ST_SECLABEL	Security Label
ST_STICKY	Sticky bit flag (keep loaded executable in storage), set to 0 or 1
ST_TYPE	Numeric value that represents the file type for this file. You can use a numeric value (see Appendix A, "REXX predefined variables," on page 241) or any of the predefined variables that begin with S_ to determine the file type: S_ISCHR Character special file S_ISDIR Directory S_ISFIFO FIFO special file S_ISREG Regular file S_ISSYM Symbolic link
ST_UAUDIT	Area for user audit information
ST_UID	User ID of the owner of the file

The stem variable *stem.st_type* is a number that represents the file type for this file. You can use the predefined variables beginning with S_ or their equivalent numeric values to determine the file type. For example, if *stem.st_type* is S_ISDIR, the file is a directory.

Usage notes

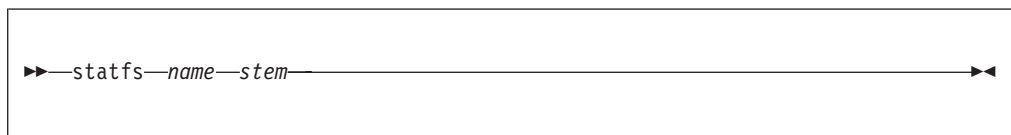
All time fields in *stem* are in POSIX format. You can use **gmtime** to convert it to other forms.

Example

In the following example, assume that *path* was assigned a value earlier in the exec:

```
"stat (path) st."
```

statfs



Function

statfs invokes the statfs callable service to obtain status information about a specified file system.

Parameters

name

The name of the file system to be mounted, specified as the name of an HFS data set. You must specify the HFS data set name as a fully qualified name in uppercase letters. Do not enclose the data set name in single quotes.

stem

The name of a stem variable used to return the information. Upon return, *stem.0* contains the number of variables returned. You can use the predefined variables beginning with STFS_ or their equivalent numeric values to access the status values they represent. (See Appendix A, "REXX predefined variables," on page 241 for the numeric values.) For example, *stem.stfs_avail* accesses the number of blocks available in the file system.

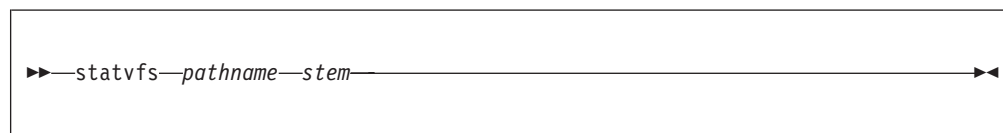
Variable	Description
STFS_AVAIL	Space available to unprivileged users in block-size units.
STFS_BFREE	Total number of free blocks.
STFS_BLOCKSIZE	Block size.
STFS_FAVAIL	Number of free file nodes available to unprivileged users.
STFS_FFREET	Total number of free file nodes.
STFS_FILES	Total number of file nodes in the file system.
STFS_FRSIZE	Fundamental file system block size.
STFS_FSID	File system ID set by the logical file system.
STFS_INUSE	Allocated space in block-size units.
STFS_INVARSEC	Number of seconds the file system will remain unchanged.
STFS_NAMEMAX	Maximum length of file name.
STFS_NOSEC	Mount data set with no security bit.
STFS_NOSUID	SETUID and SETGID are not supported.
STFS_RDONLY	File system is read-only.
STFS_TOTAL	Total space in block-size units.

Example

In the following example, assume that *fname* was assigned a value earlier in the exec:

```
"statfs (fname) st."
```

statvfs



Function

statvfs invokes the statvfs callable service to obtain status information about a file system, given the name of a file in the file system.

Parameters

pathname

The name of a file in a file system for which status information is to be obtained.

statvfs

stem

The name of a stem variable used to return the information. Upon return, *stem.0* contains the number of variables returned. You can use the predefined variables beginning with STFS_ or their equivalent numeric values (see Appendix A, "REXX predefined variables," on page 241) to access the status values they represent. For example, *stem.stfs_avail* accesses the number of blocks available in the file system.

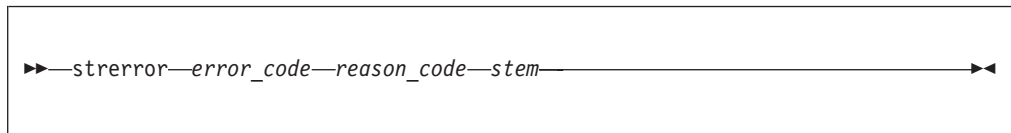
Variable	Description
STFS_AVAIL	Space available to unprivileged users in block-size units.
STFS_BFREE	Total number of free blocks.
STFS_BLOCKSIZE	Block size.
STFS_FAVAIL	Number of free file nodes available to unprivileged users.
STFS_FFREET	Total number of free file nodes.
STFS_FILES	Total number of file nodes in the file system.
STFS_FRSIZE	Fundamental file system block size.
STFS_FSID	File system ID set by the logical file system.
STFS_INUSE	Allocated space in block-size units.
STFS_INVARSEC	Number of seconds the file system will remain unchanged.
STFS_NAMEMAX	Maximum length of file name.
STFS_NOSEC	Mount data set with no security bit.
STFS_NOSUID	SETUID and SETGID are not supported.
STFS_RDONLY	File system is read-only.
STFS_TOTAL	Total space in block-size units.

Example

In the following example, assume that *fsname* was assigned a value earlier in the exec:

```
"statvfs (fsname) st."
```

strerror



Function

`strerror` retrieves diagnostic text for error codes and reason codes.

Parameters

error_code

Hex value for an error code as returned in `ERRNO` for other `SYSCALL` host commands. Specify 0 if text for this code is not being requested.

reason_code

Hex value for the reason code as returned in `ERRNOJR` for other `SYSCALL` host commands. Specify 0 if this code is not being requested.

stem

The name of a stem variable used to return the information. Upon return, *stem.0* contains the number of variables returned. You can use the predefined

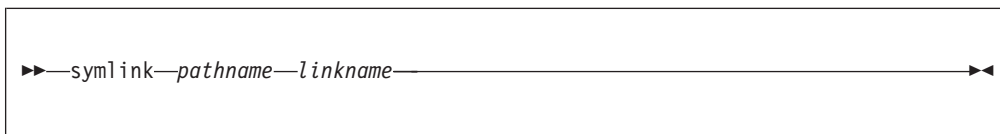
variables beginning with SE_ or their equivalent numeric values to access the values that they represent. See Appendix A, “REXX predefined variables,” on page 241 for the numeric values. For example, *stem.se_reason* accesses the reason code text. If text is unavailable, a null string is returned.

Variable	Description
SE_ERRNO	Text for the error number.
SE_REASON	Text for the reason code.
SE_ACTION	Text for any action to be taken to correct this error. This variable will be available only when a reason code is requested.
SE_MODID	Name of the module that detected the error. This variable will be available only when a reason code is requested.

Example

To get error text for the last syscall error:
`"strerror" errno errnojr "err."`

symlink



Function

symlink invokes the symlink callable service to create a symbolic link to a path name. This creates a symbolic link file.

Parameters

pathname

A path name for the file for which you are creating a symbolic link.

linkname

The path name for the symbolic link.

Usage notes

Like a hard link (described in “link” on page 90), a symbolic link allows a file to have more than one name. The presence of a hard link guarantees the existence of a file, even after the original name has been removed. A symbolic link, however, provides no such assurance; in fact, the file identified by *pathname* need not exist when the symbolic link is created. In addition, a symbolic link can cross file system boundaries.

When a component of a path name refers to a symbolic link rather than to a directory, the path name contained in the symbolic link is resolved. If the path name in the symbolic link begins with / (slash), the symbolic link path name is resolved relative to the process root directory. If the path name in the symbolic link does not begin with /, the symbolic link path name is resolved relative to the directory that contains the symbolic link.

symlink

If the symbolic link is not the last component of the original path name, remaining components of the original path name are resolved from there. When a symbolic link is the last component of a path name, it may or may not be resolved. Resolution depends on the function using the path name. For example, a rename request does not have a symbolic link resolved when it appears as the final component of either the new or old path name. However, an open request does have a symbolic link resolved when it appears as the last component. When a slash is the last component of a path name, and it is preceded by a symbolic link, the symbolic link is always resolved.

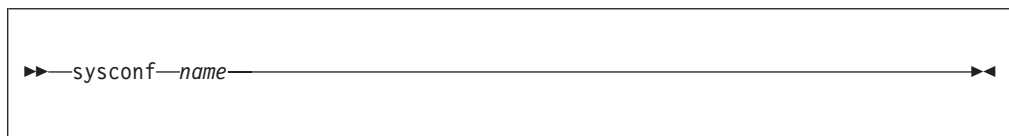
Because the mode of a symbolic link cannot be changed, its mode is ignored during the lookup process. Any files and directories to which a symbolic link refers are checked for access permission.

Example

To create a symbolic link named `/bin` for the file `/v.1.1.0/bin`:

```
"symlink /v.1.1.0/bin /bin"
```

sysconf



Function

`sysconf` invokes the `sysconf` callable service to get the value of a configurable system variable.

Parameters

name

A numeric value that specifies the configurable variable to be returned. You can specify a numeric value (see Appendix A, “REXX predefined variables,” on page 241) or the predefined variable beginning with `SC_` used to derive the appropriate numeric value:

Variable	Description
<code>SC_ARG_MAX</code>	The maximum length of all arguments and environment strings to <code>exec()</code>
<code>SC_CHILD_MAX</code>	The maximum number of simultaneous processes per real user ID
<code>SC_CLK_TCK</code>	The number of intervals per second used in defining the type <code>clock_t</code> , which is used to measure process execution times
<code>SC_JOB_CONTROL</code>	Support for job control
<code>SC_NGROUPS_MAX</code>	Maximum number of simultaneous supplementary group IDs per process
<code>SC_OPEN_MAX</code>	Maximum number of simultaneous open files per process
<code>SC_SAVED_IDS</code>	Support for saved set-user-IDs and set-group-IDs
<code>SC_THREAD_TASKS_MAX_NP</code>	Constant for querying the maximum number of threaded tasks per calling process

Variable	Description
SC_THREADS_MAX_NP	Constant for querying the maximum number of threads per calling process
SC_TZNAME_MAX	The number of bytes supported for the name of a time zone
SC_VERSION	The integer value 199009L
SC_2_CHAR_TERM	Constant for querying whether the system supports at least one raw mode terminal

Usage notes

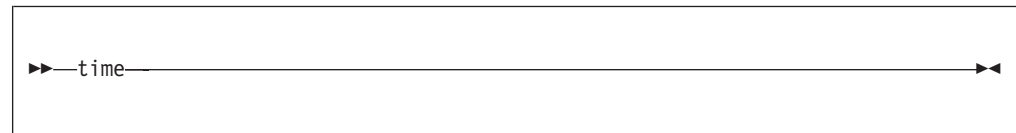
SC_MAX_THREADS_NP and SC_MAX_THREAD_TASKS_NP return the limits defined for the caller's process, not the systemwide limits.

Example

To determine the maximum number of files that a single process can have open at one time:

```
"sysconf" sc_open_max
maxopenfiles = retval /*value of desired configurable system
                      variable is returned in retval*/
```

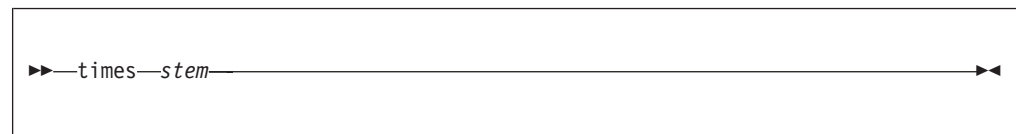
time



Function

time returns in RETVAL the time in POSIX format (seconds since the Epoch, 00:00:00 on 1 January 1970). You can use **gmtime** to convert it to other forms.

times



Function

times invokes the `times` callable service to collect information about processor time used by the current process or related processes. The elapsed time since the process was dubbed is returned in RETVAL. This value is of the type `clock_t`, which needs to be divided by `sysconf(_SC_CLK_TK)` to convert it to seconds. For z/OS UNIX, this value is expressed in hundredths of a second.

Parameters

stem

The name of a stem variable used to return the information. Upon return,

times

stem.0 contains the number of variables returned. Four variables are returned in the stem. To access the stem variables, use a numeric value or the predefined variables beginning with TMS_ used to derive that numeric value. (For the numeric values, see Appendix A, "REXX predefined variables," on page 241.) For example, you could specify *stem.4* or *stem.tms_cstime* to obtain system CPU values:

Variable	Description
TMS_CSTIME	The sum of system CPU time values and child system CPU time values for all waited-for child processes that have terminated. Zero if the current process has no waited-for children.
TMS_CUTIME	The sum of user CPU time values and child user CPU time values for all waited-for child processes that have terminated. Zero if the current process has no waited-for children.
TMS_STIME	The system CPU time of current process in hundredths of a second. This is the task control block (TCB) time accumulated while running in the kernel address space.
TMS_UTIME	The user CPU time of current process in hundredths of a second. This includes the TCB and service request block (SRB) time of the calling process minus the TCB time accumulated while running in the kernel address space.

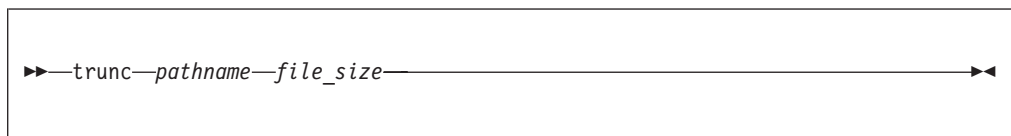
Usage notes

Processor times for a child process that has ended are not added to the TMS_CUTIME and TMS_CSTIME of the parent process until the parent issues a wait or waitpid for that child process.

Example

```
"times tm."
```

trunc



Function

trunc invokes the trunc callable service to change the size of the file identified by *pathname*.

Parameters

pathname

The pathname of the file.

file_size

The new size of the file, in bytes.

Usage notes

1. The file specified must be a regular file to which the calling process has write access.
2. The file size changes beginning from the first byte of the file. If the file was previously larger than the new size, the data from *file_size* to the original end of the file is removed. If the file was previously shorter than *file_size*, bytes between the old and new lengths are read as zeros.
3. If *file_size* is greater than the current file size limit for the process, the request fails with EFBIG, and the SIGXFSZ signal is generated for the process.

Example

To set the file size of `/tmp/xxx` to 1000 bytes:

```
"trunc /tmp/xxx 1000"
```

ttyname



```
▶▶ ttyname -fd -variable ◀◀
```

Function

ttyname invokes the `ttyname` callable service to obtain the pathname of the terminal associated with the file descriptor.

Parameters

fd The file descriptor (a number) for the character special file for the terminal.

variable

The name of the variable that stores the pathname for the character special file for the terminal.

Usage notes

This service does not return -1 to indicate a failure. If the file descriptor is incorrect, a null string is returned.

Example

To obtain the pathname for file descriptor 0:

```
"ttyname 0 path"
```

umask



```
▶▶ umask -mask ◀◀
```

Function

umask invokes the umask callable service to change your process's file mode creation mask. The file mode creation mask is used by the security package (RACF) to turn off permission bits in the mode parameter specified. Bit positions that are set in the file mode creation mask are cleared in the mode of the created file.

Parameters

mask

A permission bit mask that you specify as a three-digit number. Each digit must be in the range 0 to 7, and all digits must be specified. For more information on permissions, see Appendix B, "Setting permissions for files and directories," on page 253.

Usage notes

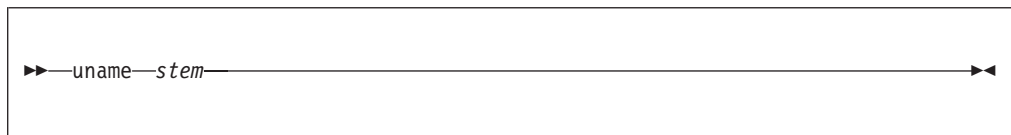
1. The umask service changes the process's file creation mask. This mask controls file permission bits that are set whenever the process creates a file. File permission bits that are turned on in the file creation mask are turned off in the file permission bits of files created by the process. For example, if a call to the open service, BPX1OPN, specifies a *mode* argument with file permission bits, the process's file creation mask affects that argument: bits that are on in the mask will be turned off in the *mode* argument, and therefore in the mode of the created file.
2. Only the file permission bits of new mask are used.

Example

To create a mask that sets read-write-execute permission on for the owner of the file and off for everyone else:

```
"umask 077"
```

uname



Function

uname invokes the uname callable service to obtain information about the system you are running on.

Parameters

stem

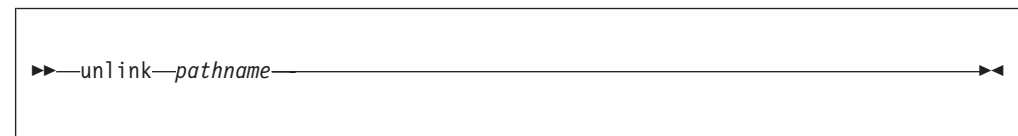
The name of a stem variable used to return the information. Upon return, *stem.0* contains the number of variables returned. To access the information, you can specify a numeric value (see Appendix A, "REXX predefined variables," on page 241) or the predefined variables beginning with `U_` that derive the appropriate numeric value. For example, both *stem.1* and *stem.u_sysname* access the name of the operating system.

Variable	Description
U_MACHINE	The name of the hardware type on which the system is running
U_NODENAME	The name of this node within the communication network
U_RELEASE	The current release level of this implementation
U_SYSNAME	The name of this implementation of the operating system (z/OS)
U_VERSION	The current version level of this release

Example

```
"uname uts."
```

unlink



Function

unlink invokes the unlink callable service to remove a directory entry.

Parameters

pathname

A pathname for the directory entry. The directory entry could be identified by a pathname for a file, the name of a hard link to a file, or the name of a symbolic link.

Usage notes

1. If the name specified refers to a symbolic link, the symbolic link file named by *pathname* is deleted.
2. If a file is deleted (that is, if the unlink service request is successful and the link count becomes zero), the contents of the file are discarded, and the space it occupied is freed for reuse. However, if another process (or more than one) has the file open when the last link is removed, the file is not removed until the last process closes it.
3. When the unlink service is successful in removing the directory entry and decrementing the link count, whether or not the link count becomes zero, it returns control to the caller with RETVAL set to 0. It updates the change and modification times for the parent directory, and the change time for the file itself (unless the file is deleted).
4. Directories cannot be removed using unlink. To remove a directory, refer to "rmdir" on page 117.

Example

In the following example, assume that *file* was assigned a value earlier in the exec:

```
"unlink (file)"
```

unmount



Function

unmount invokes the unmount callable service to unmount a file system; that is, it removes a file system from the file hierarchy.

Requirement: The caller must have unmount authorities to unmount a file system. See the section on mount authority in *z/OS UNIX System Services Planning*.

Parameters

name

The name of the file system to be unmounted, specified as the name of an HFS data set. You must specify the HFS data set name as a fully qualified name in uppercase letters. Do not enclose the data set name in single quotes.

flags

The unmount options, expressed as a numeric value. You can specify a numeric value (see Appendix A, "REXX predefined variables," on page 241) or the predefined variable beginning with MTM_ used to derive the appropriate numeric value:

Variable	Description
MTM_DRAIN	An unmount drain request. All uses of the file system are normally ended before the file system is unmounted.
MTM_FORCE	An unmount force request. The file system is unmounted immediately, forcing any users of the named file system to fail. All data changes made up to the time of the request are saved. If there is a problem saving data, the unmount continues and the data may be lost. So that data will not be lost, you must issue an unmount immediate request before an unmount force request.
MTM_IMMED	An unmount immediate request. The file system is unmounted immediately, forcing any users of the named file system to fail. All data changes made up to the time of the request are saved. If there is a problem saving data, the unmount request fails.
MTM_NORMAL	A normal unmount request. If no one is using the named file system, the unmount request is done. Otherwise, the request is rejected.
MTM_RESET	A reset unmount request. This stops a previous unmount drain request.
MTM_REMOUNT	Unmounts the file system, changes the mount mode, and remounts the file system. A read/write mount mode changes to read-only. A read-only mount mode changes to read/write.
MTM_SAMEMODE	Remounts the file system without changing the mount mode. This can be used to regain the use of a file system that has I/O errors.

Usage notes

1. A file system that has file systems mounted on it cannot be unmounted. Any child file systems must be unmounted first.
2. A reset request can stop only an unmount service drain request. There is no effect if it is issued when there is no unmount request outstanding.

Example

To request a normal unmount of the file system HFS.USR.CRISP:

```
"unmount HFS.USR.CRISP" mtm_normal
```

unquiesce

```
►► unquiesce name flag ◄◄
```

Function

unquiesce invokes the unquiesce callable service to unquiesce a file system, making the files in it available for use again. You must be a superuser to use this function.

Parameters

name

The name of the file system to be unquiesced. You must specify an HFS data set name as a fully qualified name in uppercase letters. Do not enclose the data set name in single quotes.

flag

A number specifying the type of unquiesce:

- 0 Normal unquiesce
- 1 Forced unquiesce. Request is allowed even if the requester process is not the process that made the quiesce request.

Usage notes

An unquiesce service makes a file system available for use again following a previous quiesce request.

Example

To request a normal unquiesce of the file system HFS.USR.ELIZAB:

```
"unquiesce HFS.USR.ELIZAB 0"
```

utime

```
►► utime pathname atime mtime ◄◄
```

utime

Function

utime invokes the utime callable service to set the access and modification times of a file.

Parameters

pathname

A pathname for the file.

atime

A numeric value for the new access time for the file, specified as POSIX time (seconds since the Epoch, 00:00:00 1 January 1970).

mtime

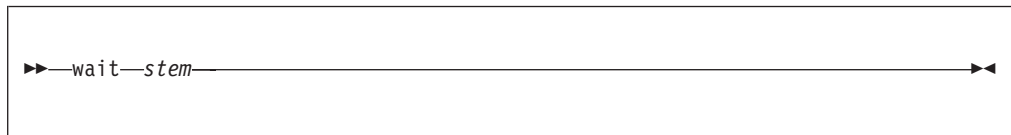
A numeric value for the new modification time for the file, specified as POSIX time (seconds since the Epoch, 00:00:00 1 January 1970).

Example

In the following example, assume that *file*, *atm*, and *mtm* were assigned values earlier in the exec:

```
"utime (file)" atm mtm
```

wait



Function

wait invokes the wait callable service to obtain the status of any child process that has ended or stopped. You can use the wait service to obtain the status of a process that is being debugged with the ptrace facilities. The term *child* refers to a child process created by a fork as well as a process attached by ptrace.

Parameters

stem

The name of a stem variable used to return the information. Upon return, *stem.0* contains the number of variables returned. To access the information in the stem variables, you can specify a numeric value (see Appendix A, "REXX predefined variables," on page 241) or the predefined variables beginning with *W_* that derive the appropriate numeric value.

Variable	Description
W_CONTINUED	Process continued from stop
W_EXITSTATUS	The exit status of the child process
W_IFEXITED	The child process ended normally
W_IFSIGNALED	The child process ended because of a signal that was not caught
W_IFSTOPPED	Wait if the child process is stopped

Variable	Description
W_STAT3	Byte 3 of the BPXYWAST macro. See BPXYWAST — Map the Wait Status Word in <i>z/OS UNIX System Services Programming: Assembler Callable Services Reference</i> .
W_STAT4	Byte 4 of the BPXYWAST macro. See BPXYWAST — Map the Wait Status Word in <i>z/OS UNIX System Services Programming: Assembler Callable Services Reference</i> .
W_STOPSIG	The signal number that caused the child process to stop
W_TERMSIG	The signal number that caused the child process to end

Usage notes

1. The wait service suspends execution of the calling thread until one of the requested child or debugged processes ends or until it obtains information about the process that ended. If a child or debugged process has already ended but its status has not been reported when wait is called, the routine immediately returns with that status information to the caller.
2. If the WUNTRACED option is specified, the foregoing also applies for stopped children or stopped debugged processes.
3. The wait service always returns status for stopped *debugged* processes, even if WUNTRACED is not specified.
4. If status is available for one or more processes, the order in which the status is reported is unspecified.

Note: A debugged process is one that is being monitored for debugging purposes with the ptrace service.

Example

See “Set up a signal to enforce a time limit for a program” on page 171 for an example of signal coding that interprets the stem this returns:

```
"wait wstat."
```

waitpid

```
►►—waitpid—pid—stem—option—◄◄
```

Function

waitpid invokes the wait callable service to obtain the status of a child process that has ended or stopped. You can use the wait service to obtain the status of a process that is being debugged with the ptrace facilities. The term *child* refers to a child process created by a fork as well as a process attached by ptrace.

Parameters

pid

A numeric value indicating the event the caller is waiting upon:

- A value greater than zero is assumed to be a process ID. The caller waits for the child or debugged process with that specific process ID to end or to stop.
- A value of zero specifies that the caller is waiting for any children or debugged processes with a process group ID equal to the caller's to end or to stop.
- A value of -1 specifies that the caller is waiting for any of its children or debugged processes to end or to stop.
- If the value is negative and less than -1, its absolute value is assumed to be a process group ID. The caller waits for any children or debugged processes with that process group ID to end or to stop.

stem

The name of a stem variable used to return the information. Upon return, *stem.0* contains the number of variables returned. To access the information in the stem variables, you can use numeric values (see Appendix A, "REXX predefined variables," on page 241), or the predefined variables beginning with *W_* (see their description in "wait" on page 154).

options

A numeric value or its equivalent predefined variable beginning with *W_* that indicates the wait options for this invocation of the wait service. These options can be specified separately or together. (For the numeric values, see Appendix A, "REXX predefined variables," on page 241.)

Variable	Description
0	Wait for the child process to end (default processing)
W_NOHANG	The wait service does not suspend execution of the calling process if status is not immediately available for one of the child processes specified by <i>Process_ID</i> .
W_UNTRACED	The wait service also returns the status of any child processes specified by <i>Process_ID</i> that are stopped, and whose status has not yet been reported since they stopped. If this option is not specified, the wait service returns only the status of processes that end.

Usage notes

1. Use **waitpid** when you want to wait for a specified child process. The *pid* argument specifies a set of child processes for which status is requested; **waitpid** returns the status of a child process from this set.
2. The waitpid service suspends execution of the calling thread until one of the requested child or debugged processes ends or until it obtains information about the process that ended. If a child or debugged process has already ended but its status has not been reported when waitpid is called, the routine immediately returns with that status information to the caller.
3. If the WUNTRACED option is specified, the foregoing also applies for stopped children or stopped debugged processes. A debugged process is one that is being monitored for debugging purposes with the ptrace service.
4. The wait service always returns status for stopped *debugged* processes, even if WUNTRACED is not specified.

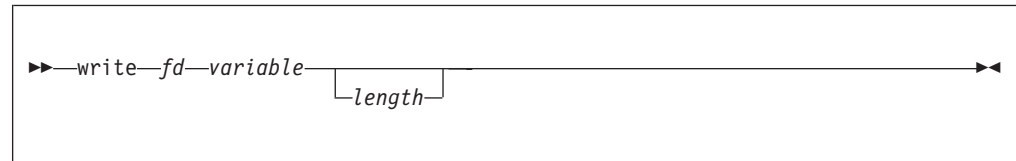
- If status is available for one or more processes, the order in which the status is reported is unspecified.

Example

See “Set up a signal to enforce a time limit for a program” on page 171 for an example of signal coding that interprets the stem this returns:

```
"waitpid -1 wst. 0"
```

write



Function

write invokes the write callable service to copy data to a buffer and then write it to an open file. The number of bytes written is returned in RETVAL.

Refer to *z/OS UNIX System Services Programming: Assembler Callable Services Reference* for restrictions when running in a conversion environment.

Parameters

fd The file descriptor (a number) for a file.

variable

The name of the variable that is to store the data to be written to the file.

length

The number of bytes to be written to the file identified by *fd*. If you want a length longer than 4096 bytes, specify the *length* parameter. If you do not specify *length*, the length of *variable* is used, up to a maximum length of 4096 bytes. A *variable* longer than 4096 bytes is truncated to 4096, and RC is set to 4.

Usage notes

- Within the variable, you can use the predefined variables beginning with ESC_ the same way you use C language escape sequences to avoid code page dependence with some control characters. For example:

```
buf='line 1' || esc_n
```

appends a newline character to the string 'line 1'.

ESC_A

Alert (bell)

ESC_B

Backspace

ESC_F

Form feed (new page)

ESC_N

Newline

ESC_R
Carriage return

ESC_T
Horizontal tab

ESC_V
Vertical tab

2. Return codes:
 - 4 indicates one of these:
 - If *length* was specified, the number of characters specified by *length* is not the same as the length of *variable*. The data is truncated or padded as required. The characters used for padding are arbitrarily selected.
 - If *length* was not specified, 4 indicates that a variable longer than 4096 bytes was truncated to 4096.
 - -24 indicates that storage could not be obtained for the buffer.
3. File offset: If *fd* specifies a regular file or any other type of file on which you can seek, the write service begins writing at the file offset associated with that file descriptor. A successful write operation increments the file offset by the number of bytes written. If the incremented file offset is greater than the previous length of the file, the file is extended: the length of the file is set to the new file offset.

If the file descriptor refers to a file on which you cannot seek, the service begins writing at the current position. No file offset is associated with such a file.

If the file was opened with the append option, the write routine sets the file offset to the end of the file before writing output.

4. Number of bytes written: Ordinarily, the number of bytes written to the output file is the number you specify in the *length* parameter. (This number can be zero. If you ask to write zero bytes, the service simply returns a return value of zero, without attempting any other action.)

If the *length* you specify is greater than the remaining space on the output device, fewer bytes than you requested are written. When at least 1 byte is written, the write is considered successful. The return value shows the number of bytes written. An attempt to write again to the same file, however, causes an ENOSPC error unless you are using a pseudoterminal. With a pseudoterminal, if there is not enough room in the buffer for the whole write, the number of bytes that can fit are written, and the number of bytes written is returned. However, on the next write (assuming the buffer is still full) there is a block or EAGAIN is returned, depending on whether the file was opened blocking or nonblocking.

Similarly, fewer bytes are written if the service is interrupted by a signal after some but not all of the specified number of bytes are written. The return value shows the number of bytes written. If no bytes were written before the routine was interrupted, the return value is -1 and an EINTR error is reported.

5. The write service causes signal SIGTTOU to be sent under all the following conditions:
 - The process is attempting to write to its controlling terminal.
 - TOSTOP is set as a terminal attribute.
 - The process is running in a background process group.
 - The SIGTTOU signal is not blocked or ignored.
 - The process is not an orphan.

If all the conditions are met, SIGTTOU is sent.

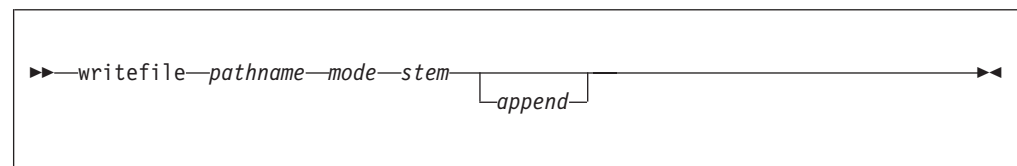
6. Write requests to a pipe (FIFO) are handled in the same as write requests to a regular file, with the following exceptions:
- There is no file offset associated with a pipe; each write request appends to the end of the pipe.
 - If the size of the write request is less than or equal to the value of the PIPE_BUFF variable (described in the pathconf service), the write is guaranteed to be atomic. The data is not interleaved with data from other write processes on the same pipe. If the size of the write request is greater than the value of PIPE_BUFF, the data can be interleaved, on arbitrary boundaries, with writes by other processes, whether or not the O_NONBLOCK flag is set.

Example

In the following example, assume that *fd* and *buf* were assigned values earlier in the exec:

```
"write" fd "buf"
```

writefile



Function

`writefile` invokes the open, write, and close callable services to write or append text files, with lines up to 1024 characters long.

Parameters

pathname

A pathname for the file.

mode

A three- or four-digit number, corresponding to the access permission bits. Each digit must be in the range 0–7, and at least three digits must be specified. For more information on permissions, see Appendix A, “REXX predefined variables,” on page 241.

stem

The name of a stem variable that contains the information to be written to the file. *stem.0* is set to the number of lines to be written. *stem.1* through *stem.n*, where *n* is the total number of variables written, each contain a line to be written. A newline is written to the file following each line.

Within the stem, you can use the predefined variables beginning with ESC_ the same way you use C language escape sequences to avoid code page dependence with some control characters. See the usage notes for “write” on page 157.

append

An optional flag to indicate that the information is to be appended to the file (the file must already exist), as follows:

writefile

- 0 Do not append. (This is the default value.)
- 1 Append the information to the file.

Usage notes

File I/O stops when **writefile** sets a return code. **writefile** can set the following return codes:

- 4 A line is longer than 1024 characters.
- 8 A write was attempted but not all of the data could be written. This situation typically happens when the file system runs out of space.

For further usage notes, see:

- “close” on page 40
- “open” on page 102
- “write” on page 157

Example

In the following example, assume that *fname* and the stem *file.* were assigned values earlier in the exec:

```
"writefile (fname) 600 file."
```

Chapter 4. Examples: Using syscall commands

The examples in this chapter are provided to assist you with coding REXX programs that use z/OS UNIX syscall commands.

The first four examples can be run in TSO/E, batch, or from the z/OS shells. They begin with call `syscalls 'ON'`.

Read the root directory into a stem and print it

This example prints the contents of the root directory to standard output:

```
/* rexx */
call syscalls 'ON'
address syscall
'readdir / root.'
do i=1 to root.0
  say root.i
end
```

The following line saves the results from the previous example in a text file:

```
'writefile /u/schoen/root.dir 777 root.'
```

Open, write, and close a file

```
/* rexx */
call syscalls 'ON'
address syscall
path='/u/schoen/my.file'
'open' path,
      0_rdw+0_creat+0_trunc,
      660
if retval=-1 then
  do
  say 'file not opened, error codes' errno errnojr
  return
  end
fd=retval
rec='hello world' || esc_n
'write' fd 'rec' length(rec)
if retval=-1 then
  say 'record not written, error codes' errno errnojr
'close' fd
```

Open a file, read, and close it

```
/* rexx */
call syscalls 'ON'
address syscall
path='/u/schoen/my.file'
'open (path)',
      0_rdonly,
      000
if retval=-1 then
  do
  say 'file not opened, error codes' errno errnojr
  return
  end
fd=retval
'read' fd 'bytes 80'
```

```

if retval=-1 then
  say 'bytes not read, error codes' errno errnojr
else
  say bytes
'close' fd

```

List all users and groups

```

/* rexx */
call syscalls 'ON'
address syscall
say "users:"
do forever
  "getpwent pw."
  if retval=0 | retval=-1 then
    leave
  say pw.pw_name
end
say "groups:"
do forever
  "getgrent gr."
  if retval=0 | retval=-1 then
    leave
  say gr.gr_name
end
return 0

```

Display the working directory and list a specified directory

This REXX program runs in the z/OS shells; it uses both the SH and SYSCALL environments. The program identifies your working directory and lists the contents of a directory that you specify as a parameter or after a prompt.

```

/* rexx */
parse arg dir
address syscall 'getcwd cwd'
say 'current directory is' cwd
if dir=' ' then
  do
    say 'enter directory name to list'
    parse pull dir
  end
'ls -l' dir
return

```

Parse arguments passed to a REXX program: the getopt function

The following simple utility function is used by some of the examples to parse the arguments to a REXX program that is run from a shell. This function parses the stem __argv for options in the format used by most POSIX commands.

```

/*****
/* Function: getopt
/* This function parses __argv. stem for options in the format */
/* used by most POSIX commands. This supports simple option */
/* letters and option letters followed by a single parameter. */
/* The stem OPT. is setup with the parsed information. The */
/* options letter in appropriate case is used to access the */
/* variable: op='a'; if opt.op=1 then say 'option a found' */
/* or, if it has a parameter: */
/*   op='a'; if opt.op<>' then say 'option a has value' opt.op */
/*
/* Parameters: option letters taking no parms */
/*               option letters taking 1 parm
/*

```

```

/* Returns: index to the first element of __argv. that is not */
/*          an option. This is usually the first of a list of files.*/
/*          A value of 0 means there was an error in the options and */
/*          a message was issued. */
/*          */
/* Usage: This function must be included in the source for the exec */
/*          */
/* Example: the following code segment will call GETOPTS to parse */
/*          the arguments for options a, b, c, and d. Options a */
/*          and b are simple letter options and c and d each take */
/*          one argument. It will then display what options were */
/*          specified and their values. If a list of files is */
/*          specified after the options, they will be listed. */
/*          */
/* parse value 'a b c d' with, */
/*          lca lcb lcc lcd . */
/* argx=getopts('ab','cd') */
/* if argx=0 then exit 1 */
/* if opt.0=0 then */
/*   say 'No options were specified' */
/* else */
/*   do */
/*     if opt.lca<>' then say 'Option a was specified' */
/*     if opt.lcb<>' then say 'Option b was specified' */
/*     if opt.lcc<>' then say 'Option c was specified as' opt.lcc */
/*     if opt.lcd<>' then say 'Option d was specified as' opt.lcd */
/*   end */
/* if __argv.0>=argx then */
/*   say 'Files were specified:' */
/* else */
/*   say 'Files were not specified' */
/* do i=argx to __argv.0 */
/*   say __argv.i */
/* end */
/*          */
/*          */
/*****/
getopts: procedure expose opt. __argv.
  parse arg arg0,arg1
  argc=__argv.0
  opt.='
  opt.0=0
  optn=0
  do i=2 to argc
    if substr(__argv.i,1,1)<>'-' then leave
    if __argv.i='-' then leave
    opt=substr(__argv.i,2)
    do j=1 to length(opt)
      op=substr(opt,j,1)
      if pos(op,arg0)>0 then
        do
          opt.op=1
          optn=optn+1
        end
      else
        if pos(op,arg1)>0 then
          do
            if substr(opt,j+1)<>' then
              do
                opt.op=substr(opt,j+1)
                j=length(opt)
              end
            else
              do
                i=i+1
                if i>argc then
                  do
                    say 'Option' op 'requires an argument'

```

```

        return 0
    end
    opt.op=__argv.i
end
optn=optn+1
end
else
do
say 'Invalid option =' op
say 'Valid options are:' arg0 arg1
return 0
end
end
end
opt.0=optn
return i

```

Count newlines, words, and bytes

This is an example of a REXX program that can run as a shell command or filter. It is a REXX implementation of the `wc` (word count) utility supporting the options `-c`, `-w`, and `-l`.

The program uses `open`, `close`, and `EXECIO`. To read standard input, it accesses the file `/dev/fd0`.

```

/* rexx */
parse value 'l w c' with,
    lcl lcw lcc .          /* init lower case access vars */
argx=getopts('lwc')      /* parse options */
if argx=0 then return 1   /* return on error */
if opt.0=0 then          /* no opts, set defaults */
    parse value 'l 1 1' with,
        opt.lcl opt.lcw opt.lcc .
if __argv.0>argx then    /* multiple files specified */
    single=0
else
if __argv.0=argx then    /* one file specified */
    single=1
else
do                       /* no files specified, use stdin */
    single=2
    __argv.argx='/dev/fd0' /* handle it like fd0 specified */
    __argv.0=argx
end
parse value '0 0 0' with,
    twc tcc tlc .        /* clear total counters */
address syscall
do i=argx to __argv.0    /* loop through files */
    fi=__argv.i           /* get file name */
    parse value '0 0 0' with,
        wc cc lc .      /* clear file counters */
    'open (fi)' o_rdnly 000 /* open the file */
    fd=retval
    if fd=-1 then        /* open failed */
        do
            say 'unable to open' fi
            iterate
        end
    do forever           /* loop reading 1 line at a time */
        address mvs 'execio 1 disk' fd '(stem LN.'
        if rc<>0 | ln.0=0 then leave /* error or end of file */
        if opt.lcw=1 then

```

```

        wc=wc+words(ln.1)           /* count words in line      */
    if opt.lcc=1 then
        cc=cc+length(ln.1)+1      /* count chars in line + NL char */
    if opt.lcl=1 then
        lc=lc+1                    /* count lines              */
    end
    'close' fd                      /* close file                */
    twc=twc+wc                      /* accumulate total words    */
    tlc=tlc+lc                      /* accumulate total lines    */
    tcc=tcc+cc                      /* accumulate total chars    */
    if opt.lcw1 then wc=''          /* format word count        */
        else wc=right(wc,7)
    if opt.lcl<>1 then lc=''        /* format line count        */
        else lc=right(lc,7)
    if opt.lcc<>1 then cc=''        /* format char count        */
        else cc=right(cc,7)
    if single=2 then fi=''         /* if stdin used clear filename */
    say lc wc cc ' ' fi            /* put out counts message   */
end
if single=0 then                  /* if multiple files specified */
do                                  /* format and output totals line */
    if opt.lcw<>1 then twc=''
        else twc=right(twc,7)
    if opt.lcl<>1 then tlc=''
        else tlc=right(tlc,7)
    if opt.lcc<>1 then tcc=''
        else tcc=right(tcc,7)
    say tlc twc tcc ' total'
end
return 0

```

Obtain information about the mounted file system

This REXX program uses **getmntent** and **statfs** to list all mount points, the name of the mounted file system, and the available space in the file system. This sample REXX program, as coded, must be run from a shell.

```

/* rexx */
address syscall
numeric digits 12
'getmntent m.'
do i=1 to m.0
    'statfs' m.mnte_fsname.i 's.'
    j=s.stfs_avail * s.stfs_blocksize
    if length(m.mnte_path.i)>20 then
        say m.mnte_path.i ' ' strip(m.mnte_fsname.i) '('j')'
    else
        say left(m.mnte_path.i,20) strip(m.mnte_fsname.i) '('j')'
end

```

Mount a file system

This REXX program uses **mount** to mount a file system, an action that requires superuser authority. The name of this program is **mountx** and it is in **/samples**.

The syntax is:

```
mountx pathname fsn options
```

where *pathname* is the name of the directory where the file system is to be mounted, and *fsn* is the name of the HFS data set. The *options* are:

-p parm

Parameter data, in the form of a single string.

- r** Mount file system as read-only.
- t type** File system type. The default file system type is HFS. If the file system type is HFS, the program changes the file system name to uppercase.

The path name and the file system name can be specified in any order. **stat** is used to determine which name is the path name.

For the **getopts** function called in this program, see "Parse arguments passed to a REXX program: the getopts function" on page 162.

```

/* rexx */
/*****
/* Determine which options were specified. */
/*****
lcp='p'
lcr='r'
lct='t'
ix=getopts('r','pt')
/*****
/* If -r specified, mount file system read-only. */
/*****
if opt.lcr<>' then
    mtm=mtm_rdonly
else
    mtm=mtm_rdwrr
/*****
/* If -t 'name' specified, direct mount request to file system 'name'.*/
/* Otherwise, direct mount request to file system named HFS. */
/*****
if opt.lct<>' then
    type=translate(opt.lct)
else
    type='HFS'

/*****
/* Complain if required parameters are missing. */
/*****
if __argv.0<>ix+1 then
do
    say 'Pathname and file system name are required, and '
    say 'they must follow the options: MOUNT <options> <pathname> <fsn>'
    return 1
end

/*****
/* Direct function calls to REXX z/OS UNIX services. */
/*****
address syscall
/*****
/* Determine which of the parameters is the mount point name */
/* (it must be a pathname), and which is the file system to be */
/* mounted. */
/*****
'stat (__argv.ix) st.'
if st.st_type<>s_isdir then
do
    fsn=__argv.ix
    ix=ix+1
    path=__argv.ix
end
else
do
    path=__argv.ix
    ix=ix+1
    fsn=__argv.ix

```



```

    end
'stat (path) st.'
if st.st_type<>s_isdir then
do
    say "Can't figure out pathname, neither name is a directory:"
    say path
    say fsn
    return 1
end

/*****
/* HFS file system requires mounted file systems to be data sets, */
/* so translate the file system name to upper case. */
/*****
if type='HFS' then
    fsn=translate(fsn)
/*****
/* Mount the file system. */
/* Return code Return code Meaning */
/* from system to caller */
/* 0 0 Success, file system now mounted */
/* 1 2 Success, file system mount in progress */
/* -1 1 Error, explained by ERRNO and reason code*/
/*****
"mount (path) (fsn) (type) (mtm) (opt.lcp)"
select
when retval= 0 then
do
    say fsn 'is now mounted at'
    say path
    return 0
end
when retval= 1 then
do
    say fsn 'will be mounted asynchronously at'
    say path
    return 2
end
otherwise
do
    say 'Mount failed:'
    say ' Error number was' errno'x('x2d(errno)')'
    say ' Reason code was ' right(errnojr,8,0)
    return 1
end
end
end

```

Unmount a file system

This REXX program uses **unmount** to unmount a file system, an action that requires mount authority as described in the section on mount authority in *z/OS UNIX System Services Planning*. The name of this program is **unmountx** and it is in **/samples**.

The syntax is:

```
unmountx name
```

or

```
unmountx -t filesystem
```

where:

- *name* is the path name where the file system is mounted, or the name of an HFS data set.

- *filestype* is the type name of the physical file system (PFS).

The program assumes that the case of the requested file system name is entered correctly in uppercase. If the unmount fails, it folds the file system name to uppercase and retries the unmount.

```

/*****
/* Direct commands to REXX z/OS UNIX services.          */
/*****

address syscall

/*****
/* Verify that exactly one operand or a pfs type was specified.  */
/*****
if __argv.0=3 & __argv.2='-t' then
  return fstype(__argv.3)
if __argv.0<>2 then
  do
    say 'Syntax:  unmount <name>  or  unmount -t <filestype>'
    return 2
  end
/*****
/* Determine if the name is a pathname.  If so, determine file system */
/* name via stat().  Otherwise, use the name as entered.          */
/*****
'stat (__argv.2) st.'
if retval =0 & st.st_type=s_isdir then
  do
    getmntent mnt. x2d(st.st_dev)
    fsn=mnt.mnt_fsname.1
  end
else
  fsn=__argv.2
/*****
/* Unmount the file system, trying both the name as entered      */
/* and the name uppercased, since the HFS file system requires   */
/* mounted file systems to be data sets.                          */
/*                                                                */
/* Return code  Return code  Meaning                               */
/* from system  to caller                                         */
/* Not -1      0           Success, file system unmount complete */
/* -1         1           Error, explained by ERRNO and reason code*/
/*****
"unmount (fsn)" mtm_normal /* unmount name as specified          */
if retval =0 then
  do
    fsn=translate(fsn) /* if fails, upcase name and retry      */
    "unmount (fsn)" mtm_normal
  end
if retval<>-1 then
  do
    say 'Unmount complete for' fsn
    return 0
  end
else
  do
    say 'Unmount failed:'
    say ' Error number was' errno'x('x2d(errno))'
    say ' Reason code was ' right(errnojr,8,0)
    return 1
  end
/*****
/* Unmount all file systems for a PFS type                        */
/*****
fstype:
  arg name . /* make upper case          */

```

```

/*****
/* Loop through mount table until unable to do any unmounts.      */
/* This handles cascaded mounts.                                  */
/*****
do until didone=0
  didone=0
  'getmntent m.'
  do i=1 to m.0
    if m.mnte_fstype.i=name then
      do
        "unmount (m.mnte_fsname.i)" mtm_normal
        if retval<>-1 then
          do
            didone=1
            say 'unmounted:' m.mnte_fsname.i
          end
        end
      end
    end
  end
end
/*****
/* Make one more pass and show errors for failed unmounts      */
/*****
goterr=0
'getmntent m.'
do i=1 to m.0
  if m.mnte_fstype.i=name then
    do
      "unmount (m.mnte_fsname.i)" mtm_normal
      if retval=-1 then
        do
          say 'Unmount failed for' m.mnte_fsname.i ':'
          say ' Error number was' errno'x('x2d(errno)')'
          say ' Reason code was ' right(errnojr,8,0)
          goterr=1
        end
      else
        say 'unmounted:' m.mnte_fsname.i
      end
    end
  end
end
return goterr

```

Run a shell command and read its output into a stem

This REXX program runs the `ls` shell command and reads the output from the command into a stem. The program uses `pipe`, `close`, and `EXECIO`. It accesses `/dev/fdn`, where *n* is a number that the `exec` concatenates to `/dev/fd`.

Note: You can use this example to trap output from commands when the output is relatively small (less than 100KB). For command output that could be larger, you should use the `spawn` service.

```

/* rexx */
address syscall 'pipe p.' /* make a pipe */
'ls>/dev/fd' || p.2 /* run the ls command and
                    redirect output to the
                    write end of the pipe */
address syscall 'close' p.2 /* close output side */
address mvs 'execio * diskr' p.1 '(stem s.' /* read data in pipe */
do i=1 to s.0 /* process the data */
  say s.i
end

```

Print the group member names

This REXX program uses **getgrgid**, **getgrnam**, and **write** to print the names of the users that are connected to a group. The group can be specified as either a GID or a group name.

```
/* rexx */
arg group .
address syscall
if datatype(group,'W')=1 then
  'getgrgid (group) gr.' /* use getgrgid if GID specified */
else
  'getgrnam (group) gr.' /* use getgrnam if group name specified */
if retval<=0 then /* check for error */
do
  say 'No information available for group' group
  return 1
end
say 'Connected users for group' strip(gr.gr_name)('gr.gr_gid')
j=1
do i=gr_mem to gr.0
  buf= ' gr.i
  if j//5=0 | i=gr.0 then
    buf=buf || esc_n /* write newline every 5 groups or at end */
  j=j+1
  'write 1 buf'
end
return 0
```

Obtain information about a user

This REXX program prints information from the user database for a user. The user can be specified as either a UID or a user name. The program uses **getpwuid**, **getpwnam**, **getgrgid**, **getgroupsbyname**, and **write**.

```
/* rexx */
arg user .
address syscall
if datatype(user,'W')=1 then
  'getpwuid (user) pw.' /* use getpwuid if UID specified */
else
  'getpwnam (user) pw.' /* use getpwnam if user name specified */
if retval<=0 then /* check for error */
do
  say 'No information available for user' user
  return 1
end
say 'Information for user' strip(pw.pw_name)('pw.pw_uid')
say ' Home directory: ' strip(pw.pw_dir)
say ' Initial program:' strip(pw.pw_shell)
'getgrgid' pw.pw_gid 'gr.'
if retval<=0 then /* check for error */
do
  say ' Group information not available'
  return 1
end
say ' Primary group: ' strip(gr.gr_name)('gr.gr_gid')
'getgroupsbyname' pw.pw_name 's.'
if retval<=0 then /* check for error */
do
  say ' Supplemental group information not available'
  return 1
end
say ' Supplemental GIDs:'
```

```

do i=1 to s.0
  buf=right(s,i,12)
  if i//5=0 | i=s.0 then
    buf=buf || esc_n      /* write newline every 5 groups or at end */
  'write 1 buf'
end
return 0

```

Set up a signal to enforce a time limit for a program

This REXX program runs `/bin/ls` to list files in the `/bin` directory, and sets up a signal that enforces a time limit of 10 seconds for the program to run:

```

/* REXX                                                                    */
address syscall                                                            */
/* initialize file descriptor map (see note 1)                              */
fd.0=-1                                                                    */
fd.2=-1                                                                    */
'creat /tmp/dirlist 600'
fd.1=retval
/* initialize parameter stem (see note 2)                                  */
parm.1='/bin/ls'                                                            */
parm.2='-l'
parm.3='/bin'
parm.0=3
/* spawn new process (see note 3)                                          */
'spawn /bin/ls 3 fd. parm. __environment.'
pid=retval
/* set up signals to wait up to 10 seconds (see note 4)                   */
call syscalls 'SIGON'
'sigaction' sigalrm sig_cat 0 'oldh oldf'
'sigprocmask' sig_unblock sigaddset(sigsetempty(),sigalrm) 'mask'
'alarm' 10 /* set alarm */
'waitpid (pid) st. 0' /* wait for process term or alarm */
srv=retval
'alarm 0' /* make sure alarm is now off */
if srv=-1 then /* if alarm went off */
  do
    'kill' pid sigkill /* cancel child process */
    'waitpid (pid) st. 0' /* wait for completion */
  end
call syscalls 'SIGOFF' /* turn off signals */
/* determine process status code (see note 5)                              */
select
  when st.w_ifexited then
    say 'exited with code' st.w_exitstatus
  when st.w_ifsignaled then
    say 'terminated with signal' st.w_termsig
  when st.w_ifstopped then
    say 'stopped by signal' st.w_stopsig
end
return
sigsetempty: return copies(0,64)
sigaddset: return overlay(1,arg(1),arg(2))

```

Note:

1. The file descriptor map is set up so that the file descriptor for the new file being created is remapped to file descriptor 1 (standard output) for the new process. File descriptors 0 and 2 will not be opened in the new process.
2. The first parameter is set to the path name for the file being spawned. Additional parameters are set in the format the program expects. In this case, they specify a long directory listing for the `/bin` directory.

3. The new process is spawned to run `/bin/ls`. File descriptors greater than or equal to 3 are not available to the new process. `fd.0` to `fd.2` are used in remapping file descriptors from the parent. The current environment for the parent is passed to the new process.
4. The `syscalls` function is called to enable signals. If this program were to be exec'd (run from the shell), this would not be necessary, and, similarly, the call later on to turn off signals would also be unnecessary.

`sigaction` is used to set the action for `sigalrm` to be caught. `sigprocmask` is used to unblock `sigalrm`. This call also uses `sigaddset` and `sigsetempty` to create a signal mask with the `sigalrm` bit.

The alarm is set by using the `alarm` service, and the process waits for completion of the child or the alarm. "ALARM 0" is used just to make sure the alarm is off.

5. A SELECT instruction is used on the status stem returned by the `waitpid` service. This determines if the process was terminated by a signal, if it exited, or if the process is stopped. If it exited, the exit status is available; otherwise, the signal number is available.

List the ACL entries for a file

This example will show the access ACL entries for a path name given as a parameter:

```

/* REXX */
parse arg path
call syscalls 'ON'
address syscall
'acclinit accl' /* init variable ACL to hold accl */
'acclget accl (path)' accl_type_access /* get the file access accl */
do i=1 by 1 /* get each accl entry */
  'acclgetentry accl accl.' i
  if rc<0 | retval=-1 then leave /* error, assume no more */
  parse value '- - -' with pr pw px
  if accl.accl_read=1 then pr='R' /* set rwx permissions */
  if accl.accl_write=1 then pw='W'
  if accl.accl_execute=1 then px='X'
  acclid=accl.accl_id /* get uid or gid */
  /* determine accl type */
  if accl.accl_entry_type=accl_entry_user then type='UID='
  else
    if accl.accl_entry_type=accl_entry_group then type='GID='
    else
      type='???'
  say pr || pw || px type || acclid /* print line */
end
'acclfree accl' /* must free accl buffer */

```

Chapter 5. z/OS UNIX REXX functions

The z/OS UNIX REXX functions extend the REXX language on z/OS when it is used in the z/OS UNIX REXX environment. Except for **bpxwunix()** and **syscalls()**, these functions must be run in a z/OS UNIX environment. The z/OS UNIX REXX functions include functions for standard REXX I/O, and for accessing common file services and environment variables.

All of the functions are fully enabled for large files (>2 GB).

All numbers that are used as input on the functions must be integers. The default precision for REXX is 9 digits. If arithmetic is used on large numbers, be sure to change your precision appropriately, using the **NUMERIC DIGITS** statement.

REXX I/O functions

The REXX input functions are **charin()** and **linein()**. The **chars()** and **lines()** functions determine if data remains in an input stream.

The REXX output functions are **charout()** and **lineout()**.

The REXX **stream()** function controls the processing of I/O streams (file streams and process streams) and obtains their status. A number of commands are used within the **stream()** function to control stream processing.

Opening a stream implicitly

File streams can be opened implicitly or explicitly. You open a stream implicitly by using a path name as the stream name for one of the six input or output functions. If the function is **charout()** or **lineout()**, the file is opened for output. If the file does not exist, it is created. For newly created files, the permission bits 0666 are applied to your process umask. The position for the first write is set to the end of file, unless the function call explicitly specifies a write location. If one of the four input functions is used, the file is opened for input.

When it is opened implicitly, a stream can be opened for both input and output. The input and output locations are independent of each other. If the stream is opened for both input and output, two file descriptors are used. The **stream close** command closes the stream and both file descriptors.

Opening a stream explicitly

You open a stream explicitly by using the **stream()** function.

To explicitly open a file stream, use the **open** command. The advantage of opening file streams explicitly is that the program can determine that the stream **open** failed. The program can also have several separate streams for the same file.

The **stream open** command fails if:

- The file does not exist.
- The path name cannot be accessed for either input or output.

If the file cannot be opened, a message is written to **stderr**. The I/O function returns as if the stream is empty, at end of file, and the file cannot be extended.

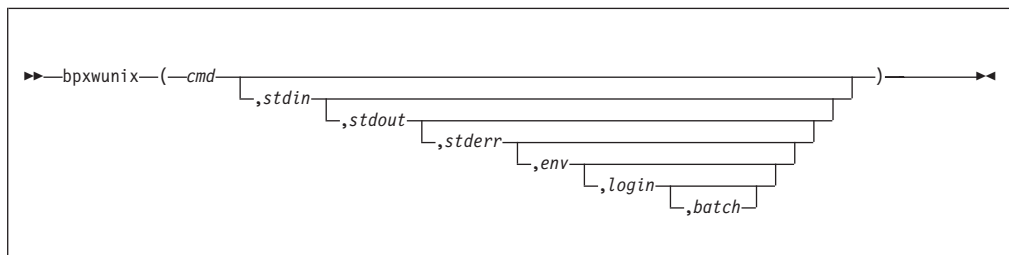
When a stream is explicitly opened, the **stream()** function returns a string that is the name of the stream. This is the only name that can be used to identify that stream. Multiple opens for the same path name open multiple streams, each with its own name.

Process streams

Process streams can only be opened explicitly, with the **popen** command. With process streams, you can run a shell command and provide its input or receive its output. To write the input for the command, use the **popen write** command. To receive the command's output, use the **popen read** command. The **stream()** function returns a string that is the name of the process stream. Use this name on the input functions for **popen read** or the output functions for **popen write**.

A process stream spawns **/bin/sh -c your_command**. The process inherits **stderr** and either **stdin** or **stdout**. If the process is opened for read, **stdout** for the process is a pipe; otherwise, **stdin** is a pipe. The shell completion code is returned as the result for the stream **pclose** command. **pclose** closes your end of the pipe and waits for the process to terminate.

bpxwunix()



Function

Runs a shell command and optionally:

- Provides its standard input (stdin)
- Traps its standard output (stdout)
- Traps its standard error (stderr)
- Exports a set of environment variables

Parameters

cmd

The shell command that you want to run. The shell is run as **/bin/sh -c** followed by the string you specify as the command.

stdin

An optional argument, `stdin` is the name of a compound variable (stem) that contains input for the command.

stdin.0 must contain the number of lines that are to be redirected to the command. **stdin.1**, **stdin.2**, ... contain the lines.

`stdin` can also be specified as:

- The string **STACK**, if the input is on the stack.
- **DD:ddname**, if the input is to be read from an allocated DD.

If this argument is not specified, your current `stdin` file is passed to the shell for `stdin`.

stdout

An optional argument, `stdout` is the name of a compound variable (stem) that, upon return, contains the normal output from the command. `stdout` is the number of lines output by the command. `stdin.1`, `stdin.2`, ... contain the output lines.

`stdout` can also be specified as:

- The string `STACK`, if the output is to be returned on the stack
- `DD:ddname`, if the output is to be written to an allocated `DD`

If `stdout` is not specified, your current `stdout` file is passed to the shell for `stdout`.

stderr

An optional argument, `stderr` is the name of a compound variable (stem) that, upon return, contains the error output from the command. `stderr.0` is the number of lines output by the command. `stdin.1`, `stdin.2`, ... contain the output lines. `stderr` is can also be specified as:

- The string `STACK`, if the output is to be returned on the stack
- `DD:ddname`, if the output is to be written to an allocated `DD`

If this argument is not specified, your current `stderr` file is passed to the shell for `stderr`.

env

An optional argument, `env` is the name of a compound variable (stem) that contains environment variables for the command. `env.0` must contain the number of environment variables to be passed to the command. `env.1`, `env.2`, ... contain the variables in the form `variable_name=variable_value`. If `env` is not specified, your current environment is passed to the shell for `stdin`.

login

An optional argument that specifies whether a login shell should be run.

- 0** A login shell is not used.
- 1** A login shell is used.

batch

An optional argument that specifies whether the command should run as a UNIX background process

- 0** The command runs in foreground synchronously with **bpxwunix**.
- 1** The command runs in background under `/bin/nohup`, asynchronously with **bpxwunix**. The `stdin`, `stdout`, and `stderr` arguments cannot be specified for a background process; however, standard shell redirection can be used. Any output from the **nohup** command is directed to the file **nohup.out**.

Usage notes

1. The **bpxwunix()** function runs the shell by passing a single command similar to **sh -c cmd**. It does not run a login shell.
2. **bpxwunix()** can be used outside of the z/OS UNIX REXX environment (for example, in TSO/E). In this case, `stdin`, `stdout`, `stderr`, and environment variables are not inherited from the current process environment. For example, when executing a REXX exec in this environment, you must either export the `PATH` statement before invoking the REXX exec (`command = 'export`

bpxwunix()

PATH;tsocmd time'), or supply the PATH statement to BPXWUNIX (env.1='PATH=/bin') in order for the REXX exec to execute properly. Otherwise, the REXX exec fails and message BPXW0000I is displayed.

3. If the stdout or stderr stems are specified, they are filled as appropriate. If the stdout or stderr stems are specified, **stem.0** contains the number of lines returned. The lines are returned in **stem.1, stem.2,....**
4. The DD names that are used for input and output are processed by the standard REXX input and output services. They have the same restrictions as REXX in terms of the types of allocations they can handle.
5. Standard output (stdout) lines cannot exceed 16383 characters.

Return codes

BPXWUNIX returns the following return codes:

- A return value in the range 0-255 is the exit status of the command.
- A negative return value indicates failure, and is usually a signal number. -256 is a general error code and a message is issued to the error stream describing the error.
- A number less than -1000 indicates a stop code.

Example

To trap output from the `ls` command and display it:

```
call bpxwunix 'ls -l',,out.
  do i=1 to out.0
    say out.i
  end
```

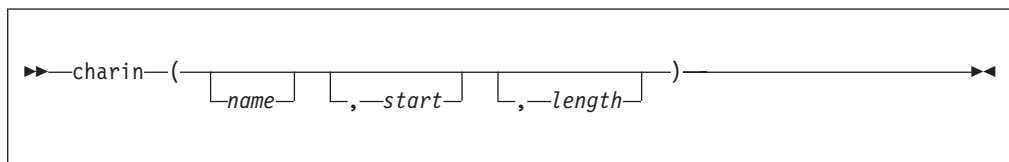
To send output from the above example to word count and print the byte count:

```
call bpxwunix 'wc',out.,bc.
  parse var bc.1 . . count
  say 'byte count is' count
```

To trap output on stack and feed it to word count:

```
if bpxwunix('ls -l',,stack)=0 then
  call bpxwunix 'wc',stack
```

charin()



Function

Returns a string of up to *length* characters read from the stream specified by *name*. The location for the next read is the current location increased by the number of characters returned. This function does no editing of the data.

Parameters

name

The name for the stream can be a path name or a string that was returned from the **stream open** or **popen** commands. If *name* is omitted, the standard input stream is used.

start

For a persistent stream, specifies the byte number in the file where the read begins. *start* should not be specified for other types of streams.

length

The number of bytes returned by the function. If *length* bytes are not available in the stream, the function returns the number of bytes that are available and marks an error condition on the stream. For non-persistent streams, this function either blocks until *length* bytes are available, or returns with fewer bytes, depending on the file type and open flags. If *length* is 0, no characters are read, a null string is returned, and the read position is set based on the value of *start*.

Example

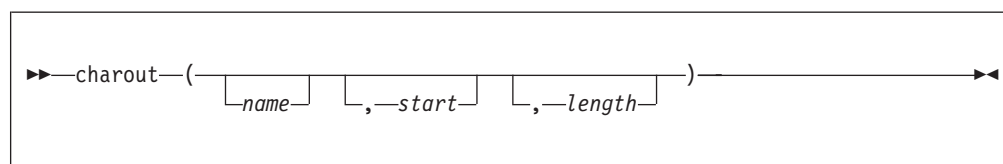
To read the next 256 characters:

```
say charin(file,,256)
```

To set the read location to the sixth 80-byte record:

```
call charin file,5*80+1,0
```

charout()



Function

Returns the number of characters remaining after an attempt to write *string* to the stream specified by *name*. The location for the next write is the current location increased by the number of characters written.

Parameters

name

The name for the stream can be a path name or a string that was returned from the **stream open** or **popen** commands. If *name* is omitted, the standard output stream is used.

string

Data to be written to the stream specified by *name*.

start

For a persistent stream, specifies the byte number in the file where the write begins. *start* should not be specified for other types of streams.

charout()

Usage notes

1. If *string* is omitted, no data is written, 0 is returned, and the write location is set to the value *start*.
2. If *start* is also omitted, the write position is set to the end of file.

Example

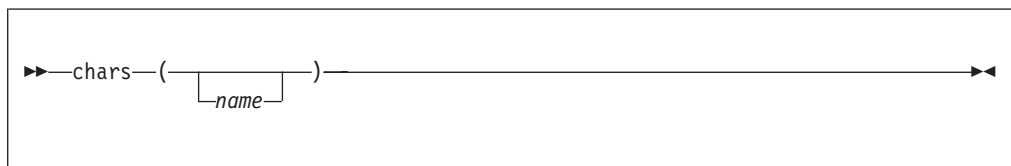
To write the string to **stdout**:

```
call charout , 'hello world' esc_n
```

To set the write position to end:

```
call charout file
```

chars()



Function

Returns the number of characters remaining in the input stream specified by *name*. For persistent streams, this is the number of characters between the current read location and the end of the stream. If the stream was created by the **stream popen** command, **chars()**, while the process is active or bytes remain in the stream, returns either 1 or the number of bytes in the stream. After the process has terminated and the stream is empty, **chars()** returns a value of 0.

Parameters

name

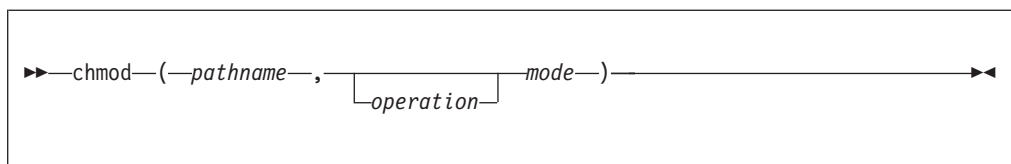
The name for the stream, which can be a path name or a string that was returned from the **stream open** or **popen** commands. If *name* is omitted, the standard input stream is used.

Example

To get the number of bytes in the **stdin** stream:

```
remainder=chars()
```

chmod()



Function

Changes the mode for the specified path name. It returns 0 if the mode for the specified path name is changed; otherwise, it returns the system call error number.

Parameters

pathname

An absolute or relative path name for a file.

operation

Specifies whether mode bits are to be set, added, or deleted:

- = Set the mode bits. If an operation is not specified, = is used.
- + Add the mode bits to what is currently set for the file.
- Remove the mode bits from what is currently set for the file.

mode

A string of octal digits for the new file mode.

Example

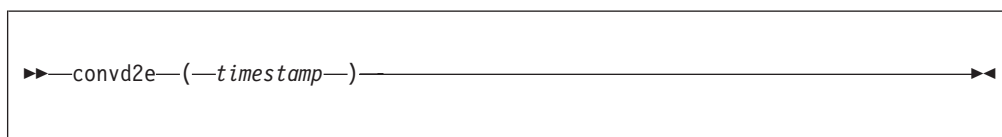
To set permissions for owner and group to read-write:

```
call chmod file,660
```

To add read permission for other:

```
call chmod file,+4
```

convd2e()



Function

Converts *timestamp* to POSIX epoch time, and returns the time in seconds past the POSIX epoch (1/1/1970).

Parameters

timestamp

A 14-character string in the form `mmddyyyyHHMMSS`.

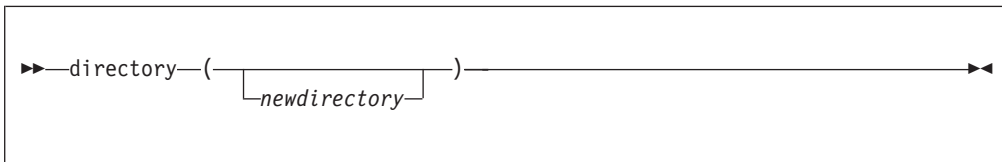
Example

To set POSIX time for 4/21/99 7:15:00 :

```
say convd2e('04211999071500')
```

directory()

directory()



Function

Returns the full path name to the current directory, first changing it to *newdirectory* if the argument is supplied and you have access to that directory.

Parameters

newdirectory

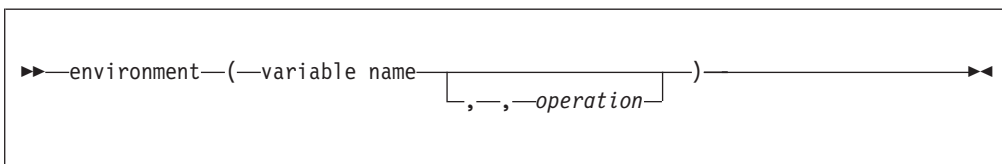
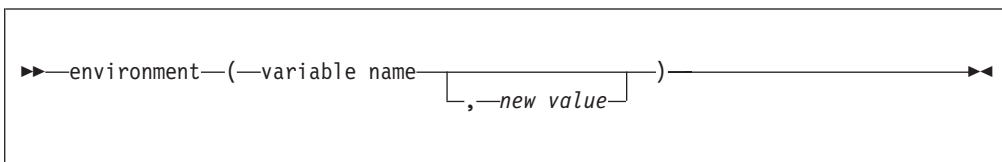
An absolute or relative path name for a directory.

Example

To change the current directory to `/u/ehk`:

call `directory '/u/ehk'`

environment()



Function

Queries and alters environment variables. The stem `__environment.` is not altered through this service. That stem contains the environment variables on entry to the REXX program, and is available for your use. Alterations of the environment are used on subsequent calls to the **stream popen** command and ADDRESS SH.

Parameters

variable name

The name of the environment variable to operate on. If this is the only argument specified, the value of the variable is returned and the variable is not affected.

new value

A string to replace the value of *variable name*. The previous value of the variable is returned.

operation

An optional argument that specifies the operation to be performed. Only the first character is significant. The values can be:

exists Tests the existence of the variable. The function returns 1 if the variable is defined, and 0 if it is not defined.

delete Deletes the variable from the environment, if it exists. The function returns 0 if the variable is successfully deleted, and 1 if the variable is not defined.

Example

To get the value of the PATH environment variable:

```
path=environment('PATH')
```

To reset PATH to the current directory:

```
call environment 'PATH','.'
```

To delete the PATH environment variable:

```
call environment 'PATH',,'d'
```

exists()

▶▶—exists—(*filename*)—▶▶

Function

Returns the full path name for the specified file. If the file does not exist, the function returns a null string.

Parameters**filename**

A string that names a file

Example

To print the full path name for the file myfile:

```
say exists('myfile')
```

getpass()

getpass()

Diagram showing the syntax for the `getpass()` function: `getpass(—prompt—)`. The prompt is enclosed in a box with a double-headed arrow, indicating it is an optional parameter.

Function

Prints *prompt* on the controlling TTY and reads and returns one line of input with terminal echo suppressed.

Parameters

prompt

A string that prints on the controlling TTY

Example

To prompt for a password and read it:

```
psw=getpass('enter password')
```

linein()

Diagram showing the syntax for the `linein()` function: `linein(—name—,start—,count—)`. The parameters `name`, `start`, and `count` are enclosed in boxes with arrows pointing to their respective positions in the function signature.

Function

Returns one line or no lines from the stream specified in *name*, and sets the location for the next read to the beginning of the next line. The data is assumed to be text. The newline character is the line delimiter and is not returned. A null string is returned if no line is returned; this appears the same as a null line in the file. Use **chars()** or **lines()** to determine if you are at end of file. Use **stream()** to determine if there is an error condition on the stream.

Parameters

name

The name for the stream can be a path name or a string that was returned from the **stream open** or **popen** commands. If *name* is omitted, the standard input stream is used.

start

For a persistent stream only, this argument can have the value 1, to begin reading at the beginning of the stream. No other value for *start* is supported. Do not specify *start* for other types of streams.

count

Specify 0 or 1 for the number of lines to be returned by the function. If one line is not available in the stream, the function returns a null string and marks an error condition on the stream. For non-persistent streams, this function either blocks until a line is available, or returns as though it is end of file, depending on whether any data remains in the file, on the file open flags, and on the type of file. If count is 0, no lines are read, a null string is returned, and, if *start* is 1, the read position is set to the beginning of the stream.

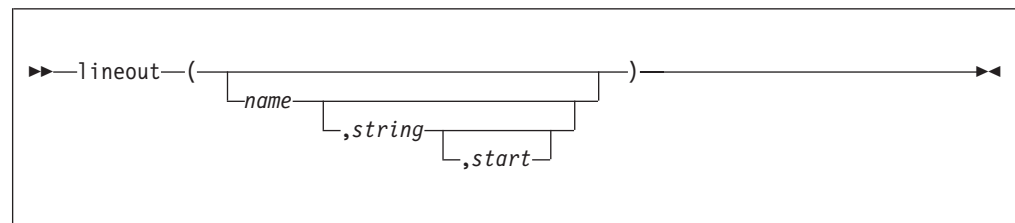
Example

To read the next line:

```
line=linein(file)
```

To read the first line in the file:

```
line=linein(file,1)
```

lineout()**Function**

Returns 1 line or 0 lines that are remaining to be written after an attempt to write *string* to the stream specified by *name*. A newline character is written following *string*. If an error occurs on the write, some data may be written to the stream, and the function returns the value 1.

Parameters**name**

The name for the stream can be a path name or a string that was returned from the **stream open** or **popen** commands. If *name* is omitted, the standard output stream is used.

string

Data that is to be written to the stream specified by *name*. If *string* is omitted, no data is written, 0 is returned, and the write position is set based on the value of *start*. If both *string* and *start* are omitted, the write position is set to the beginning of the file.

start

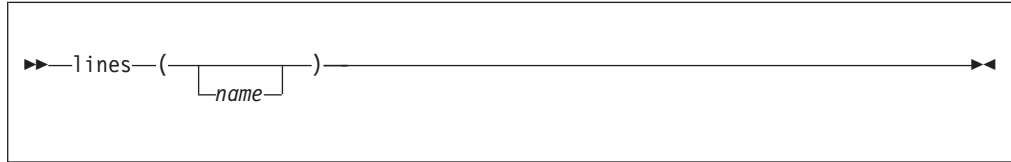
For a persistent stream, this argument can have the value 1, to begin writing at the beginning of the stream. No other value for *start* is supported. Do not specify *start* for other types of streams.

lineout()

Example

to write the line to **stdout**:
call lineout ,'hello world'

lines()



Function

Returns 1 if data remains in the stream; otherwise it returns 0. Programs should check for a value of 0 or nonzero.

Parameters

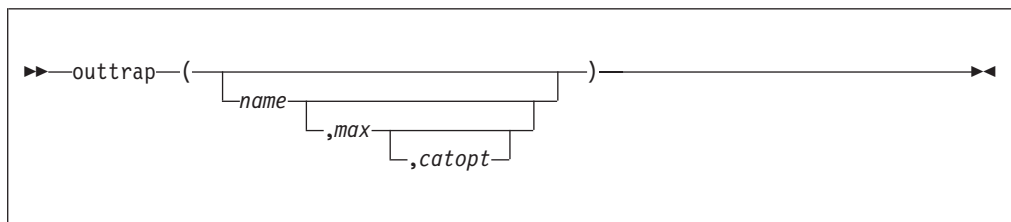
name

The name for the stream can be a path name or a string that was returned from the **stream open** or **popen** commands. If *name* is omitted, the standard input stream is used.

Example

To set *more* to nonzero, if **stdin** has data:
`more=lines()`

outtrap()



Function

Enables or disables the trapping of output from commands run using ADDRESS TSO, and returns the name of the variable in which trapped output is stored. If trapping is off, the word OFF is returned. Note that **outtrap** does not trap output for ADDRESS SH or any other command environment besides TSO. To trap shell command output, see “`bpxwunix()`” on page 174.

Parameters

name

The name of a stem, a variable prefix used to contain command output, or the string OFF to turn off trapping.

max

The maximum number of lines to trap. If *max* is not specified, or if it is specified as * or a blank, the number of lines is set to 999 999 999.

catopt

Specify one of these:

concat

Each command output trapping begins following the previous command output.

noconcat

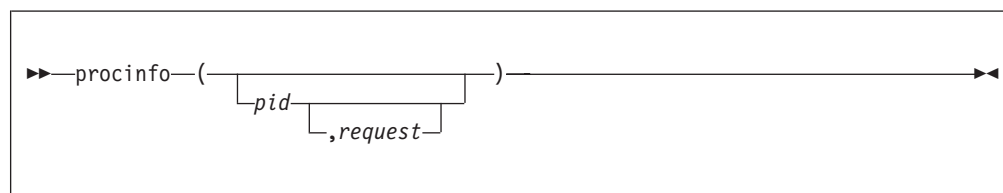
Output from each command is trapped starting with the variable concatenated with 1. Unused variables do not have their values altered.

For additional information about using the TSO command environment, see “The TSO command environment” on page 5.

Example

To trap TSO command output, use the stem OUT, rewriting the stem on each command:

```
call outtrap 'out.', 'NOCONCAT'
```

procinfo()**Function**

Retrieves information about one or more processes.

Parameters**pid**

The process ID number of the process for which information is to be returned. If *pid* is not specified, basic information is retrieved for all processes you have authorization to view.

request

Specify one of these:

file Retrieves file information for the specified process.

filepath

Retrieves file information for the specified process and returns any portion of the path name that is available.

process

Retrieves information about the specified process. This is the default.

thread Retrieves information for all threads in the specified process.

Results

Information is returned in simple and compound variables. Each variable name has a prefix that defaults to `bpxw_`. This prefix can be changed using the `rexxopt()` function. A stem can be used as the prefix, in which case the tails are the simple suffixes set by this function.

If *pid* is not specified, the following suffixes are set:

Suffix	Description
PID.n	Each variable contains one process ID number. <i>PID.0</i> is the number of PIDs returned. <i>PID.1</i> , <i>PID.2</i> , ... are the PID numbers.
THREADS.n	The number of threads for the process in the corresponding <i>PID.n</i>
ASID.n	The address space ID for the process in the corresponding <i>PID.n</i>
JOBNAME.n	The job name for the process in the corresponding <i>PID.n</i>
LOGNAME.n	The login name (user ID) for the process in the corresponding <i>PID.n</i>

If process information is requested for a process, the following suffixes are set:

Suffix	Description
STATE	Contains 0 or more of the following strings: <ul style="list-style-type: none"> • MULPROCESS • SWAP • TRACE • STOPPED • INCOMPLETE • ZOMBIE
PID	Process ID number
PPID	Parent process ID number
PGRID	Process group ID number
SID	Session ID number
FPGID	Foreground process group number
EUID	Effective user ID
RUID	Real user ID
SUID	Saved set user ID
EGID	Effective group ID
RGID	Real group ID
SGID	Saved set group ID
SIZE	Region size
SLOWPATH	Number of slow path syscalls
USERTIME	Time spent in user code
SYSTIME	Time spent in system code
STARTTIME	Time when the process started (dub time)

Suffix	Description
OETHREADS	Number of z/OS UNIX threads
PTCREATE	Number of threads created using pthread_create
THREADS	Number of threads in the process
ASID	Address space ID of the process
JOBNAME	Job name for the process
LOGNAME	Login name (user ID) for the process
CONTTY	Path name for the controlling TTY
CMDPATH	Path name for the command that started the process
CMDLINE	Command line that started the process

If file information is requested for a process, the following suffixes are set:

Suffix	Description
DEVNO	Device number
INODE	Inode number
NODES	Count of the total number of file nodes returned
OPENF	Open flags
PATH	If request type FILEPATH is used, then PATH is set to any portion of the path name that is available.
PATHTRUNC	If request type FILEPATH is used, then PATHTRUNC is set to either 0 or 1. <ul style="list-style-type: none"> • 0, if the path name is not truncated. • 1, if the path name might be truncated. The path name may be truncated on the left side, right side, or both.
TYPE	Type number of the file
TYPECD	Type code of the file: <ul style="list-style-type: none"> • rd: root directory • cd: current directory • fd: file descriptor • vd: vnode descriptor

If thread information is requested for a process, the following suffixes are set:

Suffix	Description
THREAD_ID	Thread ID
SYSCALL	The current syscall, if in kernel
TCB	TCB address
RUNTIME	Time running in milliseconds
WAITTIME	Time waiting in milliseconds
SEMNUM	Semaphore number if on a semop (ptrunwait.x='D')

procinfo()

Suffix	Description
SEMVALUE	Semaphore value if on a semop (ptrunwait.x='D')
LATCHPID	Latch that the process waited for
SIGPENDMASK	Signal pending mask
LOGNAME	Login name
LASTSYSCALL.n	Last 5 syscalls
PTCREATED	Contains J if this was pthread-created
PTRUNWAIT	Contains one of the following letters, to indicate the current run or wait state of the thread: A msgrecv wait B msgsend wait C communication wait D Semaphore wait F File System Wait G MVS in Pause K Other kernel wait P PTwaiting R Running or non-kernel wait S Sleep W Waiting for child X Fork new process Y MVS wait
PTTYPE	Contains one of the following letters to indicate thread type: N Medium-wait thread O Asynchronous thread U Initial process thread Z Process terminated and parent has not completed wait
PTDETACH	Contains V if the thread is detached
PTTRACE	Contains E if the thread is in quiesce freeze
PTTAG	Contains the ptagdata string if it exists
THREADS	Contains the total number of threads

Usage notes

1. This function uses the `__getthent` callable service. For additional information, see `__getthent` (BPX1GTH, BPX4GTH) — Get thread data in *z/OS UNIX System Services Programming: Assembler Callable Services Reference*.
2. The **procinfo()** function returns as its value the prefix used to create variable names. If there is an error returned by the `__getthent` service, the function returns a null string, and sets the variables `ERRNO` and `ERRNOJR` to the hex values of the error codes returned by the `__getthent` service.

Example

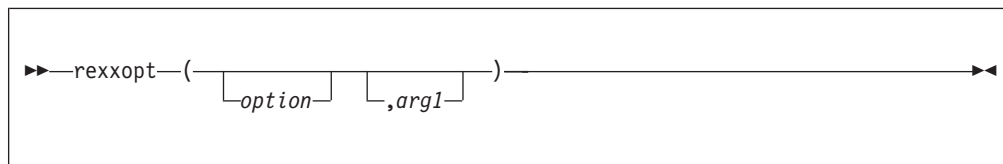
To show the command line for each process:

```
call procinfo
do i=1 to bpxw_pid.0
  if procinfo(bpxw_pid.1)<>' ' then /* ignore processes that ended */
    say bpxw_cmdline
  end
end
```

To show the current directory for each process:

```
call procinfo
do i=1 to bpxw_pid.0
  if procinfo(bpxw_pid.i,'file')<>' ' then /* ignore processes that ended */
    do j=1 to bpxw_nodes
      if bpxw_typecd.j='cd' then
        do
          address syscall 'getmntent m.' bpxw_devno.j
          strm=stream('find' m.mnte_path.1 '-xdev -inum', bpxw_inode.j,,
                    'c', 'popen read')
          say right(bpxw_pid.i,12) linein(strm)
          call stream strm,'c','pclose'
        end
      end
    end
  end
end
```

rexxopt()



Function

Sets, resets, or queries z/OS UNIX REXX options.

Parameters

option

Specify one of these:

immed

Associates the immediate command interrupt handler to a signal number.

noimmed

Restores the default action for a signal. *arg1* is the signal number.

version

Returns a string showing last compile time for the function package.

varpref

Sets the variable name prefix used by some of the REXX functions. *arg1* is the new prefix. If *arg1* is not specified, the current value is not changed. If it is specified but null, it is defaulted to `bpxw_`. This returns a string that is the current setting.

rexopt()

notsoin

Disables any input to the TSO command processor that was started using ADDRESS TSO, including the stack. This returns the string TSO input disabled.

tsoin

Sets back to its default setting the input mode to the TSO command processor that was started using ADDRESS TSO. This returns the string TSO input enabled.

arg1

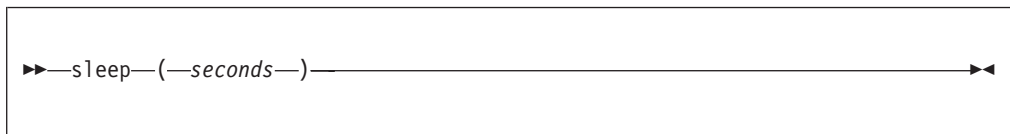
A signal number.

Example

To make the interrupt signal prompt for a command:

```
call rexopt 'immed',sigint
```

sleep()



Function

Places the process in a signals-enabled wait, and returns after the wait expires. If a signal interrupts the wait, the function returns the number of seconds remaining for the wait, otherwise it returns 0.

Parameters

seconds

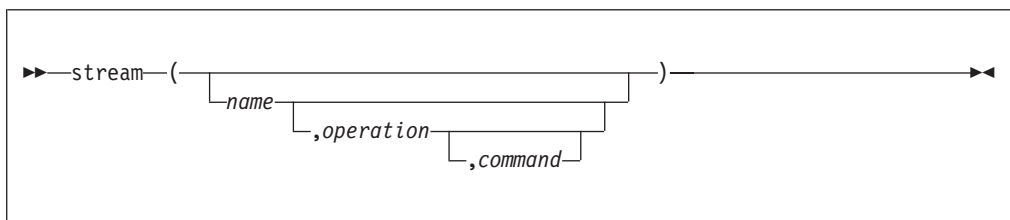
A positive whole number for the number of seconds to sleep

Example

To wait 5 seconds:

```
call sleep 5
```

stream()



Function

Returns the state of the stream or the result of the command.

Parameters

name

The stream name for the operation. This argument is case-sensitive.

operation

The operation that is to be performed. Only the first character is significant. The operations can be:

- S** Returns the current state of the stream. This is the default operation. The following values can be returned:
- ready** The stream is available for normal input or output operations.
 - error** The stream encountered an error on an input or output operation. After this call, the error condition is reset, and normal input or output operations can be attempted once again.
 - unknown** The specified stream is not open.
- D** This operation is almost the same as S, with one exception. When ERROR is returned, it is followed by text describing the error. This text usually contains the error number and reason code if the failure is due to a system call failure.
- C** Execute a command for the specified stream. The command is specified as the third argument.

command

The stream command that is to be executed. This argument is valid only when the operation is C. The following commands are supported:

clearfile

Truncates a file to zero bytes for a stream that is opened for write. **clearfile** should only be used on a persistent stream.

This example empties the file:

```
call stream name,'c','clearfile'
```

- close** Closes a stream. On success, the function returns the string ready. If the stream is not known, it returns the string UNKNOWN.

This example closes the stream:

```
call stream file,'c','close'
```

infileno

Returns the file descriptor number of the input side of a stream. The file descriptor can be used on lower-level system calls, using, for example, ADDRESS SYSCALL. This example gets the file descriptor for the read side of the stream:

```
fd=stream(name,'c','infileno')
```

nosignal

Disables Halt Interruption for stream errors. See 193. This example disables the halt signal for the standard input/output stream:

```
call stream , 'c', 'nosignal'
```

stream

open <*open-type*>

Opens a stream. *open-type* specifies either read or write. If <*open-type*> is not specified, read is assumed. The function returns a string that is the name to be used for the stream on subsequent I/O functions. This string is the only name by which this stream will be known. Note that streams do not have to be explicitly opened. A path name that is used as a stream name on the other I/O functions will cause the stream to be opened, and the name of the stream will be that path name. Optional arguments may be specified with write, optionally followed by octal permission bits. Write always opens the file with O_CREAT, creating the file if it does not exist. The additional arguments are:

replace

Opens the file with O_TRUNC, setting the file size to 0.

append

Opens the file with O_APPEND, causing all writes to be appended to the file.

This example opens a stream for the file mydata.txt:

```
file=stream('mydata.txt','c','open write')
```

This example opens a stream for the file mydata.txt, but replaces the file if it exists:

```
file=stream('mydata.txt','c','open write replace')
```

outfileno

Returns the file descriptor number of the output side of a stream. The file descriptor can be used on lower-level system calls, using, for example, ADDRESS SYSCALL. This example gets the file descriptor for the write side of the stream:

```
fd=stream(name,'c','outfileno')
```

pclose Closes a process stream. On success, the function returns the completion code for the process run via the **popen** command.

pid Returns the process ID number for the shell process opened with popen. This example gets the PID for a stream opened with **popen**:

```
pid=stream(name,'c','pid')
```

popen <*open-type*>

Opens a pipe to a shell command that is specified by the stream name.<*open-type*> must specify read or write. If read is specified, the input functions can be used to read the standard output from the command. If write is specified, the output functions can be used to pipe data to the standard input of the shell command. In either case, the shell command inherits the standard error file for the calling process.

The command that is run is always **/bin/sh -c** followed by the specified shell command. This means that the completion code is the one returned by the shell. It usually returns the command's completion code. You can obtain the completion code by using the **pclose** command.

The function returns a string that is the name to be used for the stream on subsequent I/O functions. This string is the only name by which this stream will be known.

This example opens a pipe stream to the output from the shell command `ls | wc`:

```
file=stream('ls | wc','c','popen read')
```

query <attribute>

Queries a stream attribute and returns the result:

exists Returns the full path name of the stream name. This is equivalent to the **exists()** function, but is more portable. This command does not cause a stream to be opened.

size Returns the size of the stream. This is equivalent to the **size stream** command, but is more portable.

This example prints the path name of the stream:

```
say stream('myfile','c','query exists')
```

readpos [*location*]

Returns the position in the file where the next read will begin. If *location* is specified, the position is also set to the byte specified by *location*. *location* is specified as a number optionally preceded by one of the following characters:

- = An absolute byte location. This is the default.
- < An offset from the end of the stream.
- + An offset forward from the current location.
- An offset backward from the current location.

This example gets the read location in the file, and then sets the read location to the sixth 80-byte record:

```
pos=stream(name,'c','readpos') /* get read location */
call stream name,'c','readpos' 5*80+1 /* set read location*/
```

signal Enables Halt Interruption for stream errors. Note that the NOTREADY REXX signal is not supported. This example enables the halt signal for the standard input/output stream:

```
call stream ,'c','signal'
```

size Returns the size of the file associated with the stream. This example prints the size of the file:

```
say stream(name,'c','size')
```

writepos <*location*>

Returns the position in the file where the next write will begin. If *location* is specified, the position is also set to the byte specified by *location*. *location* is specified as a number optionally preceded by one of the following characters:

- = An absolute byte location. This is the default.
- < An offset from the end of the stream.
- + An offset forward from the current location.
- An offset backward from the current location.

This example sets the position to the end of the file:

```
call stream name,'c','writepos <0'
```

This example sets the position to the start of the file:

```
call stream name,'c','writepos 1'
```

submit()

submit()

```
▶▶—submit—(—stem.—)—————▶▶
```

Function

Submits a job to the primary subsystem (JES), returning the job ID of the submitted job.

Parameters

stem.

The stem compound variable contains the number of lines in *stem.0*, and each variable from *stem.1*, *stem.2*, ... contains a line for the job that is being submitted.

Example

This example reads the file into the stem, sets the number of lines, and submits the job:

```
do i=1 by 1 while lines(fn)>0
  fn.i=linein(fn)
end
fn.0=i-1
say submit('fn.')
```

syscalls()

```
▶▶—syscalls—(—control.—)—————▶▶
```

Function

Establishes the SYSCALL environment or ends it; or establishes or deletes the signal interface routine (SIR).

Parameters

ON Establishes the SYSCALL environment. It sets up the REXX predefined variables and blocks all signals.

OFF

Ends the connection between the current task and z/OS UNIX. This action may undub the process. It is usually not necessary to make a **syscalls (OFF)** call.

SIGON

Establishes the signal interface routine.

SIGOFF

Deletes the signal interface routine, and resets the signal process mask to block all signals that can be blocked.

Usage notes

1. This function can be used outside of the z/OS UNIX REXX environment (for example, in TSO/E). When **syscalls** is used in such an environment, **stdin**, **stdout**, **stderr**, and environment variables are not inherited from the current process environment.
2. For more usage information, see “The SYSCALL environment” on page 2.

Example

See “The SYSCALL environment” on page 2.

syscalls()

Chapter 6. BPXWDYN: a text interface to dynamic allocation and dynamic output

BPXWDYN is a text interface to a subset of the SVC 99 (dynamic allocation) and SVC 109 (dynamic output) services. BPXWDYN supports data set allocation, unallocation, concatenation, the retrieval of certain allocation information, and the addition and deletion of output descriptors. BPXWDYN is designed to be called from REXX, but it can be called from several other programming languages, including Assembler, C, and PL/I.

This interface makes dynamic allocation and dynamic output services easily accessible to programs running outside of a TSO environment; however, it also functions in a TSO environment.

The syntax for allocation is quite similar to that of TSO for the TSO ALLOCATE and FREE commands. It should be possible to provide parameters to BPXWDYN that would be acceptable as a TSO ALLOCATE or FREE command. However, there are keys supported by TSO ALLOCATE that are not currently supported by BPXWDYN. There are also some keys that can be used with BPXWDYN which are not compatible with TSO.

The syntax for accessing dynamic output facilities is similar to that of the TSO OUTDES command, but the name of the output descriptor is identified differently. You associate an output descriptor with a SYSOUT allocation by using the OUTDES key on the SYSOUT allocation request, or by creating a default output descriptor.

Calling conventions

BPXWDYN must be called in 31-bit mode in an environment that permits dynamic allocation and dynamic output requests. To call BPXWDYN from REXX or any other program, three parameter list forms can be used:

- REXX external function parameter list
- Conventional MVS variable-length parameter string
- Null-terminated parameter string

REXX external function parameter list

The external function parameter list allows a REXX program to call the BPXWDYN program as a function or subroutine. The BPXWDYN program must be called with a single string parameter. For example:

```
if BPXWDYN("alloc dd(sysin) da(my.dataset) shr")<>0 then
  call allocfailed
```

Conventional MVS variable-length parameter string

The conventional MVS variable-length parameter string is the same parameter list as the one generated by ADDRESS LINKMVS with one parameter, and by JCL with EXEC PGM=,PARM=. Any program can easily use this parameter list form. Note that this is a single-item variable-length parameter list. The high bit is on in the parameter address word and *length* is a halfword that has a value less than 16384.

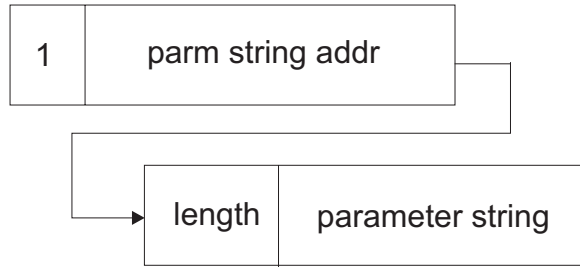


Figure 1. Example of parameter list

```

PL/I usage might include the following statements:
DCL PLIRETV BUILTIN;
DCL BPXWDYN EXTERNAL ENTRY OPTIONS(ASM INTER RETCODE);
DCL ALLOC_STR CHAR(100) VAR
    INIT('ALLOC FI(SYSIN) DA(MY.DATASET) SHR');
FETCH BPXWDYN;
CALL BPXWDYN(ALLOC_STR);

```

Conventional MVS parameter list

REXX variables that are requested to be set are set in both reentrant and non-reentrant REXX environments. Variables are not set when called from non-REXX environments. In order to obtain returned information from a non-REXX environment, BPXWDYN must be called with multiple parameters in the following format.

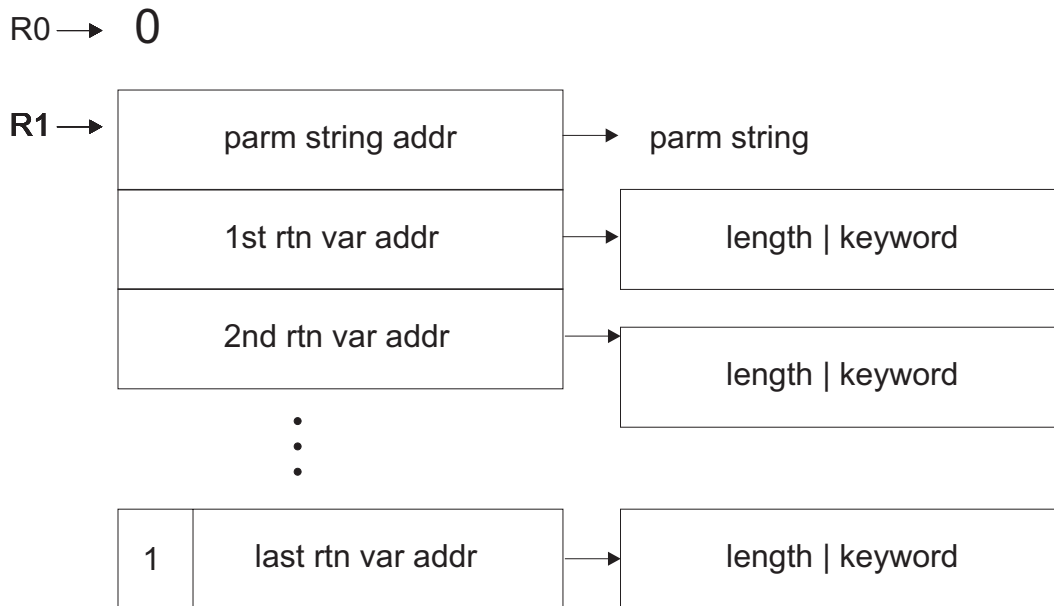


Figure 2. Example of a conventional MVS parameter list

The parm string has the format of either halfword length followed by the parameter string or a null-terminated parameter string.

The returned variable parameters are a halfword length of the return area followed by the keyword for the value to be returned. The keyword must end with a blank or null character. Examples of keywords with return values are RTDDN, RTDSN, INRTPATH, and MSG.

Upon return, the return areas will contain a null-terminated string for the field requested if it was available and if it fits into the area. A null string is returned if the data does not fit into the parameter area. The length is set to the length of the data returned excluding the terminating null.

If a keyword value is not returned, a null string is returned.

Guideline: Because a null-terminated string is returned, in order to ensure that the data is returned, the parameter area must be large enough for the expected returned data plus the null character.

MSG is the only keyword that takes a stem. Special processing takes place for stem data. The argument that contains MSG upon return contains the number of message lines issued by SVC99 or a null string if no messages are issued. The message lines are returned in arguments passed as MSG.*n* where *n* is the line number.

Example: The following C example allocates a temporary data set and then retrieves and prints the ddname, data set name and up to four lines of messages if issued.

```
typedef int EXTF();
#pragma linkage(EXTF,OS)
#include <stdlib.h>
#include <stdio.h>
void main() {
EXTF *bpxwdyn=(EXTF *)fetch("BPXWDYN ");
int i,j,rc;
char *alloc="alloc new delete";
typedef struct s_rtarg {
short len;
char str[260];
} RTARG;
RTARG ddname = {9,"rtddn"};
RTARG dsname = {45,"rtdsn"};
RTARG msg = {3,"msg"};
RTARG m[4] = {258,"msg.1",258,"msg.2",258,"msg.3",258,"msg.4"};
rc=bpxwdyn(alloc,&dsname,&ddname,&msg,&m[0],&m[1],&m[2],&m[3],
if (rc!=0) printf("bpxwdyn rc=%X\n",rc);
if (*ddname.str) printf("ddname=%s\n",ddname.str);
if (*dsname.str) printf("dsname=%s\n",dsname.str);
for (i=0,j=atoi(msg.str);i<j && i<4;i++)
printf("%s\n",m[i].
}
```

Null-terminated parameter string

This parameter list is used most easily from C, passing a string to BPXWDYN defined with a #pragma for OS linkage, or extern "OS" for C++. This string should not contain any control characters. Note that the high bit is on in the parameter address word.

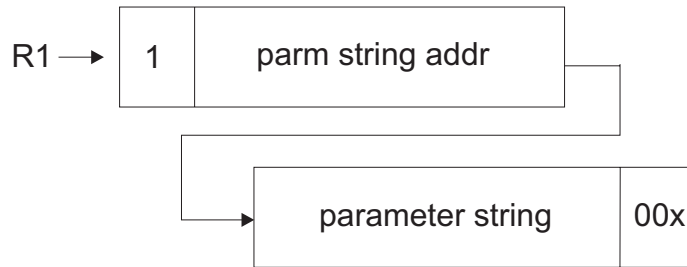


Figure 3. Example of a null-terminated parameter string

C usage might include the following statements:

```
typedef int EXTf();
#pragma linkage(EXTf,0S)
EXTf *bpxwdyn;
int rc;
bpxwdyn=(EXTf *)fetch("BPXWDYN");
rc=bpxwdyn("alloc fi(sysin) da(my.dataset) shr");
```

Request types

The request is specified in a string parameter. The request type must be the first keyword in the parameter string. If a valid request type is not the first keyword, or if a request type is not specified, the request type defaults to ALLOC.

The following request types are supported:

ALLOC

Dynamic allocation of a data set

CONCAT

Dynamic concatenation of a ddname list

DECONCAT

Deconcatenation of a ddname list

FREE Dynamic unallocation of a ddname, or freeing of an output descriptor

INFO Retrieval of allocation information

OUTDES

Creation of an output descriptor

Keywords

Any BPXWDYN keyword that accepts a variable name for returned information allows for a file descriptor number. For example, `inrtpath(1)` writes the path name, if available, as a line to descriptor 1. Additionally, when BPXWDYN is used from languages other than REXX and variables are not available, information can be returned by means of the parameter list as described in "Conventional MVS parameter list" on page 198.

The ordering of keywords on the request is arbitrary. You cannot specify the same keys multiple times for one request.

BPXWDYN does not perform consistency checking on keys. If some keys are not valid when combined together on a single request, dynamic allocation or dynamic output fails the request.

Some keywords accept arguments within the following guidelines:

- Keyword arguments must be specified within parentheses. Spaces are not permitted between the key and the opening parenthesis, or anywhere within the additional argument string through the closing parenthesis, unless the argument string is quoted. Spaces are permitted between key specifications. Where multiple arguments are permitted, the arguments must be separated by commas.
- An argument can be enclosed within single (') or double (") quotes. Two adjacent quotes cannot be used to represent a single quote within a quoted string.
- Arguments that are not quoted are treated as uppercase. Arguments that are quoted are only treated as mixed case if it makes sense for the keyword.
- Arguments that accept REXX variable names are accepted when BPXWDYN is called from any environment, but are only effective when called from a REXX environment.

BPXWDYN return codes

When BPXWDYN is called as a REXX function or subroutine, the return code can be accessed in RESULT or as the value of the function. When BPXWDYN is called as a program, the return code is available in R15.

BPXWDYN returns the following codes:

0 Success.

20 Invalid parameter list. See “Calling conventions” on page 197 for parameter list formats.

-21 to -9999
Key error. See “Key errors” for more information.

-100nn
Message processing error. IEFDB476 returned code *nn*.

>0, or -1610612737 to -2147483648
Dynamic allocation or dynamic output error codes. See “Error codes for dynamic allocation” and “Error codes for dynamic output” on page 202 for more information.

Key errors

The low-order two digits are the number of the key that failed the parse, offset by 20. The first key has the value 21, resulting in a return code of -21.

The high-order two digits are an internal diagnostic code that indicates where in the code the parse failed. If the high digits are 0, the key itself was probably not recognized. Other values usually indicate a problem with the argument for the failing keyword. Likely causes for failure are:

- Blanks in arguments that are not quoted
- Misspelling of keywords

Error codes for dynamic allocation

The return code contains the dynamic allocation error reason code in the high two bytes, and the information reason code in the low two bytes.

You can use the high four hexadecimal digits to look up the error code in the dynamic allocation error reason codes table found in the *z/OS MVS Programming: Authorized Assembler Services Guide*.

Note: For the returned decimal codes (>0, or -1610612737 to -2147483648), convert them to hexadecimal codes before determining the value for the high and low two bytes.

Error codes for dynamic output

Dynamic output errors do not produce messages. The return code contains:

- The dynamic output return code (S99ERROR) in the high two bytes
- The information code (S99INFO) in the low two bytes

You can use the high four hexadecimal digits to look up the error code in the dynamic output return codes table found in the *z/OS MVS Programming: Authorized Assembler Services Guide*. You can use the low four digits to look up the information code.

There is no indication of the key that is in error.

Some of the most common reason codes are as follows:

300-30C, 312, 380

The arguments may be specified incorrectly.

401 The output descriptor already exists.

402 The output descriptor does not exist.

403 Output descriptors created by JCL cannot be deleted by dynamic output.

Message processing

Dynamic allocation provides message text for failed allocation requests. It is usually easier to use this message text rather than decode allocation return codes.

BPXWDYN can return these messages to a REXX program. By default, the messages are returned in the S99MSG. stem. S99MSG.0 contains the number of messages and S99MSG.1 through S99MSG.n contain the message text.

To change the name of the stem, use the MSG key. A stem is not required. Digits are simply appended to the variable specified by the MSG key.

You can also use the MSG key to request allocation to issue the messages to your job log (write to programmer using WTO).

Under z/OS UNIX, you can have messages written to opened files by providing the file descriptor number as the argument to the MSG key. For example, MSG(2) writes messages to the STDERR file.

Requesting dynamic allocation

To request dynamic allocation, specify **alloc** at the beginning of the parameter string.

Note: Allocation verifies the validity of the path name. However, because there is no ENQ or locking of the path name, it is possible to modify a path name component, even in an asynchronous process. Modifying the pathname might cause errors in OPEN or unexpected results with no errors reported.

BPXWDYN supports single data set allocation. Many of the common allocation keys can be used. For detailed information about those common allocation keys, see *z/OS TSO/E Command Reference*. Some additional keys are supported to access additional functions.

BPXWDYN supports the following keys:

Table 3. Common keys used for dynamic allocation

Keys used for dynamic allocation	Action
DA(<i>data set name</i> [(<i>member name</i>)]) DSN(<i>data set name</i> [(<i>member name</i>)])	Specifies the data set name to allocate. The name must be fully qualified and can include a member name. Quotes can be used for TSO compatibility.
DUMMY	Allocates a dummy data set.
FI(<i>name</i>) DD(<i>name</i>)	Specifies the ddname to allocate.
FILEDATA(TEXT BINARY)	Tells the sequential access method services whether the data is to be treated as text or binary.
OLD SHR MOD NEW SYSOUT[(<i>class</i>)]	Specifies the data set status. SYSOUT specifies that the data set is to be a system output data set and optionally defines the output class.
VOL(<i>volser</i> [, <i>volser</i> ...])	Specifies the serial numbers for eligible direct access volumes where the data set is to reside.
DATACLAS(<i>data class</i>)	With SMS, specifies the data class for the data set.
MGMTCLAS(<i>management class</i>)	With SMS, specifies the management class for the data set.
STORCLAS(<i>storage class</i>)	With SMS, specifies the storage class for the data set.
SPACE(<i>primary</i> [, <i>secondary</i>])	Specifies primary and optionally secondary space allocations.
BLOCK(<i>length</i>)	Specifies block allocation with an average block or record size of <i>length</i> .
TRACKS	Specifies the unit of space in tracks.
CYL	Specifies the unit of space in cylinders.
BLKSIZE(<i>block size</i>)	Specifies the block size.
DIR(<i>directory blocks</i>)	Specifies the number of directory blocks.
DEST((<i>destination</i> <i>destination</i> [. <i>user</i>]))	Specifies the output destination or the output destination and node.
REUSE	Causes the named DD to be freed before the function is performed.
HOLD	Specifies that the output data set is to be held until released by the user or operator.
UNIT(<i>unit name</i>)	Specifies unit name, device type, or unit address.
MAXVOL(<i>num vols</i>)	Number of volumes for a multi-volume data set.
KEEP DELETE CATALOG UNCATALOG	Specifies the data set disposition after it is freed.
BUFNO(<i>number</i>)	Sets the number of buffers. This number should be in the range 1–255. Numbers outside that range give unpredictable results.
LRECL(<i>record length</i>)	Specifies the logical record length.
RECFM(<i>format</i> [, <i>format</i> ...])	Specifies the record format. The valid values are A, B, D, F, M, S, T, U, and V. Several of these can be used in combination.

Table 3. Common keys used for dynamic allocation (continued)

Keys used for dynamic allocation	Action
DSORG(PS PO DA)	Specifies the data set organization.
COPIES(<i>number of copies</i>)	Specifies the number of copies to print.
FORMS(<i>forms name</i>)	Specifies the print form.
LIKE(<i>model data set name</i>)	Copies attributes for the allocation from the model data set.
OUTDES(<i>output descriptor name</i>)	Specifies the output descriptor name.
SPIN(UNALLOC)	Spins off a sysout data set at unallocation.
DSNTYPE(LIBRARY PDS HFS EXTREQ EXTPREF BASIC LARGE)	Specifies the data set type.
WRITER(<i>external writer name</i>)	Names the external writer.
PATH(<i>pathname</i>)	Specifies the path name of a file to allocate.
PATHDISP(KEEP DELETE[,KEEP DELETE])	Specifies the file disposition for normal and abnormal termination of the job step.
PATHMODE(<i>path mode list</i>)	Set mode bits for a new allocation. This list is a list of keywords separated with commas. The supported keywords are: SIRUSR SIWUSR SIXUSR SIRWXU SIRGRP SIWGRP SIXGRP SIRWXG SIROTH SIWOTH SIXOTH SIRWXO SISUID SISGID SISVTX
PATHOPTS(<i>path options list</i>)	Sets options for path name allocation. The options list is a list of keywords separated with commas. The supported keywords are: ORDWR OEXCL OSYNC OTRUNC OCREAT OWRONLY ORDONLY OAPPEND ONOCITY ONONBLOCK
RECORG(LS)	Creates a VSAM linear data set.
SEQUENCE(<i>sequence number</i>)	Specifies the relative position (number) of a data set on a tape volume.

Table 3. Common keys used for dynamic allocation (continued)

Keys used for dynamic allocation	Action
LABEL(<i>type</i>)	Specifies the type of tape label processing to be done, as follows: NL The volume has no label. SL The volume has an IBM® standard label. NSL The volume has a non-standard label. SUL The volume has both an IBM standard label and a user label. BLP Bypass label processing for the volume. LTM The system is to check for and bypass a leading tape mark on a DOS unlabeled tape. AL The volume has an American National Standard label. AUL The volume has both an American National Standard label and a user label.
RETPD(<i>number of days</i>)	Specifies the data set retention period, in days.
TRTCH(<i>technique</i>)	Specifies the tape recording technique, as follows: NONCOMP Non-compaction mode COMP Compaction mode C Data conversion E Even parity ET Even parity and BCD/EBCDIC translation T BCD/EBCDIC translation
SUBSYS(<i>subsystem name</i> [, <i>subsys parm...</i>])	Specifies that the data set is a subsystem data set, and also specifies the name of the subsystem and any parameters necessary for the subsystem to process the data set. Case and special characters are preserved for quoted parameters.

The following additional keys are unique to BPXWDYN.

Table 4. Additional keys used for dynamic allocation

Additional keys used for dynamic allocation	Action
ACUCB	Uses the nocapture option (sets S99ACUCB)
AVGREC(U K M)	Specifies the allocation unit to be used when the data set is allocated u single record units k thousand record units m million record units
CKEXIST	Adds the DALRTORG TU in order to forces sv99 to read the data set control block (DSCB). This action is similar to that of TSO allocation. CKEXIST prevents an allocation request from succeeding when the DSN does not exist and a volume was specified.
CLOSE	Frees the allocation when the file is closed.

Table 4. Additional keys used for dynamic allocation (continued)

Additional keys used for dynamic allocation	Action
DIAG (stem <i>fdnum</i> STDOUT)	Specifies that diagnostic information is to be displayed. The information includes a compile time stamp, the input string, key parsing information for subsequent keys, and the list of generated text units. Specify STDOUT to send output information to the standard REXX output stream.
<ul style="list-style-type: none"> • DUMP(<BEFORE AFTER NOSVC>, • <DD:ddname DA:datasetname> 	Specifies that a dump be taken using IEATDUMP either before or after the SVC99 or skip the allocation. If a ddname is used, it must already have been allocated. If a data set name is used, a data set name or pattern name must be specified that conforms to the requirements of IEATDUMP.
EATTR (NO OPT)	Specifies whether extended attributes are allowed.
EXPDL(<i>yyyymmdd</i>)	Specifies the expiration date of the data set.
EXPDT(<i>yyddd</i>)	Specifies the expiration date of the data set.
GDGNT	Sets the S99GDGNT flag in the S99FLAG1 field. For information about this flag, see <i>z/OS MVS Programming: Authorized Assembler Services Guide</i> .
MOUNT	Resets the S99NOMNT flag, allowing volumes to be mounted.
MSG (WTP default.S99MSG. <i>stemname</i> <i>fdnum</i>)	Directs allocation messages to your job log (WTP), a REXX stem, or a file identified by a file descriptor number. If this key is not specified, messages are returned in the S99MSG. stem, if possible. If BPXWDYN was not called from a REXX environment, the messages will be lost.
NORECALL	Fails the allocation request if the data set is migrated.
PATHPERM(<i>octal path mode</i>)	Sets mode bits for a new allocation. This key is effectively the same as PATHMODE but accepts a simple octal number for the mode bit settings.
RELEASE	Space allocated to an output data set but not used is released when the data set is closed.
RTDDN(<i>variable</i>)	Returns allocated ddname into the REXX variable <i>variable</i> .
RTDSN(<i>variable</i>)	Return allocated data set name into the REXX variable <i>variable</i> .
RTVOL(<i>variable</i>)	Returns allocated volume name into the REXX variable <i>variable</i> .
SHORTRC	If the dynamic allocation fails, the dynamic allocation error code (S99ERROR) is returned in R15 and the information code is not returned. This is useful if the application can only process R15 as a halfword, such as with PL/I.
SYNTAX	When this is used, BPXWDYN parses the request and does not issue the dynamic allocation or dynamic output request.
TU(<i>hex-tu-value</i>)	Builds an allocation text unit (TU) from the specified string. For example, <code>alloc tu(000100010002c1c2)</code> is equivalent to <code>alloc fi(ab)</code> . TU can be used multiple times in an allocation request.

Table 4. Additional keys used for dynamic allocation (continued)

Additional keys used for dynamic allocation	Action
UCOUNT(<i>number</i>)	Specifies the number of devices to be allocated.

Requesting dynamic concatenation

To request dynamic concatenation, specify **concat** at the beginning of the parameter string.

This function concatenates multiple DDs to a single DD. The DDs are concatenated in the order specified in the DDLIST key.

Table 5. Keys used for dynamic concatenation

Keys used for dynamic concatenation	Action
DDLIST(<i>DDname1,DDname2[,DDnamex...]</i>)	Specifies a list of ddnames to concatenate to <i>DDname1</i> . Use this key with the CONCAT key.
DIAG (<i>stem</i> <i>fdnum</i> STDOUT)	Specifies that diagnostic information is to be displayed. The information includes a compile time stamp, the input string, key parsing information for subsequent keys, and the list of generated text units. Specify STDOUT to send output information to the standard REXX output stream.
MSG (WTP default.S99MSG. <i>stemname</i> <i>fdnum</i>)	Directs allocation messages to your job log (WTP), a REXX stem, or a file identified by a file descriptor number. If this key is not specified, messages are returned in the S99MSG. stem, if possible. If BPXWDYN was not called from a REXX environment, the messages will be lost.
PERMC	Specifies the permanently concatenated attribute.
SHORTRC	If the dynamic allocation fails, the dynamic allocation error code (S99ERROR) is returned in R15 and the information code is not returned. This is useful if the application can only process R15 as a halfword, such as with PL/I.

Requesting dynamic unallocation

To request dynamic unallocation, specify **free** at the beginning of the parameter string.

BPXWDYN supports single data set allocation. Many of the common unallocation keys can be used. For detailed information about those common keys, see *z/OS TSO/E Command Reference*. Some additional keys are supported to access additional functions.

Table 6. Common keys used for dynamic unallocation

Common keys used for dynamic allocation	Action
DA(<i>data set name[(member name)]</i>) DSN(<i>data set name[(member name)]</i>)	Specifies the data set name to free. The name must be fully qualified and can include a member name. Quotes are optional.
FI(<i>name</i>) DD(<i>name</i>)	Identifies the ddname to free.
KEEP DELETE CATALOG UNCATALOG	Overrides the data set disposition.

Table 6. Common keys used for dynamic unallocation (continued)

Common keys used for dynamic allocation	Action
SPIN(UNALLOC)	Spins off a sysout data set at unallocation.
SYSOUT(<i>class</i>)	Overrides the output class.

The following additional keys are unique to BPXWDYN.

Table 7. Additional keys used for dynamic unallocation

Additional keys for dynamic allocation	Action
DIAG (stem <i>fdnum</i> STDOUT)	Specifies that diagnostic information is to be displayed. The information includes a compile time stamp, the input string, key parsing information for subsequent keys, and the list of generated text units. Specify STDOUT to send output information to the standard REXX output stream.
MSG (WTP default.S99MSG. <i>stemname</i> <i>fdnum</i>)	Directs allocation messages to your job log (WTP), a REXX stem, or a file identified by a file descriptor number. If this key is not specified, messages are returned in the S99MSG. stem, if possible. If BPXWDYN was not called from a REXX environment, the messages will be lost. WTP should be specified to obtain messages.
SHORTRC	If the dynamic allocation fails, the dynamic allocation error code (S99ERROR) is returned in R15 and the information code is not returned. This is useful if the application can only process R15 as a halfword, such as with PL/I.
TU(<i>hex-tu-value</i>)	Builds an allocation text unit (TU) from the specified string. For example, <code>alloc tu(000100010002c1c2)</code> is equivalent to <code>alloc fi(ab)</code> . TU can be used multiple times in an allocation request.

Requesting allocation information

To request information about the current dynamic allocation environment, specify **info** at the beginning of the parameter string.

BPXWDYN currently supports the retrieval of the ddnames, data set names, and path names for current allocations; other allocation attributes are not supported.

BPXWDYN supports single data set allocation. You can use specific keys to specify which allocation requests information is to be returned for.

Table 8. Keys used to specify which allocation request information is to be returned for

Keys used in allocation requests	Action
DA(<i>data set name</i> [(<i>member name</i>)] DSN(<i>data set name</i> [(<i>member name</i>)]))	Specifies the data set name of the allocated resource about which you are requesting information. The <i>data set name</i> can contain special characters if the data set name is enclosed in apostrophes. It can also contain system symbols (see the section on using system symbols in <i>z/OS MVS JCL Reference</i> for more information). The maximum length of the data set name is 44 characters, excluding any enclosing apostrophes and compressing any double apostrophes within the data set name. This key is mutually exclusive with the ddname (FI, DD), path name (PATH), and relative entry (INRELNO) keys.
FI(<i>name</i>) DD(<i>name</i>)	Specifies the ddname of the allocated resource about which you are requesting information. <i>name</i> contains the ddname. This key is mutually exclusive with the data set name (DA, DSN), path name (PATH), and relative entry (INRELNO) keys.
PATH(<i>pathname</i>)	Specifies the path name of the file for which you are requesting information. <i>pathname</i> contains the path name and can contain system symbols (see the section on using system symbols in <i>z/OS MVS JCL Reference</i> for more information). This key is mutually exclusive with the data set name (DA, DSN), ddname (FI, DD), and relative entry (INRELNO) keys.

The following keys are unique to BPXWDYN.

Table 9. Additional keys used for dynamic information retrieval

Additional keys for dynamic retrieval	Action
DIAG (stem <i>fdnum</i> STDOUT)	Specifies that diagnostic information is to be displayed. The information includes a compile time stamp, the input string, key parsing information for subsequent keys, and the list of generated text units. Specify STDOUT to send output information to the standard REXX output stream.
MSG (WTP default.S99MSG. <i>stemname</i> <i>fdnum</i>)	Directs messages to your job log (WTP), a REXX stem, or a file identified by a file descriptor number. If this key is not specified, messages are returned in the S99MSG stem, if possible. If BPXWDYN was not called from a REXX environment, the messages will be lost. WTP should be specified to obtain messages.
INRCNTL(<i>variable</i>)	Requests return of the JCL CNTL statement reference.
INRDSNT(<i>variable</i>)	Requests return of the data set type. Upon return to the program, the REXX variable <i>variable</i> contains one of the following values or is blank: <ul style="list-style-type: none"> • PDS • PIPE • HFS
INRDAACL(<i>variable</i>)	Requests return of the data class of the specified SMS-managed data set. Upon return to the program, the REXX variable <i>variable</i> contains the data class identifier or is blank.

Table 9. Additional keys used for dynamic information retrieval (continued)

Additional keys for dynamic retrieval	Action
INRELNO(<i>request number</i>)	Specifies the relative request number of the allocation for which you are requesting information. <i>request number</i> contains the relative request number. This key is mutually exclusive with the data set name (DA, DSN), ddname (FI, DD), and path name (PATH) keys.
INRKEYO(<i>variable</i>)	Requests return of the key offset of a new VSAM data set.
INRLIKE(<i>variable</i>)	Requests return of the data set name on the LIKE parameter.
INRMGCL(<i>variable</i>)	Requests return of the management class of the specified SMS-managed data set. Upon return to the program, the REXX variable <i>variable</i> contains the management class identifier or is blank.
INRPMDE(<i>variable</i>)	Requests return of the open mode for the HFS file. Upon return to the program, the REXX variable <i>variable</i> contains a comma-separated list with any of the following values or is blank: SISUID SIGID SIRUSR SIWUSR SIXUSR SIRGRP SIWGRP SIXGRP SIROTH SIWOTH SIXOTH
INRPOPT(<i>variable</i>)	Requests return of the open options for the HFS file. Upon return to the program, the REXX variable <i>variable</i> contains a comma-separated list with any of the following values or is blank: OCREAT OEXCL ONOCTTY OTRUNC OAPPEND ONONBLOCK ORDWR ORDONLY OWRONLY
INRPNDS(<i>variable</i>)	Requests return of the normal disposition for the HFS file. Upon return to the program, the REXX variable <i>variable</i> contains one of the following values or is blank: DELETE KEEP
INRRECO(<i>variable</i>)	Requests return of the organization of records in the specified VSAM data set. Upon return to the program, the REXX variable <i>variable</i> contains one of the following values: LS Linear space RR Relative record ES Entry-sequenced KS Key-sequenced

Table 9. Additional keys used for dynamic information retrieval (continued)

Additional keys for dynamic retrieval	Action
INRKEYO(<i>variable</i>)	Requests return of the key offset of a new VSAM data set.
INRLIKE(<i>variable</i>)	Requests return of the data set name on the LIKE parameter.
INRREFD(<i>variable</i>)	Requests return of the DD name specified by the REFDD parameter of the DD statement.
INRSECM(<i>variable</i>)	Requests return of the RACF security data set profile.
INRSEGM(<i>variable</i>)	Requests return of the number of logical, line-mode pages to be produced for a SYSOUT data set before the segment becomes eligible for immediate printing.
INRSPIN(<i>variable</i>)	Requests return of the spin data set specification.
INRTATT(<i>variable</i>)	Requests return of the dynamic allocation attribute specification. This value is returned as two hexadecimal digits in character form.
INRTCDP(<i>variable</i>)	Requests return of the conditional disposition of the data set. Upon return to the program, the REXX variable <i>variable</i> contains one of the following values: <ul style="list-style-type: none"> • CATALOG • DELETE • KEEP • PASS • UNCATALOG
INRTLIM(<i>variable</i>)	Requests return of the number of resources that must be deallocated before making a new allocation.
INRTCDP(<i>variable</i>)	Requests return of the conditional disposition of the data set. Upon return to the program, the REXX variable <i>variable</i> contains one of the following values or is blank: <ul style="list-style-type: none"> • CATALOG • DELETE • KEEP • PASS • UNCATALOG
INRTDDN(<i>variable</i>)	Requests the ddname associated with the specified allocation. Upon return to the program, the REXX variable <i>variable</i> contains the requested ddname. If the data set you specify is a member of a concatenated group and is not the first member, there is no ddname associated with it.
INRTLIM(<i>variable</i>)	Requests return of the number of resources that must be deallocated before making a new allocation.
INRTLST(<i>variable</i>)	Requests an indication of whether the relative entry request number, ddname, or data set name you specify is the last relative entry. Upon return to the program, the REXX variable <i>variable</i> contains one of the following values: <p>128 Last relative entry 0 Not the last relative entry</p>
INRTMEM(<i>variable</i>)	Requests the PDS member name.
INRTDSN(<i>variable</i>)	Requests the data set name of the specified allocation. Upon return to the program, the REXX variable <i>variable</i> contains the requested data set name.

Table 9. Additional keys used for dynamic information retrieval (continued)

Additional keys for dynamic retrieval	Action
INRTNDP(<i>variable</i>)	Requests the normal disposition of the resource. Upon return to the program, the REXX variable <i>variable</i> contains one of the following values or is blank: UNCATALOG CATALOG DELETE KEEP PASS
INRTORG(<i>variable</i>)	Requests the data sets organization. One of the following strings is returned: blanks, TR, VSAM, TQ, TX, GS, PO, POU, MQ, CQ, CX, DA, DAU, PS, PSU, IS, ISU, or four hexadecimal digits as returned from SVC99.
INRTPATH(<i>variable</i>)	Requests the path name for the z/OS UNIX file associated with the specified allocation. Upon return to the program, the REXX variable <i>variable</i> contains the path name, if any, associated with the allocation.
INRTSTA(<i>variable</i>)	Requests the data set status of the allocation. Upon return to the program, the REXX variable <i>variable</i> contains one of the following values or is blank: OLD MOD NEW SHR
INRSTCL(<i>variable</i>)	Requests return of the storage class of the specified SMS-managed data set. Upon return to the program, the REXX variable <i>variable</i> contains the storage class identifier or is blank.
INRTTYP(<i>variable</i>)	Requests the data set type. One of the following strings is returned: blanks, DUMMY, TERM, SYSIN, SYSOUT, or two hexadecimal digits as returned from SVC99.
INRTVOL(<i>variable</i>)	Requests the return of the first volume serial number associated with the specified allocation.

Requesting dynamic output

To request dynamic output, specify **outdes** at the beginning of the parameter string. This keyword takes an argument that names the output descriptor. The keyword is described in Table 10, which also lists the rest of the supported keys.

Table 10. Keys used for dynamic output

Keys used for dynamic output	Action
OUTDES(<i>descriptor name</i>)	Names the output descriptor to be added. This must be the first key specified in the parameter string.
ADDRESS(<i>address</i> [, <i>address</i> ...])	Specifies the delivery address. Dynamic output allows up to four arguments to be specified.
BUILDING(<i>building</i>)	Specifies the building location.
BURST	Directs output to a stacker.
CHARS(<i>chars</i> [, <i>chars</i> ...])	Names the character arrangement tables.
CLASS(<i>class</i>)	Assigns the SYSOUT class.

Table 10. Keys used for dynamic output (continued)

Keys used for dynamic output	Action
CONTROL(<i>spacing</i>)	Specifies either line spacing or that the records begin with carriage control characters. The valid values are: Single Single spacing Double Double spacing Triple Triple spacing Program Records begin with carriage control characters
COPIES(<i>number of copies</i>)	Specifies the number of copies to be printed.
DEFAULT	Specifies that this is a default output descriptor.
DEPT(<i>department</i>)	Specifies the department identification.
DEST(<i>node[.user]</i>)	Sends the SYSOUT to the specified destination.
DIAG (<i>stem fdnum STDOUT</i>)	Specifies that diagnostic information is to be displayed. The information includes a compile time stamp, the input string, key parsing information for subsequent keys, and the list of generated text units. Specify STDOUT to send output information to the standard REXX output stream.
DPAGELBL	Specifies that a security label be placed on the output.
FCB(<i>fcbl name</i>)	Specifies the FCB image.
FLASH	Specifies the forms overlay.
FORMDEF(<i>formdef name</i>)	Names the formdef.
FORMS(<i>forms name</i>)	Names the forms to print on.
MAILFROM(<i>sender name</i>)	Specifies a string that contains the name of the sender.
MAILTO(<i>destination[.destination...]</i>)	Specifies one or more email addresses of the recipients.
MODIFY(<i>trc number</i>)	Specifies which character arrangement table is to be used.
NAME(<i>owner name</i>)	Specifies the owner's name.
NOTIFY(<i>[node.]user</i>)	Sends print completion message to the destination.
OUTDISP(<i>disposition</i>)	Specifies the data set disposition.
PAGEDEF(<i>pagedef name</i>)	Names the pagedef.
PRMODE(<i>print mode</i>)	Identifies the process mode (LINE or PAGE).
ROOM(<i>room identification</i>)	Specifies the room identification.
TITLE(<i>separator title</i>)	Specifies the separator page title.
TRC	Specifies that the data set contains TRC codes.
UCS(<i>UCS name</i>)	Names the UCS or character arrangement table.
USERDATA(<i>userdata[,userdata...]</i>)	Specifies installation-specific user data for a dynamic output statement. Dynamic output allows up to 16 strings that consist of 1 to 60 character strings. Case is preserved if the string is quoted.
USERLIB(<i>dsn[,dsn...]</i>)	Specifies the names of libraries that contain AFP resources.
WRITER(<i>external writer name</i>)	Names an external writer to process the data set.

Freeing an output descriptor

To request that an output descriptor be freed, specify **free** at the beginning of the parameter string. Only one keyword is supported:

Table 11. Key used to free an output descriptor

Key used to free output descriptor	Action
OUTDES(<i>descriptor name</i>)	Names the output descriptor to be freed.

Examples: Calling BPXWDYN from a REXX program

Allocate

This example allocates SYS1.MACLIB to SYSLIB and directs messages to z/OS UNIX standard error (stderr):

```
call bpxwdyn "alloc fi(syslib) da(sys1.maclib) shr msg(2)"
```

Info

This example requests that the name of the data set allocated to ddname SYSLIB be returned in the REXX variable *dsnvar*.

```
call bpxwdyn "info fi(syslib) inrtdsn(dsnvar)"
```

Free

This example frees SYSLIB and traps messages in stem S99MSG.:

```
call bpxwdyn "free fi(syslib)"
```

Concatenate

```
if bpxwdyn("alloc fi(tmp) da(sys1.sbpexec) shr msg(2)")=0 then  
  call bpxwdyn "concat ddlist(sysproc,tmp) msg(2)"
```

Create dynamic output descriptor

This example creates descriptor P20 with distribution information:

```
call bpxwdyn "outdes(p20) dest(kgn.p20n10)",  
  "address('WJ Schoen','M/S 619')",  
  "name(wschoen) bin(0004) dept(64ba)"
```

Free descriptor

This example frees descriptor P20:

```
call bpxwdyn "free outdes(p20)"
```

Example: calling BPXWDYN from C

Allocate

This example allocates SYS1.MACLIB to SYSLIB and directs messages to standard error (stderr):

```
typedef int EXTf();  
#pragma linkage(EXTf,OS_UPSTACK)  
  
int call_alloc()  
{  
  EXTf *bpxwdyn=(EXTf *)fetch("BPXWDYN");  
  return bpxwdyn("alloc fi(syslib) da(sys1.maclib) shr msg(2)")  
}
```

Chapter 7. Virtual file system (VFS) server syscall commands

A number of syscall commands are intended for file system server applications, such as a Network File System server. Although it is unlikely that you would implement a Network File System server using REXX, you can access the server callable services using z/OS UNIX REXX syscall commands. Because it is possible for a server to create a file in the file hierarchy with a name that cannot be accessed through conventional C functions, for example, a file name that has a slash (/) in it, these syscall commands might be useful if you need to obtain local access to such a file.

For more information about these services, see *z/OS UNIX System Services File System Interface Reference*.

Security

The file system server services are available only to a registered server. Only a superuser can use the callable service `v_reg` to register the process as a server.

The server services can bypass system security for file access.

For examples of REXX coding using these commands, see Appendix A, “REXX predefined variables,” on page 241.

Tokens

Many tokens flow across the server interface. The types of tokens are:

VFS Represents a mounted file system.

vnode Represents a file or directory that is currently in use. This identifier is valid only until the token is released, using the `v_rel` callable service.

FID Uniquely identifies a file or directory in a particular mounted file system. The file or directory may or may not be currently in use. This identifier is valid across mounting and unmounting of the file system, as well as across z/OS UNIX re-IPLs.

You must specify tokens as variable names, not as strings. See “Specifying a syscall command” on page 19 for information about specifying variable names. When a token is returned to the `exec`, the value of the token is stored in the variable. When a token variable is used as a parameter on a syscall command, the token value is extracted from the variable. The format for a token is 8 bytes of binary data.

`v_close`



►►—`v_close`—`vntoken`—`opentoken`—◄◄

v_close

Function

v_close closes a file previously opened using v_open.

Parameters

vntoken

A variable name that contains the vnode token for the directory of the file that is to be closed.

opentoken

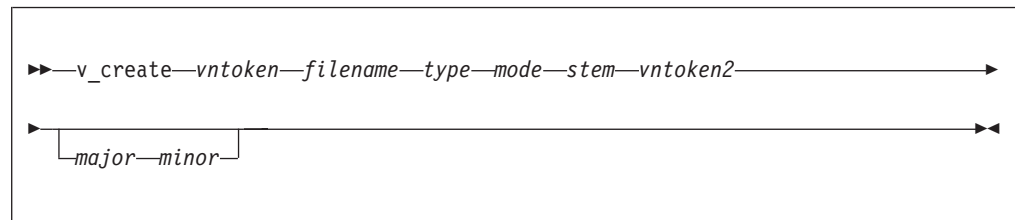
A variable name that contains the open token as returned from the v_open.

Example

Close a file represented by vntok with a v_open represented by otok:

```
"v_close vntok otok"
```

v_create



Function

v_create invokes the v_create callable service to create a new file in the directory represented by *vntoken*. The file can be a regular, FIFO, or character special file.

Parameters

vntoken

A variable name that contains the vnode token for the directory in which the file *filename* is to be created.

filename

The name of the file. It must not contain null characters.

type

A number used to specify the type of file to be created: a regular, FIFO, or character special file. You can specify one of the predefined variables beginning with *S_* to set the value—for example, *S_ISREG*. For a list of the variables, see “fstat” on page 65.

mode

A three-digit number, corresponding to the access permission bits. Each digit must be in the range 0–7, and all three digits must be specified. For more information about permissions, see Appendix A, “REXX predefined variables,” on page 241.

stem

The name of a stem variable used to return the status information. Upon return, *stem.0* contains the number of variables that are returned. To obtain the desired status information, you can use a numeric value or the predefined

variables beginning with ST_ used to derive the numeric value. See “stat” on page 140 for a list of the predefined variables or Appendix A, “REXX predefined variables,” on page 241 for the numeric values.

vntoken2

A variable name that will contain the vntoken of the created file upon return.

major

For a character special file (S_ISCHR), the device major number. For a complete description, see “mknod” on page 94.

minor

For a character special file (S_ISCHR), the device minor number. For a complete description, see “mknod” on page 94.

Usage notes

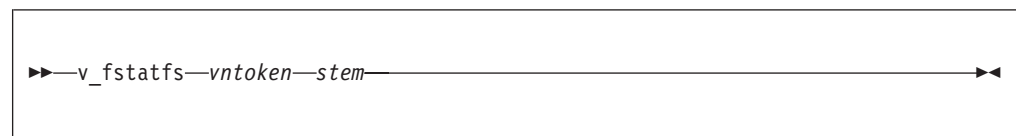
1. If the file named in *filename* already exists, the v_create service returns a failing return code, and no vnode token is returned.
2. The caller is responsible for freeing vnode tokens returned by the service by calling to the v_rel service when they are no longer needed.

Example

In the following example, assume that *filenm* and *vnod* were assigned values earlier in the exec:

```
"v_create vnod (filenm)" s_isreg 777 "st. filetok"
```

v_fstatfs



Function

v_fstatfs invokes the v_fstatfs callable service to return file system status for the file system containing the file or directory represented by the specified *vntoken*.

Parameters

vntoken

A variable name that contains the vnode token for a file or directory in the file system whose status is to be checked.

stem

The name of a stem variable used to return the status information. Upon return, *stem.0* contains the number of variables that are returned. To obtain the desired status information, you can use a numeric value or the predefined variables beginning with STFS_ used to derive the numeric value. For example, *stem.stfs_avail* accesses the number of blocks available in the file system. See “statfs” on page 142 for a list of the predefined variables, or Appendix A, “REXX predefined variables,” on page 241 for the numeric values.

v_fstatfs

Example

In the following example, assume that *vnod* was assigned a value earlier in the exec:

```
" v_fstatfs vnod st."
```

v_get

```
>>v_get--vfstoken--fid--vntoken<<<
```

Function

v_get invokes the **v_get** callable service to return a vnode token for the file or directory represented by the input FID within the mounted file system represented by the input VFS token.

Parameters

vfstoken

A variable name that contains the VFS token for the file system where the file identified by *fid* resides.

fid

A variable that contains a file ID. File IDs are returned in the file attribute structure in the stem index **ST_FID**.

vntoken

A variable name that stores the vnode token for the requested file.

Usage notes

1. The FID (file identifier) uniquely identifies a file in a particular mounted file system, and its validity persists across mounting and unmounting of the file system, as well as z/OS UNIX re-IPLs. This distinguishes the FID from the vnode token, which relates to a file in active use, and whose validity persists only until the token is released via the **v_rel** callable service.
A server application uses **v_get** to convert an FID to a vnode token when it is preparing to use a file, since the vnode token identifies the file to the other services.
2. The FID for a file is returned in a stem variable by such services as **v_rpn** and **v_lookup**.
3. The caller is responsible for freeing vnode tokens returned by **v_get** by calling to the **v_rel** service when they are no longer needed.

Example

In the following example, assume that *vfs* and *st.st_fid* were assigned values earlier in the exec:

```
"v_get vfs st.st_fid vnod"
```

v_getattr

```
▶▶—v_getattr—vntoken—stem—————▶▶
```

Function

v_getattr invokes the `v_getattr` callable service to get the attributes of the file represented by *vntoken*.

Parameters

vntoken

A variable name that contains the vnode token of the file for which the attributes are returned.

stem

The name of a stem variable used to return the file attribute information. Upon return, *stem.0* contains the number of attribute variables returned. To obtain the desired information, you can use a numeric value or the predefined variables beginning with `ST_` used to derive the numeric value. See “stat” on page 140 for a list of the variables, or Appendix A, “REXX predefined variables,” on page 241 for the numeric values.

Example

In the following example, assume that *vnod* was assigned a value earlier in the exec:

```
"v_getattr vnod attr."
```

v_link

```
▶▶—v_link—vntoken—filename—vntoken2—————▶▶
```

Function

v_link invokes the `v_link` callable service to create a link to the file specified by *vntoken* in the directory specified by *vntoken2*. The link is a new name, *filename*, identifying an existing file.

Parameters

vntoken

A variable name that contains the vnode token for the file being linked to.

filename

The new name for the existing file.

v_link

vntoken2

A variable name that contains the vnode token for the directory to which *filename* is to be added.

Usage notes

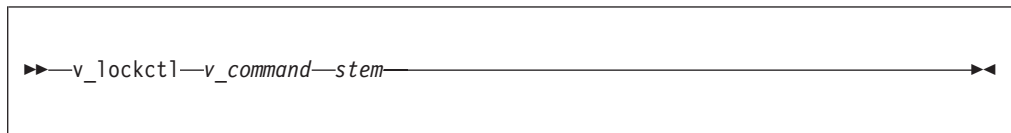
1. **v_link** creates a link named *filename* to an existing file specified by *vntoken*. This provides an alternative path name for the existing file, so that you can access it by the old name or the new name. You can store the link under the same directory as the original file, or under a different directory on the same file system.
2. If the link is created successfully, the service routine increments the link count of the file. The link count shows how many links to a file exist. (If the link is not created successfully, the link count is not incremented.)
3. Links are not allowed to directories.
4. If the link is created successfully, the change time of the linked-to file is updated, as are the change and modification times of the directory that contains *filename*—that is, the directory that holds the link.

Example

In the following example, assume that *filetok*, *name*, and *dirtok* were assigned values earlier in the exec:

```
"v_link filetok (name) dirtok"
```

v_lockctl



Function

v_lockctl invokes the **v_lockctl** callable service to control advisory byte-range locks on a file.

Note: All locks are advisory only. Client and local processes can use locks to inform each other that they want to protect parts of a file, but locks do not prevent I/O on the locked parts. A process that has appropriate permissions on a file can perform whatever I/O it chooses, regardless of which locks are set. Therefore, file locking is only a convention, and it works only when all processes respect the convention.

Parameters

v_command

The name of a predefined command variable that is used to control the lock. You can specify a numeric value (see Appendix A, “REXX predefined variables,” on page 241) or a predefined VL_command variable that derives the appropriate numeric value. The command variables are

Variable	Description
VL_REGLOCKER	Register the lock server (locker).

Variable	Description
VL_UNREGLOCKER	Unregister the locker.
VL_LOCK	Set a lock in a specified byte range.
VL_LOCKWAIT	Set a lock in a specified byte range or wait to set the lock until the byte range is free.
VL_UNLOCK	Unlock all locks in a specified byte range.
VL_QUERY	Query for lock information about a file.
VL_PURGE	Release all locks on all files, held by a locker or a group of lockers.

stem

The name of a stem variable that is the structure used to obtain information about the lock. To access the information, you can specify a numeric value (see Appendix A, “REXX predefined variables,” on page 241) or the predefined variables beginning with VL_ or L_ that derive the appropriate numeric value—for example, *stem.vl_serverpid*. The variables beginning with VL_ are:

Variable	Description
VL_SERVERPID	Server's PID
VL_CLIENTPID	Server's client's process ID (PID)
VL_LOCKERTOK	Token for locker
VL_CLIENTTID	Client's thread's TID. The TID is the individual lock owner within a locker.
VL_OBJCLASS	The class for an object (a single locked file)—for example, HFS for an HFS file, MVS for an MVS data set, LFSESA for a LAN file server.
VL_OBJID	The unique ID for an object (locked file) within its class. For an HFS file, the VL_OBJID contains the device number and FID of the file.
VL_OBJTOK	A token to identify the object (locked file) on a subsequent lock request.
VL_DOSMODE	DOS file-sharing field
VL_DOSACCESS	DOS file-sharing field

For a description of the variables beginning with L_, see “f_getlk” on page 54.

Usage notes

1. The v_lockctl service locks out other cooperating lockers from part of a file, so that the locker can read or write to that part of the file without interference from others.
2. Each locker must be registered before issuing any lock requests. On a REGLOCKER command, the caller must provide stem variables with these suffixes:
 - VL_SERVERPID
 - VL_CLIENTPID
 The VL_LOCKERTOK variable is returned to the caller; it is a token to identify the locker on subsequent lock requests.
3. On a QUERY, LOCK, LOCKWAIT, or UNLOCK command, the caller provides stem variables with these suffixes:
 - VL_LOCKERTOK
 - VL_CLIENTTID
 - VL_OBJCLASS
 - VL_OBJID
 - VL_OBJTOK (This is optional, but it will improve performance for multiple lock requests)

To describe the byte range for the command, the caller must also provide stem variables with the following L_ suffixes:

QUERY

L_TYPE, L_START, L_LEN, L_WHENCE

LOCK L_TYPE, L_START, L_LEN, L_WHENCE

LOCKWAIT

L_TYPE, L_START, L_LEN, L_WHENCE

UNLOCK

L_START, L_LEN, L_WHENCE

The L_ variables are described in “f_getlk” on page 54.

VL_OBJTOK is returned to the caller; it is a token to identify the object on a subsequent lock request. On a QUERY, lock information describing a lock that would prevent the proposed lock from being set is returned to the caller.

4. Stem variables with the suffixes L_TYPE, L_START, and L_LEN are needed whether the request is for setting a lock, releasing a lock, or querying a particular byte range for a lock. L_WHENCE is always treated as SEEK_SET, the start of the file.

The L_TYPE variable is used to specify the type of lock to be set or queried. (L_TYPE is not used to unlock.) You can use a numeric value (see Appendix A, “REXX predefined variables,” on page 241) or one of the following predefined variables used to derive the appropriate value:

Type Description**F_RDLCK**

A read lock, also known as a shared lock. This type of lock specifies that the locker can read the locked part of the file, and other lockers cannot write on that part of the file in the meantime. A locker can change a held write lock, or any part of it, to a read lock, thereby making it available for other lockers to read. Multiple lockers can have read locks on the same part of a file simultaneously.

F_WRLCK

A write lock, also known as an exclusive lock. This type of lock indicates that the locker can write on the locked part of the file, without interference from other lockers. If one locker puts a write lock on part of a file, no other locker can establish a read lock or write lock on that same part of the file. A locker cannot put a write lock on part of a file if there is already a read lock on an overlapping part of the file, unless that locker is the only owner of that overlapping read lock. In such a case, the read lock on the overlapping section is replaced by the write lock being requested.

F_UNLCK

Returned on a query, when there are no locks that would prevent the proposed lock operation from completing successfully.

The L_WHENCE variable specifies how the byte range offset is to be found within the file; L_WHENCE is always treated as SEEK_SET, which stands for the start of the file.

The L_START variable is used to identify the part of the file that is to be locked, unlocked, or queried. The part of the file affected by the lock begins at this offset from the start of the file. For example, if L_START has the value 10, a lock request attempts to set a lock beginning 10 bytes past the start of the file.

Note: Although you cannot request a byte range that begins or extends beyond the beginning of the file, you can request a byte range that begins or extends beyond the end of the file.

The `L_LEN` variable is used to give the size of the locked part of the file, in bytes. The value specified for `L_LEN` cannot be negative. If a negative value is specified for `L_LEN`, a `RETV` of -1 and an `EINVAL` `ERRNO` are returned. If `L_LEN` is zero, the locked part of the file begins at `L_START` and extends to the end of the file.

The `L_PID` variable identifies the `VL_CLIENTPID` of the locker that holds the lock found on a query request, if one was found.

5. You can set locks by specifying a `VL_LOCK` as the command parameter. If the lock cannot be obtained, a `RETV` of -1 is returned along with an appropriate `ERRNO` and `ERRNOJR`.

You can also set locks by specifying a `VL_LOCKWAIT` as the command parameter. If the lock cannot be obtained because another process has a lock on all or part of the requested range, the `LOCKWAIT` request waits until the specified range becomes free and the request can be completed.

If a signal interrupts a call to the `v_lockctl` service while it is waiting in a `LockWait` operation, the function returns with a `RETV` of -1, and the `ERRNO` `EINTR`.

`LockWait` operations have the potential for encountering deadlocks. This happens when locker A is waiting for locker B to unlock a byte range, and B is waiting for A to unlock a different byte range. If the system detects that a `LockWait` might cause a deadlock, the `v_lockctl` service returns with a `RETV` of -1 and the `ERRNO` `EDEADLK`.

6. A process can determine locking information about a file using `VL_QUERY` as the command parameter. The stem should describe a lock operation that the caller would like to perform. When the `v_lockctl` service returns, the structure is modified to describe the first lock found that would prevent the proposed lock operation from finishing successfully.

If a lock is found that would prevent the proposed lock from being set, the query request returns a stem whose `L_WHENCE` value is always `SEEK_SET`, whose `L_START` value gives the offset of the locked portion from the beginning of the file, whose `L_LEN` value is set to the length of the locked portion of the file, and whose `L_PID` value is set to the `ClientProcessID` of the locker that is holding the lock. If there are no locks that would prevent the proposed lock operation from finishing successfully, the returned structure is modified to have an `L_TYPE` of `F_UNLCK`, but otherwise it remains unchanged.

7. A locker can have several locks on a file simultaneously but can have only one type of lock set on any given byte. Therefore, if a locker sets a new lock on part of a file that it had previously locked, the locker has only one lock on that part of the file, and the lock type is the one given by the most recent locking operation.
8. When a `VL_UNLOCK` command is issued to unlock a byte range of a file, all locks held by that locker within the specified byte range are released. In other words, each byte specified on an unlock request is freed from any lock that is held against it by the requesting locker.
9. Each locker should be unregistered when done issuing lock requests. On a `VL_UNLOCKER` command, the caller provides the stem variable `VL_LOCKERTOK` to identify the locker to unregister.
10. The `VL_PURGE` command releases all locks on all files, held by a locker or a group of lockers. The following stem variables are provided by the caller:

v_lockctl

VL_SERVERPID
VL_CLIENTPID
VL_CLIENTTID

Example

This example illustrates several calls to `v_lockctl` to register a locker, lock a range, unlock a range, and unregister a locker:

```
/* rexx */
address syscall
'v_reg 2 RxLocker'          /* register server as a lock server */
/*****
/* register locker
/*****
lk.vl_serverpid=0          /* use my pid as server pid */
lk.vl_clientpid=1         /* set client process id */
'v_lockctl' vl_reglocker 'lk.' /* register client as a locker */
cltok=lk.vl_lockertok     /* save client locker token */
/*****
/* lock a range
/*****
lk.vl_lockertok=cltok     /* set client locker token */
lk.vl_clienttid='thread1' /* invent a thread id */
lk.vl_objclass=1         /* invent an object class */
lk.vl_objid='objectname' /* invent an object name */
lk.vl_objtok=''         /* no object token */
lk.l_len=40              /* set length of range to lock */
lk.l_start=80            /* set start of range to lock */
lk.l_whence=seek_set     /* start of range is absolute */
lk.l_type=f_wrlck       /* set write lock */
'v_lockctl' vl_lock 'lk.' /* try to do the lock */
obj1=lk.vl_objtok        /* keep returned object token */
/*****
/* unlock a range
/*****
lk.vl_lockertok=cltok     /* set client locker token */
lk.vl_clienttid='thread1' /* invent a thread id */
lk.vl_objclass=1         /* invent an object class */
lk.vl_objid='objectname' /* invent an object name */
lk.vl_objtok=obj1        /* set object token */
lk.l_len=40              /* set length of range to lock */
lk.l_start=80            /* set start of range to lock */
lk.l_whence=seek_set     /* start of range is absolute */
lk.l_type=f_unlck       /* set unlock */
'v_lockctl' vl_unlock 'lk.' /* unlock the range */
/*****
/* unregister locker
/*****
lk.vl_lockertok=cltok     /* set client locker token */
'v_lockctl' vl_unreglocker 'lk.' /* unregister client as a locker */
return
```

v_lookup

►►—v_lookup—vntoken—filename—stem—vntoken2—◄◄

Function

v_lookup invokes the v_lookup callable service to search a directory for a file. v_lookup accepts a vnode token representing a directory and a name identifying a file. If the file is found in the directory, a vnode token for the file and the attributes of the file are returned.

Parameters

vntoken

A variable name that contains the vnode token for the directory in which *filename* is looked up.

filename

The name of the file.

stem

The same file attribute information is returned in *stem* as if a v_getattr had been used on the file looked up. Upon return, *stem.0* contains the number of attribute variables returned. To access the attribute values, you can use a numeric value or the predefined variables beginning with ST_ used to derive the numeric value. See “stat” on page 140 for a list of the predefined variables, or Appendix A, “REXX predefined variables,” on page 241 for the numeric values.

vntoken2

The variable name for the buffer that, when returned, will contain the vnode token for the looked-up file.

Usage notes

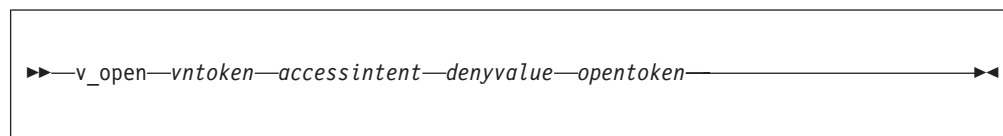
The caller is responsible for freeing vnode tokens returned by the v_lookup service by calling to the v_rel service when they are no longer needed.

Example

In the following example, assume that *dirtok* and *file* were assigned values earlier in the exec:

```
"v_lookup dirtok (file) st. outtok"
```

v_open



Function

v_open opens an existing file and optionally establishes share reservations on the file.

v_open

Parameters

vntoken

A variable name that contains the vnode token for the directory of the file that is to be opened.

accessintent

A number used to specify the intended access mode, read, write, or both. You can specify one of the predefined variables to set the value:

- o_ronly
- o_wronly
- o_rdwr

denyvalue

A number used to specify access modes to be denied, read, write, or both. You can specify one of the predefined variables to set the value:

- o_ronly
- o_wronly
- o_rdwr

opentoken

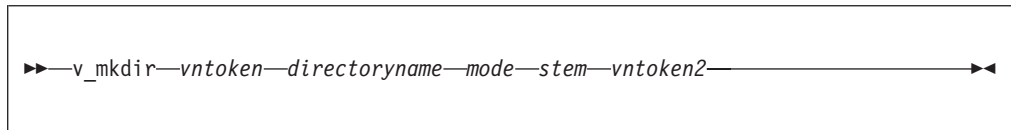
A variable name that will contain the open token on successful completion of the service. The open token can be used on v_close, v_read, _setattr, and v_write services to perform the operation under context of this open.

Example

Open a file represented by vntok for read and deny opens for write:

```
"v_open vntok" o_ronly o_wronly "otok"
```

v_mkdir



Function

v_mkdir invokes the v_mkdir callable service to create a new empty directory in the directory represented by *vntoken*, with the permission specified in *mode*.

Parameters

vntoken

A variable name that contains the vnode token for the directory in which *filename* is to be created.

directoryname

The name of the directory.

mode

A three-digit number, corresponding to the access permission bits for the directory. Each digit must be in the range 0–7, and all three digits must be specified. For more information on permissions, see Appendix A, “REXX predefined variables,” on page 241.

stem

The same file attribute information is returned in *stem* as if a `v_getattr` had been used on the file specified. Upon return, *stem.0* contains the number of attribute variables returned. To access the attribute information, you can use a numeric value or the predefined variables beginning with `ST_` used to derive the numeric value. See “stat” on page 140 for a list of the predefined variables, or Appendix A, “REXX predefined variables,” on page 241 for the numeric values.

vntoken2

The variable name for the buffer that will contain the vnode token for the newly created directory.

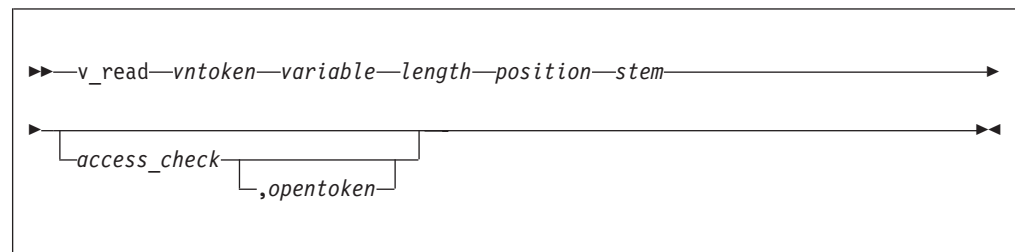
Usage notes

1. If the directory that is named in *directoryname* already exists, the `v_mkdir` service returns a failing return code, and no *vntoken2* is returned.
2. The caller is responsible for freeing vnode tokens returned by the `v_mkdir` service by calling to the `v_rel` service when they are no longer needed.

Example

In the following example, assume that *dirtok*, *file*, and *perm* were assigned values earlier in the exec:

```
"v_mkdir dirtok (file)" perm "st. newtok"
```

v_read**Function**

`v_read` invokes the `v_rdwr` callable service to accept a vnode token representing a file and to read data from the file. The file attributes are returned when the read completes. The number of bytes read is returned in `RETVAl`.

Parameters**vntoken**

A variable name that contains the vnode token for the file to be read.

variable

The name of the buffer into which data will be read.

length

The maximum number of characters to read. After the read completes, the length of *variable* is the number of bytes read. This value is also returned in `RETVAl`.

v_read

position

The file offset where the read is to begin, specified in bytes.

stem

The name of a stem variable used to return the file attribute information. Upon return, *stem.0* contains the number of attribute variables returned. The same information is returned in *stem* as if a `v_getattr` had been used on the file. To obtain the attribute information, you can use a numeric value or the predefined variables beginning with `ST_` used to derive the numeric value. See "stat" on page 140 for a list of the variables, or Appendix A, "REXX predefined variables," on page 241 for the numeric values.

access_check

Specify a 0 for no access check, or 1 to indicate the system is to check the user for read access to the file. The user is defined by the effective UID and GID. Setting the effective GID does not affect supplemental groups. Also, there is no support for altering the MVS user identity of the task using the ACEE (accessor environment element).

opentoken

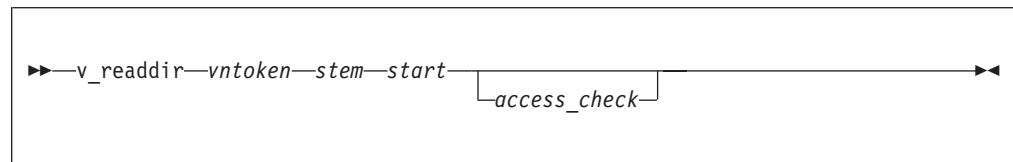
A variable name that contains the open token as returned from the `v_open` service. If the `opentoken` parameter is specified, the `access_check` parameter must also be specified.

Example

In the following example, assume that *filetok*, *bytes*, and *pos* were assigned values earlier in the exec:

```
"v_read filetok buffer" bytes pos stem. 0
```

v_readdir



Function

`v_readdir` invokes the `v_readdir` callable service to accept a vnode token representing a directory and return directory entries from this directory.

Parameters

vntoken

A variable name that contains the vnode token for the directory to be read.

stem

The name of a stem variable used to return the directory entries. Upon return, *stem.0* contains the number of directory entries returned. *stem.1* through *stem.n* (where *n* is the number of entries returned) each contain a directory entry.

Note: Only small directories can be read in a single call. To ensure that you read to the end of the directory, make calls to `v_readdir` until no entries are returned.

start

The number of the first directory entry to be returned. The numbers 0 and 1 both indicate that the read should start at the beginning of the directory.

access_check

Specify a 0 for no access_check, or 1 to specify that the system is to check the user for read access to the file. The user is defined by the effective UID and GID. Setting the effective GID does not affect supplemental groups. Also, there is no support for altering the MVS user identity of the task using the ACEE.

Example

In the following example, assume that *dirtok* was assigned a value earlier in the exec:

```
"v_readdir dirtok dir. 0"
```

v_readlink

```
▶▶—v_readlink—vntoken—variable————▶▶
```

Function

v_readlink invokes the v_readlink callable service to read the symbolic link file represented by the vnode token and return its contents in *variable*.

Parameters**vntoken**

A variable name that contains the vnode token for the symbolic link to be read.

The attribute stem returned on call to another function (for example, v_getattr) identifies whether the symbolic link is a link to an external name in the stem index ST_EXTLINK. An external name is the name of an object outside the HFS.

variable

The name of the buffer that, upon return, contains the contents of the symbolic link.

Example

In the following example, assume that *symtok* was assigned a value earlier in the exec:

```
"v_readlink symtok link"
```

v_reg

```
▶▶—v_reg—type—name————▶▶
```

v_reg

Function

v_reg invokes the **v_reg** callable service to register a process as a server. A process must be registered using this service before it can use any other vnode interface services.

Parameters

type

A numeric value that defines the type of server. You can specify:

- 1 to indicate a file server
- 2 to indicate a lock server

name

The name of the server, a character string up to 32 bytes in length. There are no restrictions on the name; for example, it does not have to be unique in the system.

Usage notes

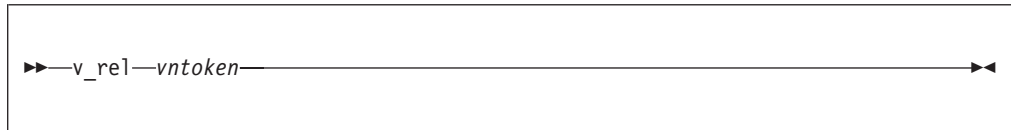
1. Only a superuser can register a process as a server.
2. The D OMVS command uses the values supplied in *type* and *name* fields to display information about the currently active servers.

Example

To register a server:

```
'v_reg 1 "My REXX server"'
```

v_rel



Function

v_rel invokes the **v_rel** callable service to accept a vnode token representing a file descriptor for a file or directory, and to release that token.

Parameters

vntoken

A variable name that contains the vnode token for the file descriptor to be released.

Usage notes

1. The vnode token is no longer valid and cannot be used for subsequent requests after **v_rel** has successfully processed it.
2. This service must be used to release all vnode tokens obtained from other operations.

Example

In the following example, assume that *vntok* was assigned a value earlier in the exec:

```
"v_rel vntok"
```

v_remove

```
▶—v_remove—vntoken—filename—▶
```

Function

v_remove invokes the v_remove callable service to remove a directory entry.

Parameters

vntoken

A variable name that contains the vnode token for the directory from which *filename* is to be removed.

filename

The name for the directory entry. The directory entry could be identified by a name for a file, the name of a hard link to a file, or the name of a symbolic link.

Usage notes

1. If the name specified refers to a symbolic link, the symbolic link file named by *filename* is deleted.
2. If the v_remove service is successful and the link count becomes zero, the file is deleted. The contents of the file are discarded, and the space it occupied is freed for reuse. However, if another process (or more than one) has the file open or has a valid vnode token when the last link is removed, the file contents are not removed until the last process closes the file or releases the vnode token.
3. When the v_remove service is successful in removing a directory entry and decrementing the link count, whether or not the link count becomes zero, it returns control to the caller with RETVAL 0. It updates the change and modification times for the parent directory, and the change time for the file itself (unless the file is deleted).
4. You cannot remove a directory using **v_remove**. To remove a directory, refer to “v_rmdir” on page 233.

Example

In the following example, assume that *dirtok* and *file* were assigned values earlier in the exec:

```
"v_remove dirtok (file)"
```

v_rename

```

  >>—v_rename—vntoken—oldname—vntoken2—newname—<<
  
```

Function

v_rename invokes the v_rename callable service to rename a file or directory to a new name.

Parameters

vntoken

A variable name that contains the vnode token for the directory that contains the filename *oldname*.

oldname

The existing name for the file or directory.

vntoken2

A variable name that contains the vnode token for the directory that is to contain the filename *newname*.

newname

The new name for the file or directory.

Usage notes

1. The v_rename service changes the name of a file or directory from *oldname* to *newname*. When renaming completes successfully, the change and modification times for the parent directories of *oldname* and *newname* are updated.
2. The calling process needs write permission for the directory containing *oldname* and the directory containing *newname*. If *oldname* and *newname* are the names of directories, the caller does not need write permission for the directories themselves.
3. **Renaming files:** If *oldname* and *newname* are links referring to the same file, **v_rename** returns successfully and does not perform any other action.
If *oldname* is the name of a file, *newname* must also name a file, not a directory. If *newname* is an existing file, it is unlinked. Then the file specified as *oldname* is given *newname*. The pathname *newname* always stays in existence; at the beginning of the operation, *newname* refers to its original file, and at the end, it refers to the file that used to be *oldname*.
4. **Renaming directories:** If *oldname* is the name of a directory, *newname* must also name a directory, not a file. If *newname* is an existing directory, it must be empty, containing no files or subdirectories. If it is empty, it is removed. *newname* cannot be a directory under *oldname*; that is, the old directory cannot be part of the pathname prefix of the new one.

Example

In the following example, assume that *olddir*, *oldfile*, *newdir*, and *newfile* were assigned values earlier in the exec:

```
"v_rename olddir (oldfile) newdir (newfile)"
```

v_rmdir

```
▶▶—v_rmdir—vntoken—dirname—◀◀
```

Function

`v_rmdir` invokes the `v_rmdir` callable service to remove an empty directory.

Parameters

`vntoken`

A variable name that contains the vnode token for the directory from which `dirname` is to be removed.

`dirname`

The name of the empty directory to be removed.

Usage notes

1. The directory specified by `dirname` must be empty.
2. If the directory is successfully removed, the change and modification times for the parent directory are updated.
3. If any process has the directory open when it is removed, the directory itself is not removed until the last process closes the directory. New files cannot be created under a directory that is removed, even if the directory is still open.

Example

In the following example, assume that `dirtok` and `dirname` were assigned values earlier in the exec:

```
"v_rmdir dirtok (dirname)"
```

v_rpn

```
▶▶—v_rpn—pathname—vfstoken—vntoken—stem—stem2—◀◀
```

Function

`v_rpn` invokes the `v_rpn` callable service to accept a pathname of a file or directory and return a vnode token that represents this file or directory and the VFS token that represents the mounted file system that contains the file or directory.

Parameters

`pathname`

The absolute pathname to be resolved, specified as a string.

v_rpn

vfstoken

The name of a variable in which the VFS token for the resolved file is stored.

vntoken

The name of a variable in which the vnode token for the resolved file is stored.

stem

The name of a stem variable used to return the mount entry for the file system. To access mount table information, you can use a numeric value or the predefined variables beginning with MNTE_ used to derive the numeric value. See "getmntent" on page 76 for a description of the MNTE_ variables; see Appendix A, "REXX predefined variables," on page 241 for the numeric values.

stem2

Upon return, *stem2.0* contains the number of attribute variables returned. The same information is returned in *stem2* as if a *v_getattr* had been used on the file that was just resolved. You can use the predefined variables beginning with ST_ to access those respective values. For example, *stem2.st_size* accesses the file size. See "stat" on page 140 for a description of the ST_ variables.

Usage notes

1. The mount point pathname is not available in the MNTE_ structure returned by the variable *stem2.mnte_path*.
2. The caller is responsible for freeing vnode tokens returned by the *v_rpn* service, by calling to the *v_rel* service when they are no longer needed.

Example

In the following example, assume that *path* was assigned a value earlier in the exec:

```
"v_rpn (path) vfstok filetok mnt. attr."
```

v_setattr

```
▶—v_setattr—vntoken—attribute_list—◀
```

Function

v_setattr invokes the *v_setattr* callable service to set the attributes associated with the file represented by the vnode token. You can change the mode, owner, access time, modification time, change time, reference time, audit flags, general attribute flags, and file size.

Parameters

vntoken

A variable name that contains the vnode token for the file for which the attributes are to be set.

attribute_list

A list of attributes to be set and their values. The attributes are expressed either as numeric values or as the predefined variables beginning with ST_, followed by arguments for that attribute. For the predefined variables

beginning with `ST_`, see “chattr” on page 33; for the numeric values, see Appendix A, “REXX predefined variables,” on page 241.

Usage notes

- For usage notes, see “chattr” on page 33.
- The tagging of `/dev/null`, `/dev/zero`, `/dev/random`, and `/dev/urandom` is ignored.

Example

In the following example, assume that `vntok` was assigned a value earlier in the exec. This example truncates a file to 0 length and sets the mode to 600:

```
"v_setattr vntok" st_size 0 st_mode 600
```

v_setattro

►—v_setattro—vntoken—opentoken—attribute_list—◄◄

Function

`v_setattro` invokes the `v_setattr` callable service to set the attributes that are associated with the file represented by `opentoken`. You can change the mode, owner, access time, modification time, change time, reference time, audit flags, general attribute flags, and file size.

Parameters

vntoken

A variable name that contains the vnode token for the file for which the attributes are to be set.

opentoken

A variable name that contains the open token as returned from the `v_open`.

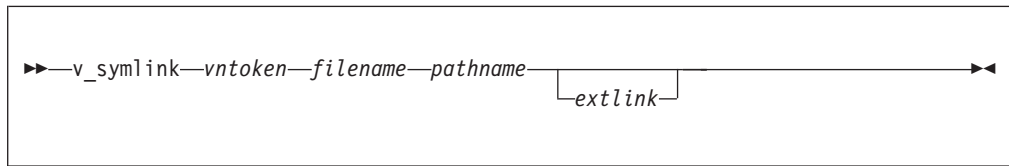
attribute_list

A list of attributes to be set and their values. The attributes are expressed either as numeric values or as the predefined variables beginning with `ST_`, followed by arguments for that attribute. For the predefined variables beginning with `ST_`, see “chattr” on page 33; for the numeric values, see Appendix A, “REXX predefined variables,” on page 241.

Example

To truncate a file represented by `vntok` with a `v_open` represented by `otok`:

```
"v_setattro vntok otok" st_size 0
```

v_symlink


Function

`v_symlink` invokes the `v_symlink` callable service to create a symbolic link to a pathname or external name. The contents of the symbolic link file is *pathname*.

Parameters

vtoken

A variable name for the directory that contains the vnode token in which *filename* is being created.

filename

The name for the symbolic link.

pathname

The absolute or relative pathname of the file you are linking to (the contents of the symbolic link).

extlink

Specify 1 if this is a symbolic link to an external name rather than to a pathname in the file hierarchy. An external name is the name of an object outside of the file hierarchy.

Usage notes

1. Like a hard link (described in “`v_link`” on page 219), a symbolic link allows a file to have more than one name. The presence of a hard link guarantees the existence of a file, even after the original name has been removed. A symbolic link, however, provides no such assurance; in fact, the file identified by *pathname* need not exist when the symbolic link is created. In addition, a symbolic link can cross file system boundaries, and it can refer to objects outside of a hierarchical file system.
2. When a component of a pathname refers to a symbolic link (but not an external symbolic link) rather than to a directory, the pathname contained in the symbolic link is resolved. For `v_rpn` or other z/OS UNIX callable services, a symbolic link in a pathname parameter is resolved as follows:
 - If the pathname in the symbolic link begins with / (slash), the symbolic link pathname is resolved relative to the process root directory.
 - If the pathname in the symbolic link does not begin with /, the symbolic link pathname is resolved relative to the directory that contains the symbolic link.
 - If the symbolic link is not the last component of the original pathname, remaining components of the original pathname are resolved from there.
 - When a symbolic link is the last component of a pathname, it may or may not be resolved. Resolution depends on the function using the pathname. For example, a rename request does not have a symbolic link resolved when it appears as the final component of either the new or old pathname. However, an open request does have a symbolic link resolved when it appears as the last component.

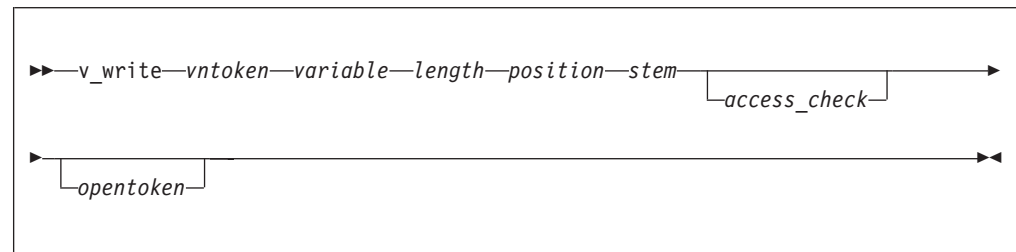
- When a slash is the last component of a pathname, and it is preceded by a symbolic link, the symbolic link is always resolved.
 - Because it cannot be changed, the mode of a symbolic link is ignored during the lookup process. Any files and directories to which a symbolic link refers are checked for access permission.
3. The external name contained in an external symbolic link is not resolved. The *filename* cannot be used as a directory component of a pathname.

Example

In the following example, assume that *dirtok*, *file*, and *linkname* were assigned values earlier in the exec:

```
"v_symlink dirtok (file) (linkname)"
```

v_write



Function

v_rdw invokes the **v_rdw** callable service to accept a vnode token representing a file and to write data to the file. The number of bytes written and the file attributes are returned in RETVAL when the write completes.

Parameters

vntoken

A variable name that contains the vnode token for the file to be written.

variable

The name of the buffer from which data is to be written.

length

The number of characters to write.

position

The file offset where the write is to start from, specified in bytes.

stem

The name of a stem variable used to return the file attributes. Upon return, *stem.0* contains the number of attribute variables returned. The same information is returned in *stem* as if a **v_getattr** was used on the file. To access the file attributes, you can use a numeric value or the predefined variables beginning with **ST_** used to derive the numeric value. See “stat” on page 140 for more information about the **ST_** predefined variables; see Appendix A, “REXX predefined variables,” on page 241 for the numeric values.

access_check

Specify 0 for no access check, or 1 for the system to check the user for read access to the file. The user is defined by the effective UID and GID. Setting the

v_write

effective GID does not affect supplemental groups. Also, there is no support for altering the MVS user identity of the task using the ACEE.

opentoken

A variable name that contains the open token as returned from the v_open service. If the opentoken parameter is specified, the access_check parameter must also be specified.

Example

In the following example, assume that *filetok*, *buf*, and *pos* were assigned values earlier in the exec:

```
"v_write filetok buf" length(buf) pos
```

Chapter 8. Examples: Using virtual file system syscall commands

These are examples of REXX programs that use the virtual file system syscall commands.

List the files in a directory

Given a directory path name, this example lists the files in the directory.

```
/* rexx */
parse arg dir                /* take directory path as argument */
if dir= ' ' then
do
  say 'directory argument required'
  return
end
call syscalls 'ON'
address syscall
'v_reg 1 dirlist'          /* register as a file server */
if retval=-1 then
do
  say 'error registering as a server - error codes:' errno errnojr
  return
end
'v_rpn (dir) vfs vn mnt. st.' /* resolve the directory path name */
if retval=-1 then
do
  say 'error resolving path' dir '- error codes:' errno errnojr
  return
end
i=1                          /* next dir entry to read is 1 */
do forever                   /* loop reading directory */
  'v_readdir vn d.' i       /* read starting at next entry */
  if retval=-1 then
  do
    say 'error reading directory - error codes:' errno errnojr
    leave
  end
  if d.0=0 then leave       /* if nothing returned then done */
  do j=1 to d.0             /* process each entry returned */
    say d.j
  end
  i=i+d.0                  /* set index to next entry */
end
'v_rel vn'                  /* release the directory vnode */
return
```

Remove a file or empty directory

Given a directory path name and the filename to delete, this example removes the file or an empty directory.

```
/* rexx */
if __argv.0<>3 then          /* check for right number of args */
do
  say 'directory and filename required'
  return
end
/* __argv.1 is program name */
```

```

dir=__argv.2                /* 1st arg is directory path name */
file=__argv.3              /* 2nd arg is file name */
call syscalls 'ON'
address syscall
"v_reg 1 'remove file'"    /* register as a file server */
if retval=-1 then
  do
    say 'error registering as a server - error codes:' errno errnojr
    return
  end
'v_rpn (dir) vfs vn mnt. st.' /* resolve the directory path name */
if retval=-1 then
  do
    say 'error resolving path' dir '- error codes:' errno errnojr
    return
  end
'v_lookup vn (file) fst. fvn' /* look up the file */
if retval=-1 then
  do
    say "error locating file" file
    say "      in directory" dir
    say "      error codes:" errno errnojr
  end
else
  do
    if fst.st_type=s_isdir then /* if the file is a directory */
      'v_rmdir vn (file)'      /* then delete it with v_rmdir */
    else
      'v_remove vn (file)'     /* else delete it with v_remove */
    if retval=-1 then
      say 'error deleting file - error codes:' errno errnojr
    'v_rel fvn'                /* release the file vnode */
  end
'v_rel vn'                    /* release the directory vnode */
return

```

Appendix A. REXX predefined variables

Predefined variables make symbolic references easier and more consistent. Instead of using a numeric value, you can use the predefined variable that will derive that numeric value. The following list shows the data type and numeric value for each predefined variable. Each variable is discussed in the section about the `syscall` command with which it can be used. Most variable names correspond to the POSIX-defined names in the XL C/C++ runtime library include files (also known as header files).

When the REXX program is initialized and the SYSCALL environment is established, the predefined variables are set up. If you call an internal subroutine that uses the PROCEDURE instruction to protect existing variables by making them unknown to that subroutine (or function), the predefined variables also become unknown. If some of the predefined variables are needed, you can either list them on the PROCEDURE EXPOSE instruction or issue another `syscalls('ON')` to reestablish the predefined variables. The predefined variables are automatically set up for external functions and subroutines. For example:

```
subroutine: procedure
junk = syscalls('ON')
parse arg dir
'readdir (dir) dir. stem.'
```

Except for the error numbers and signal numbers, all variables contain an underscore. This is also true of most of the POSIX names in the C include files.

The data types are as follows:

Bin Binary: 2-byte hexadecimal format
Char Character
Dec Decimal
Hex Hexadecimal: 4-byte hexadecimal format
Tok Token
Oct Octal

Table 12 lists the predefined variables along with their data type and numeric value.

Table 12. List of predefined variables

Variable	Data type	Numeric value
AUD_FEXEC	Dec	512
AUD_FREAD	Dec	33554432
AUD_FWRITE	Dec	131072
AUD_SEXEC	Dec	256
AUD_SREAD	Dec	16777216
AUD_SWRITE	Dec	65536
E2BIG	Hex	91
EACCES	Hex	6F
EAGAIN	Hex	70
EBADF	Hex	71

Table 12. List of predefined variables (continued)

Variable	Data type	Numeric value
EBUSY	Hex	72
ECHILD	Hex	73
EDEADLK	Hex	74
EDOM	Hex	01
EEXIST	Hex	75
EFAULT	Hex	76
EFBIG	Hex	77
EILSEQ	Hex	93
EINTR	Hex	78
EINVAL	Hex	79
EIO	Hex	7A
EISDIR	Hex	7B
ELOOP	Hex	92
EMFILE	Hex	7C
EMLINK	Hex	7D
EMVSBADCHAR	Hex	A0
EMVSCATLG	Hex	99
EMVSCVAF	Hex	98
EMVSDYNALC	Hex	97
EMVSERR	Hex	9D
EMVSINITIAL	Hex	9C
EMVSNORTL	Hex	A7
EMVSNOTUP	Hex	96
EMVSPARM	Hex	9E
EMVSPATHOPTS	Hex	A6
EMVSPFSFILE	Hex	9F
EMVSPFSPERM	Hex	A2
EMVSSAF2ERR	Hex	A4
EMVSSAFEXTRERR	Hex	A3
EMVSTODNOTSET	Hex	A5
ENAMETOOLONG	Hex	7E
ENFILE	Hex	7F
ENODEV	Hex	80
ENOENT	Hex	81
ENOEXEC	Hex	82
ENOLCK	Hex	83
ENOMEM	Hex	84
ENOSPC	Hex	85
ENOSYS	Hex	86
ENOTDIR	Hex	87

Table 12. List of predefined variables (continued)

Variable	Data type	Numeric value
ENOTEMPTY	Hex	88
ENOTTY	Hex	89
ENXIO	Hex	8A
EPERM	Hex	8B
EPIPE	Hex	8C
ERANGE	Hex	02
EROFS	Hex	8D
ESC_A	Hex	2F
ESC_B	Hex	16
ESC_F	Hex	0C
ESC_N	Hex	15
ESC_R	Hex	0D
ESC_T	Hex	05
ESC_V	Hex	0B
ESPIPE	Hex	8E
ESRCH	Hex	8F
EXDEV	Hex	90
F_OK	Dec	8
F_RDLCK	Dec	1
F_UNLCK	Dec	3
F_WRLCK	Dec	2
GR_GID	Dec	2
GR_MEM	Char	4
GR_MEMBERS	Dec	3
GR_NAME	Char	1
IPC_CREAT	Hex	1
IPC_EXCL	Hex	2
IPC_MEGA	Hex	4
IPC_SHAREAS	Hex	32
IPC_RCVTYPEPID	Hex	4
IPC_SNDTYPEPID	Hex	8
L_LEN	Dec	4
L_PID	Dec	5
L_START	Dec	3
L_TYPE	Dec	1
L_WHENCE	Dec	2
MNT_ASYNCMOUNT	Dec	130
MNT_FILEACTIVE	Dec	0
MNT_FILEDEAD	Dec	1
MNT_FILEDRAIN	Dec	4

Table 12. List of predefined variables (continued)

Variable	Data type	Numeric value
MNT_FILEFORCE	Dec	8
MNT_FILEIMMED	Dec	16
MNT_FILENORM	Dec	32
MNT_FILERESET	Dec	2
MNT_IMMEDTRIED	Dec	64
MNT_MODE_AUNMOUNT	Dec	64
MNT_MODE_CLIENT	Dec	32
MNT_MODE_EXPORT	Dec	4
MNT_MODE_NOAUTOMOVE	Dec	16
MNT_MODE_NOSEC	Dec	8
MNT_MODE_NOSETID	Dec	2
MNT_MODE_RDONLY	Dec	1
MNT_MODE_RDWR	Dec	0
MNT_MODE_SECACL	Dec	128
MNT_MOUNTINPROGRESS	Dec	129
MNT_QUIESCED	Dec	128
MNTE_AGGNAME	Char	30
MNTE_BYTESREADHW	Dec	25
MNTE_BYTESREADLW	Dec	26
MNTE_BYTESWRITTENHW	Dec	27
MNTE_BYTESWRITTENLW	Dec	28
MNTE_DD	Char	4
MNTE_DEV	Dec	3
MNTE_DIRIBC	Dec	22
MNTE_FILETAG	Char	29
MNTE_FROMSYS	Char	18
MNTE_FSNAME	Char	6
MNTE_FSTYPE	Char	5
MNTE_MODE	Dec	2
MNTE_PARDEV	Dec	9
MNTE_PARM	Char	13
MNTE_PATH	Char	7
MNTE_PFSSTATUSXCP	Char	34
MNTE_PFSSTATUSNORMAL	Char	33
MNTE_QJOBNAME	Char	11
MNTE_QPID	Dec	12
MNTE_QSYSNAME	Char	16
MNTE_READCT	Dec	20
MNTE_READIBC	Dec	23
MNTE_RFLAGS	Dec	14

Table 12. List of predefined variables (continued)

Variable	Data type	Numeric value
MNTE_ROOTINO	Dec	10
MNTE_ROSECLABEL	Char	32
MNTE_STATUS	Dec	8
MNTE_STATUS2	Dec	15
MNTE_SUCCESS	Dec	19
MNTE_SYSLIST	Char	31
MNTE_SYSNAME	Char	17
MNTE_TYPE	Dec	1
MNTE_WRITECT	Dec	21
MNTE_WRITEIBC	Dec	24
MSG_UID	Dec	1
MSG_GID	Dec	2
MSG_CUID	Dec	3
MSG_CGID	Dec	4
MSG_TYPE	Dec	5
MSG_MODE	Oct	6
MSG_QNUM	Dec	7
MSG_QBYTES	Dec	8
MSG_LSPID	Dec	9
MSG_LRPID	Dec	10
MSG_STIME	Dec	11
MSG_RTIME	Dec	12
MSG_CTIME	Dec	13
MSGBUF_MTIME	Dec	1
MSGBUF_UID	Dec	2
MSGBUF_GID	Dec	3
MSGBUF_PID	Dec	4
MTM_DRAIN	Dec	2
MTM_FORCE	Dec	4
MTM_IMMED	Dec	8
MTM_NORMAL	Dec	16
MTM_NOSUID	Dec	262144
MTM_RDONLY	Dec	128
MTM_RDWR	Dec	64
MTM_REMOUNT	Dec	65536
MTM_RESET	Dec	1
MTM_SYNCHONLY	Dec	131072
O_APPEND	Dec	8
O_CREAT	Dec	128
O_EXCL	Dec	64

Table 12. List of predefined variables (continued)

Variable	Data type	Numeric value
O_NOCITY	Dec	32
O_NONBLOCK	Dec	4
O_RDONLY	Dec	2
O_RDWR	Dec	3
O_SYNC	Dec	256
O_TRUNC	Dec	16
O_WRONLY	Dec	1
PC_ACL	Dec	10
PC_ACL_MAX	Dec	11
PC_LINK_MAX	Dec	2
PC_MAX_CANON	Dec	3
PC_MAX_INPUT	Dec	4
PC_NAME_MAX	Dec	5
PC_PATH_MAX	Dec	7
PC_PIPE_BUF	Dec	8
PC_POSIX_CHOWN_RESTRICTED	Dec	1
PC_POSIX_NO_TRUNC	Dec	6
PC_POSIX_VDISABLE	Dec	9
PS_CHILD	Char	W
PS_CMD	Dec	19
PS_CONTTY	Dec	17
PS_EGID	Dec	10
PS_EUID	Dec	7
PS_FGPID	Dec	6
PS_FORK	Char	X
PS_FREEZE	Char	E
PS_MAXVNODES	Dec	25
PS_MSGRCV	Char	A
PS_MSGSND	Char	B
PS_PATH	Char	18
PS_PAUSE	Char	G
PS_PGPID	Dec	5
PS_PID	Dec	2
PS_PPID	Dec	3
PS_QUIESCE	Char	Q
PS_RGID	Dec	11
PS_RUID	Dec	8
PS_RUN	Char	R
PS_SEMWT	Char	D
PS_SERVERFLAGS	Dec	27

Table 12. List of predefined variables (continued)

Variable	Data type	Numeric value
PS_SERVERNAME	Dec	22
PS_SERVERTYPE	Dec	21
PS_SGID	Dec	12
PS_SID	Dec	4
PS_SIZE	Dec	13
PS_SLEEP	Char	S
PS_STARTTIME	Dec	14
PS_STAT	Dec	1
PS_STATE	Char	20
PS_SUID	Dec	9
PS_SYSTIME	Dec	16
PS_USERTIME	Dec	15
PS_VNODECOUNT	Dec	26
PS_WAITC	Char	C
PS_WAITF	Char	F
PS_WAITO	Char	K
PS_ZOMBIE	Char	Z
PS_ZOMBIE2	Char	L
PW_DIR	Char	4
PW_GID	Dec	3
PW_NAME	Char	1
PW_SHELL	Char	5
PW_UID	Dec	2
R_OK	Dec	4
RLIMIT_AS	Dec	5
RLIMIT_CORE	Dec	4
RLIMIT_CPU	Dec	0
RLIMIT_FSIZE	Dec	1
RLIMIT_INFINITY	Dec	2147483647
RLIMIT_NOFILE	Dec	6
S_FFBINARY	Dec	1
S_FFCR	Dec	3
S_FFCRLF	Dec	5
S_FFCRNL	Dec	7
S_FFLF	Dec	4
S_FFLFCR	Dec	6
S_FFNA	Dec	0
S_FFNL	Dec	2
S_FFRECORD	Dec	8
S_ISCHR	Dec	2

Table 12. List of predefined variables (continued)

Variable	Data type	Numeric value
S_ISDIR	Dec	1
S_ISFIFO	Dec	4
S_ISREG	Dec	3
S_ISSYM	Dec	5
SA_NOCLDSTOP	Dec	32768
SA_NOCLDWAIT	Dec	512
SA_NORESETHAND	Dec	4096
SC_2_CHAR_TERM	Dec	12
SC_ARG_MAX	Dec	1
SC_CHILD_MAX	Dec	2
SC_CLK_TCK	Dec	3
SC_JOB_CONTROL	Dec	4
SC_NGROUPS_MAX	Dec	5
SC_OPEN_MAX	Dec	6
SC_SAVED_IDS	Dec	7
SC_TREAD_TASKS_MAX_NP	Dec	11
SC_THREADS_MAP_NP	Dec	13
SC_TZNAME_MAX	Dec	9
SC_VERSION	Dec	10
SE_ERRNO	Dec	1
SE_REASON	Dec	2
SE_ACTION	Dec	3
SE_MODID	Dec	4
SEEK_CUR	Dec	1
SEEK_END	Dec	2
SEEK_SET	Dec	0
SHM_UID	Hex	1
SHM_GID	Hex	2
SHM_CUID	Hex	3
SHM_CGID	Hex	4
SHM_TYPE	Hex	5
SHM_MODE	Oct	6
SHM_SEGSZ	Hex	7
SHM_LPID	Hex	8
SHM_CPID	Hex	9
SHM_ATIME	Hex	10
SHM_DTIME	Hex	11
SHM_CTIME	Hex	12
SIG_BLOCK	Dec	0
SIG_CAT	Dec	10

Table 12. List of predefined variables (continued)

Variable	Data type	Numeric value
SIG_DFL	Dec	0
SIG_IGN	Dec	1
SIG_QRY	Dec	11
SIG_SETMASK	Dec	2
SIG_UNBLOCK	Dec	1
SIGABND	Dec	18
SIGABRT	Dec	3
SIGALRM	Dec	14
SIGBUS	Dec	10
SIGCHLD	Dec	20
SIGCONT	Dec	19
SIGDANGER	Dec	33
SIGFPE	Dec	8
SIGHUP	Dec	1
SIGILL	Dec	4
SIGINT	Dec	2
SIGIO	Dec	23
SIGIOERR	Dec	27
SIGKILL	Dec	9
SIGPIPE	Dec	13
SIGPOLL	Dec	5
SIGPROF	Dec	32
SIGQUIT	Dec	24
SIGSEGV	Dec	11
SIGSTOP	Dec	7
SIGSYS	Dec	12
SIGTERM	Dec	15
SIGTRAP	Dec	26
SIGTSTP	Dec	25
SIGTTIN	Dec	21
SIGTTOU	Dec	22
SIGUSR1	Dec	16
SIGUSR2	Dec	17
SIGURG	Dec	6
SIGVTALRM	Dec	31
SIGXCPU	Dec	29
SIGXFSZ	Dec	30
ST_AAUDIT	Dec	16
ST_ACCESSACL	Dec	29
ST_ETIME	Dec	9

Table 12. List of predefined variables (continued)

Variable	Data type	Numeric value
ST_AUDITID	Char	20
ST_BLKSIZE	Dec	18
ST_BLOCKS	Dec	22
ST_CCSID	Char	21
ST_CRTIME	Dec	19
ST_CTIME	Dec	11
ST_DEV	Hex	4
ST_DMODELACL	Dec	31
ST_EXTLINK	Dec	24
ST_FID	Bin	27
ST_FILEFMT	Dec	28
ST_FMODELACL	Dec	30
ST_GENVALUE	Bin	25
ST_GID	Dec	7
ST_INO	Hex	3
ST_MAJOR	Dec	14
ST_MINOR	Dec	15
ST_MODE	Oct	2
ST_MTIME	Dec	10
ST_NLINK	Dec	5
ST_RTIME	Dec	26
ST_SECLABEL	Dec	32
ST_SETGID	Dec	13
ST_SETUID	Dec	12
ST_SIZE	Dec	8
ST_STICKY	Dec	23
ST_TYPE	Dec	1
ST_UAUDIT	Dec	17
ST_UID	Dec	6
STFS_AVAIL	Dec	4
STFS_BFREE	Dec	6
STFS_BLOCKSIZE	Dec	1
STFS_FAVAIL	Dec	9
STFS_FFREF	Dec	8
STFS_FILES	Dec	7
STFS_FRSIZE	Dec	5
STFS_FSID	Dec	10
STFS_INUSE	Dec	3
STFS_INVARSEC	Dec	15
STFS_NAMEMAX	Dec	13

Table 12. List of predefined variables (continued)

Variable	Data type	Numeric value
STFS_NOSEC	Dec	16
STFS_NOSUID	Dec	12
STFS_RDONLY	Dec	11
STFS_TOTAL	Dec	2
TM_HOUR	Dec	1
TM_ISDST	Dec	9
TM_MDAY	Dec	5
TM_MIN	Dec	2
TM_MON	Dec	4
TM_SEC	Dec	3
TM_WDAY	Dec	8
TM_YDAY	Dec	7
TM_YEAR	Dec	6
TMS_CSTIME	Dec	4
TMS_CUTIME	Dec	3
TMS_STIME	Dec	2
TMS_UTIME	Dec	1
U_MACHINE	Char	5
U_NODENAME	Char	2
U_RELEASE	Char	3
U_SYSNAME	Char	1
U_VERSION	Char	4
VL_CLIENTPID	Dec	7
VL_CLIENTTID	Char	9
VL_DOSMODE	Char	13
VL_DOSACCESS	Char	14
VL_LOCK	Dec	3
VL_LOCKERTOK	Tok	8
VL_LOCKWAIT	Dec	4
VL_OBJCLASS	Char	10
VL_OBJID	Char	11
VL_OBJTOK	Tok	12
VL_PURGE	Dec	7
VL_QUERY	Dec	6
VL_REGLOCKER	Dec	1
VL_SERVERPID	Dec	6
VL_UNREGLOCKER	Dec	2
VL_UNLOCK	Dec	5
W_CONTINUED	Dec	3
W_EXITSTATUS	Dec	4

Table 12. List of predefined variables (continued)

Variable	Data type	Numeric value
W_IFEXITED	Dec	3
W_IFSIGNALLED	Dec	5
W_IFSTOPPED	Dec	7
W_NOHANG	Dec	1
W_OK	Dec	2
W_STAT3	Dec	1
W_STAT4	Dec	2
W_STOPSIG	Dec	8
W_TERMSIG	Dec	6
W_UNTRACED	Dec	2
X_OK	Dec	1

Appendix B. Setting permissions for files and directories

Typically, octal permissions are specified with three or four numbers, in these positions:

1234

Each position indicates a different type of access:

- Position 1 is the bit that sets permission for set-user-ID on access, set-group-ID on access, or the *sticky bit*. Specifying this bit is optional.
- Position 2 is the bit that sets permissions for the owner of the file. Specifying this bit is required.
- Position 3 is the bit that sets permissions for the group that the owner belongs to. Specifying this bit is required.
- Position 4 is the bit that sets permissions for others. Specifying this bit is required.

Position 1

Specifying the bit in position 1 is optional. You can specify one of these values:

0	Off
1	Sticky bit on
2	Set-group-ID-on execution
3	Set-group-ID-on execution and set the sticky bit on
4	Set-user-ID on execution
5	Set-user-ID on execution and set the sticky bit on.
6	Set-user-ID and set-group-ID on execution
7	Set-user-ID and set-group-ID on execution and set the sticky bit on

Positions 2, 3, and 4

Specifying these bits is required. For each type of access—owner, group, and other—there is a corresponding octal number:

0	No access (---)
1	Execute-only access (--x)
2	Write-only access (-w-)
3	Write and execute access (-wx)
4	Read-only access (r--)
5	Read and execute access (r-x)
6	Read and write access (rw-)
7	Read, write, and execute access (rwx)

To specify permissions for a file or directory, you use at least a *3-digit* octal number, omitting the digit in the first position. When you specify just three digits,

the first digit describes owner permissions, the second digit describes group permissions, and the third digit describes permissions for all others. When the first digit is not set, some typical 3-digit permissions are specified in octal as shown in Figure 4.

Octal Number		Meaning
666	<pre> 6 6 6 / \ rw- rw- rw- </pre>	owner (rw-) group (rw-) other (rw-)
700	<pre> 7 0 0 / \ rwx --- --- </pre>	owner (rwx) group (---) other (---)
755	<pre> 7 5 5 / \ rwx r-x r-x </pre>	owner (rwx) group (r-x) other (r-x)
777	<pre> 7 7 7 / \ rwx rwx rwx </pre>	owner (rwx) group (rwx) other (rwx)

Figure 4. Three-digit permissions specified in octal format

Example: using BITOR and BITAND to set mode bits

To set a file's mode bits, use the REXX functions BITOR() and BITAND() with the octal numbers.

For example, if you have obtained a file's permission bits and want to use **chmod** to turn on the write bits, you could code:

```
'chmod (file)' BITOR(st.st_mode, 222)
```

To turn the same bits off, you could code:

```
'chmod (file)' BITAND(st.st_mode, 555)
```

Appendix C. Accessibility

Accessible publications for this product are offered through the z/OS Information Center, which is available at www.ibm.com/systems/z/os/zos/bkserv/.

If you experience difficulty with the accessibility of any z/OS information, please send a detailed message to mhvrcfs@us.ibm.com or to the following mailing address:

IBM Corporation
Attention: MHVRCFS Reader Comments
Department H6MA, Building 707
2455 South Road
Poughkeepsie, NY 12601-5400
USA

Accessibility features

Accessibility features help a user who has a physical disability, such as restricted mobility or limited vision, to use software products successfully. The major accessibility features in z/OS enable users to:

- Use assistive technologies such as screen readers and screen magnifier software
- Operate specific or equivalent features using only the keyboard
- Customize display attributes such as color, contrast, and font size.

Using assistive technologies

Assistive technology products, such as screen readers, function with the user interfaces found in z/OS. Consult the assistive technology documentation for specific information when using such products to access z/OS interfaces.

Keyboard navigation of the user interface

Users can access z/OS user interfaces using TSO/E or ISPF. Refer to *z/OS TSO/E Primer*, *z/OS TSO/E User's Guide*, and *z/OS ISPF User's Guide Vol I* for information about accessing TSO/E and ISPF interfaces. These guides describe how to use TSO/E and ISPF, including the use of keyboard shortcuts or function keys (PF keys). Each guide includes the default settings for the PF keys and explains how to modify their functions.

Dotted decimal syntax diagrams

Syntax diagrams are provided in dotted decimal format for users accessing the z/OS Information Center using a screen reader. In dotted decimal format, each syntax element is written on a separate line. If two or more syntax elements are always present together (or always absent together), they can appear on the same line, because they can be considered as a single compound syntax element.

Each line starts with a dotted decimal number; for example, 3 or 3.1 or 3.1.1. To hear these numbers correctly, make sure that your screen reader is set to read out punctuation. All the syntax elements that have the same dotted decimal number (for example, all the syntax elements that have the number 3.1) are mutually

exclusive alternatives. If you hear the lines 3.1 USERID and 3.1 SYSTEMID, you know that your syntax can include either USERID or SYSTEMID, but not both.

The dotted decimal numbering level denotes the level of nesting. For example, if a syntax element with dotted decimal number 3 is followed by a series of syntax elements with dotted decimal number 3.1, all the syntax elements numbered 3.1 are subordinate to the syntax element numbered 3.

Certain words and symbols are used next to the dotted decimal numbers to add information about the syntax elements. Occasionally, these words and symbols might occur at the beginning of the element itself. For ease of identification, if the word or symbol is a part of the syntax element, it is preceded by the backslash (\) character. The * symbol can be used next to a dotted decimal number to indicate that the syntax element repeats. For example, syntax element *FILE with dotted decimal number 3 is given the format 3 * FILE. Format 3* FILE indicates that syntax element FILE repeats. Format 3* * FILE indicates that syntax element * FILE repeats.

Characters such as commas, which are used to separate a string of syntax elements, are shown in the syntax just before the items they separate. These characters can appear on the same line as each item, or on a separate line with the same dotted decimal number as the relevant items. The line can also show another symbol giving information about the syntax elements. For example, the lines 5.1*, 5.1 LASTRUN, and 5.1 DELETE mean that if you use more than one of the LASTRUN and DELETE syntax elements, the elements must be separated by a comma. If no separator is given, assume that you use a blank to separate each syntax element.

If a syntax element is preceded by the % symbol, this indicates a reference that is defined elsewhere. The string following the % symbol is the name of a syntax fragment rather than a literal. For example, the line 2.1 %OP1 means that you should refer to separate syntax fragment OP1.

The following words and symbols are used next to the dotted decimal numbers:

- ? means an optional syntax element. A dotted decimal number followed by the ? symbol indicates that all the syntax elements with a corresponding dotted decimal number, and any subordinate syntax elements, are optional. If there is only one syntax element with a dotted decimal number, the ? symbol is displayed on the same line as the syntax element, (for example 5? NOTIFY). If there is more than one syntax element with a dotted decimal number, the ? symbol is displayed on a line by itself, followed by the syntax elements that are optional. For example, if you hear the lines 5 ?, 5 NOTIFY, and 5 UPDATE, you know that syntax elements NOTIFY and UPDATE are optional; that is, you can choose one or none of them. The ? symbol is equivalent to a bypass line in a railroad diagram.
- ! means a default syntax element. A dotted decimal number followed by the ! symbol and a syntax element indicates that the syntax element is the default option for all syntax elements that share the same dotted decimal number. Only one of the syntax elements that share the same dotted decimal number can specify a ! symbol. For example, if you hear the lines 2? FILE, 2.1! (KEEP), and 2.1 (DELETE), you know that (KEEP) is the default option for the FILE keyword. In this example, if you include the FILE keyword but do not specify an option, default option KEEP will be applied. A default option also applies to the next higher dotted decimal number. In this example, if the FILE keyword is omitted, default FILE(KEEP) is used. However, if you hear the lines 2? FILE, 2.1, 2.1.1!

(KEEP), and 2.1.1 (DELETE), the default option KEEP only applies to the next higher dotted decimal number, 2.1 (which does not have an associated keyword), and does not apply to 2? FILE. Nothing is used if the keyword FILE is omitted.

- * means a syntax element that can be repeated 0 or more times. A dotted decimal number followed by the * symbol indicates that this syntax element can be used zero or more times; that is, it is optional and can be repeated. For example, if you hear the line 5.1* data area, you know that you can include one data area, more than one data area, or no data area. If you hear the lines 3*, 3 HOST, and 3 STATE, you know that you can include HOST, STATE, both together, or nothing.

Note:

1. If a dotted decimal number has an asterisk (*) next to it and there is only one item with that dotted decimal number, you can repeat that same item more than once.
 2. If a dotted decimal number has an asterisk next to it and several items have that dotted decimal number, you can use more than one item from the list, but you cannot use the items more than once each. In the previous example, you could write HOST STATE, but you could not write HOST HOST.
 3. The * symbol is equivalent to a loop-back line in a railroad syntax diagram.
- + means a syntax element that must be included one or more times. A dotted decimal number followed by the + symbol indicates that this syntax element must be included one or more times; that is, it must be included at least once and can be repeated. For example, if you hear the line 6.1+ data area, you must include at least one data area. If you hear the lines 2+, 2 HOST, and 2 STATE, you know that you must include HOST, STATE, or both. Similar to the * symbol, the + symbol can only repeat a particular item if it is the only item with that dotted decimal number. The + symbol, like the * symbol, is equivalent to a loop-back line in a railroad syntax diagram.

Notices

This information was developed for products and services offered in the U.S.A. or elsewhere.

IBM may not offer the products, services, or features discussed in this document in other countries. Consult your local IBM representative for information on the products and services currently available in your area. Any reference to an IBM product, program, or service is not intended to state or imply that only that IBM product, program, or service may be used. Any functionally equivalent product, program, or service that does not infringe any IBM intellectual property right may be used instead. However, it is the user's responsibility to evaluate and verify the operation of any non-IBM product, program, or service.

IBM may have patents or pending patent applications covering subject matter described in this document. The furnishing of this document does not give you any license to these patents. You can send license inquiries, in writing, to:

IBM Director of Licensing
IBM Corporation
North Castle Drive
Armonk, NY 10504-1785
U.S.A

For license inquiries regarding double-byte character set (DBCS) information, contact the IBM Intellectual Property Department in your country or send inquiries, in writing, to:

Intellectual Property Licensing
Legal and Intellectual Property Law
IBM Japan, Ltd.
19-21, Nihonbashi-Hakozakicho, Chuo-ku
Tokyo 103-8510, Japan

The following paragraph does not apply to the United Kingdom or any other country where such provisions are inconsistent with local law: INTERNATIONAL BUSINESS MACHINES CORPORATION PROVIDES THIS PUBLICATION "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF NON-INFRINGEMENT, MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. Some states do not allow disclaimer of express or implied warranties in certain transactions, therefore, this statement may not apply to you.

This information could include technical inaccuracies or typographical errors. Changes are periodically made to the information herein; these changes will be incorporated in new editions of the publication. IBM may make improvements and/or changes in the product(s) and/or the program(s) described in this publication at any time without notice.

Any references in this information to non-IBM Web sites are provided for convenience only and do not in any manner serve as an endorsement of those Web sites. The materials at those Web sites are not part of the materials for this IBM product and use of those Web sites is at your own risk.

IBM may use or distribute any of the information you supply in any way it believes appropriate without incurring any obligation to you.

Licensees of this program who wish to have information about it for the purpose of enabling: (i) the exchange of information between independently created programs and other programs (including this one) and (ii) the mutual use of the information which has been exchanged, should contact:

Site Counsel
IBM Corporation
2455 South Road
Poughkeepsie, NY 12601-5400
USA

Such information may be available, subject to appropriate terms and conditions, including in some cases, payment of a fee.

The licensed program described in this information and all licensed material available for it are provided by IBM under terms of the IBM Customer Agreement, IBM International Program License Agreement, or any equivalent agreement between us.

Information concerning non-IBM products was obtained from the suppliers of those products, their published announcements or other publicly available sources. IBM has not tested those products and cannot confirm the accuracy of performance, compatibility or any other claims related to non-IBM products. Questions on the capabilities of non-IBM products should be addressed to the suppliers of those products.

All statements regarding IBM's future direction or intent are subject to change or withdrawal without notice, and represent goals and objectives only.

If you are viewing this information softcopy, the photographs and color illustrations may not appear.

COPYRIGHT LICENSE:

This information might contain sample application programs in source language, which illustrate programming techniques on various operating platforms. You may copy, modify, and distribute these sample programs in any form without payment to IBM, for the purposes of developing, using, marketing or distributing application programs conforming to the application programming interface for the operating platform for which the sample programs are written. These examples have not been thoroughly tested under all conditions. IBM, therefore, cannot guarantee or imply reliability, serviceability, or function of these programs. The sample programs are provided "AS IS", without warranty of any kind. IBM shall not be liable for any damages arising out of your use of the sample programs.

Policy for unsupported hardware

Various z/OS elements, such as DFSMS, HCD, JES2, JES3, and MVS, contain code that supports specific hardware servers or devices. In some cases, this device-related element support remains in the product even after the hardware devices pass their announced End of Service date. z/OS may continue to service element code; however, it will not provide service related to unsupported hardware devices. Software problems related to these devices will not be accepted

for service, and current service activity will cease if a problem is determined to be associated with out-of-support devices. In such cases, fixes will not be issued.

Minimum supported hardware

The minimum supported hardware for z/OS releases identified in z/OS announcements can subsequently change when service for particular servers or devices is withdrawn. Likewise, the levels of other software products supported on a particular release of z/OS are subject to the service support lifecycle of those products. Therefore, z/OS and its product publications (for example, panels, samples, messages, and product documentation) can include references to hardware and software that is no longer supported.

- For information about software support lifecycle, see: IBM Lifecycle Support for z/OS (<http://www.ibm.com/software/support/systemsz/lifecycle/>)
- For information about currently-supported IBM hardware, contact your IBM representative.

Trademarks

IBM, the IBM logo, and [ibm.com](http://www.ibm.com)[®] are trademarks or registered trademarks of International Business Machines Corporation in the United States, other countries, or both. If these and other IBM trademarked terms are marked on their first occurrence in this information with a trademark symbol ([®] or [™]), these symbols indicate U.S. registered or common law trademarks owned by IBM at the time this information was published. Such trademarks may also be registered or common law trademarks in other countries. A current list of IBM trademarks is available on the Web at www.ibm.com/legal/copytrade.shtml (<http://www.ibm.com/legal/copytrade.shtml>).

UNIX is a registered trademark of The Open Group in the United States and other countries.

Acknowledgments

InterOpen/POSIX Shell and Utilities is a source code product providing POSIX.2 (Shell and Utilities) functions to z/OS UNIX System Services. InterOpen/POSIX Shell and Utilities is developed and licensed by Mortice Kern Systems (MKS) Inc. of Waterloo, Ontario, Canada.

Index

Numerics

3270 data stream 108
3270 passthrough mode 108

A

access syscall command 22
access time, set 154
accessibility 255
 contact IBM 255
 features 255
ACL
 deleting 23
 deleting a specific entry 24
 obtain resources 27
 read 25
 read entry 26
 release resources 25
 replace 28
 update entry 28
ACL_DELETE 24, 26, 29
ACL_ENTRY_GROUP 24, 26, 29
ACL_ENTRY_TYPE 24, 26, 29
ACL_ENTRY_USER 24, 26, 29
ACL_EXECUTE 24, 26, 29
ACL_ID 24, 26, 29
ACL_READ 24, 26, 29
ACL_TYPE_ACCESS 23, 26, 28
ACL_TYPE_DIRDEFAULT 23, 26, 28
ACL_TYPE_FILEDEFAULT 23, 26, 28
ACL_WRITE 24, 26, 29
acdelete syscall command 23, 24
acfree syscall command 25
aciget syscall command 25
acigetentry syscall command 26
aclinit syscall command 27
aclset syscall command 27
acupdateentry syscall command 28
ADDRESS TSO 5
alarm syscall command 29
appropriate privileges 14
assistive technologies 255
AUD_FEXEC 36, 47
AUD_FREAD 36, 47
AUD_FWRITE 36, 47
AUD_SEXEC 36, 47
AUD_SREAD 36, 47
AUD_SWRITE 36, 47
audit flags 36, 46
authorization 14
automatic file conversion
 control 50

B

batch, running a REXX program 2
BITAND 254
BITOR 254
BPX.SUPERUSER FACILITY 14
BPXWDYN 197

BPXWDYN (*continued*)
 calling conventions 197
 conventional MVS variable-length
 parameter string 197
 error codes for dynamic
 allocation 201
 error codes for dynamic output 202
 examples 214
 freeing an output descriptor 214
 key errors 201
 keywords 200
 message processing 202
 null-terminated parameter string 199
 request types 200
 requesting allocation information 208
 requesting dynamic allocation 202
 requesting dynamic
 concatenation 207
 requesting dynamic output 212
 requesting dynamic unallocation 207
 return codes 201
 REXX external function parameter
 list 197
BPXWRBLD module 15
bpxwunix() function 174

C

calling process
 effective group ID 118, 119
 effective user ID 70, 118
 making it a session leader 125
 parent process ID 79
 process group ID 79
 process ID 79
 real group ID 119
 real user ID 85, 126
 saved set group ID 119
 saved set user ID 126
calling process, effective group ID 70
calling process, real group ID 70
calling process, supplementary group
 ID 73
calling thread
 signal mask 134
 suspending 106, 136
catclose syscall command 30
catgets syscall command 31
catopen syscall command 31
cert syscall command 32
certificate registration 32
character set portable filename 19
character special file 94
charin() function 176
charout() function 177
chars() function 178
chattr syscall command 33
chaudit syscall command 36
chdir syscall command 37
child process 137, 140
chmod syscall command 37

chmod() function 178
chown syscall command 39
close syscall command 40
closedir syscall command 40
coding examples 161, 239
command
 syntax diagrams xi
command output
 trapping 174, 184
command, shell, su 14
commands
 stream() 191
configurable limit value 57, 105, 146
convd2e() function 179
conventional MVS parameter list 198
conventional MVS variable-length
 parameter string 197
creat syscall command 41
current directory
 returning or changing 180
customization
 host command environment 13
 performance 13
 REXX environment 13
CVT_QUERYCVT 50
CVT_SETAUTOCVTON 50
CVT_SETCVTOFF 50
CVT_SETCVTON 50

D

database, group
 retrieving an entry 71
 rewinding 120
database, user
 entry retrieval 82
 rewinding the 121
delete a directory
 rmdir 117
delete a file 151
 unlink 151
directory
 changing group 39
 changing mode 37
 changing owner 39
 closing 40
 creating 93
 entry 151
 file 151
 group 48
 mode, changing 47
 name, changing 116
 owner 48, 89
 reading multiple entries 110, 113
 removing 117
 rewind 117
 stream 104
directory() function 180
dup syscall command 42
dup2 syscall command 42
dynamic allocation 197, 201

dynamic output 197, 202

E

effective group ID calling process 70, 118, 119
effective user ID calling process 70, 118, 126
entry, directory 151
environment
 REXX 15
 SH 4
 SYSCALL 2
environment variable
 altering 180
 querying 180
environment() function 180
Epoch, seconds since 147
ERRNO 22
ERRNOJR 22
error codes 202
 for dynamic allocation 201
error codes for 201, 202
error codes, retrieving text 144
ESC_A 157
ESC_B 157
ESC_F 157
ESC_N 157
ESC_R 158
ESC_T 158
ESC_V 158
examples, coding 161, 239
EXECIO 8, 16
exists() function 181
exit status from REXX 9
external functions 5
extlink syscall command 43

F

f_closfd syscall command 49
f_control_cvt syscall command 50
f_dupfd syscall command 52
f_dupfd2 syscall command 52
f_getfd syscall command 53
f_getfl syscall command 53
f_getlk syscall command 54
F_OK 22
F_RDLCK 56, 61, 63, 222
f_setfd syscall command 59
f_setfl syscall command 59
f_setlk syscall command 60
f_setlkw syscall command 62
f_settag syscall command 64
F_UNLCK 56, 61, 63, 222
F_WRLCK 56, 61, 63, 222
fchattr syscall command 43
fchaudit syscall command 46
fchmod syscall command 47
fchown syscall command 48
fcntl syscall command 52
FD_CLOEXEC 59, 108
FID 215
FIFO special file 93
file
 access and modification times 154

file (*continued*)

 automatic conversion 50
 changes on DASD 68
 changing attributes 33, 43
 changing group 39
 changing mode 37, 179
 changing owner 39, 89
 changing size 69
 closing 40
 creating 41
 deleting 151
 erasing 151
 group 48
 locks 60
 mode, changing 47, 179
 name, changing 116
 new, opening 41
 offset 91
 opening 102
 owner 48
 reading bytes 111
 reading lines 114
 removing 151
 size 148
 status 140
 status flags 53, 59
 status information 65, 92
file access
 determining 22, 184
file descriptor 102
file descriptors
 0, 1, and 2 8, 16
 closing a range 49
 duplicating 42, 52
 error 8
 file attributes, changing 43
 flags 53, 59
 REXX environment 17
file locks 54, 62
file mode creation mask 150
file stream
 opening 173
file system
 mounted 67, 75, 76, 143
 mounting 95
 quiescing 109
 status 142
 unmounting 152
 unquiescing 153
file system server
 closing file opened by v_open 216
 creating a file 216
 creating a link 219
 creating a new directory 226
 creating a symbolic link 236
 file attributes, setting
 using v_setattr 234
 using v_settatro 235
 getting file attributes 219
 locking 220
 opening files 225
 reading file data 227
 registering a process as a server 230
 releasing a vnode token 230
 removing a directory entry 231
 removing an empty directory 233
 renaming a file or directory 232

file system server (*continued*)

 returning a vnode token 218
 returning entries from a
 directory 228
 returning file system status 217
 returning the contents of a symbolic
 link 229
 returning vnode and VFS tokens 233
 searching for a file 225
 syscall commands
 FID 215
 security 215
 tokens 215
 VFS 215
 vnode 215
 writing file data 237
file tag 64
for dynamic output 202
fork syscall command 56
forkexecm syscall command 56
fpathconf syscall command 57
fstat syscall command 65
fstatvfs syscall command 77
fsync syscall command 68
ftrunc syscall command 69
functions
 external 5
 REXX 173

G

getcwd syscall command 69
getegid syscall command 70
geteuid syscall command 70
getgid syscall command 70
getgrent syscall command 71
getgrgid syscall command 71
getgrnam syscall command 72
getgroups syscall command 73
getgroupsbyname syscall command 74
getlogin syscall command 74
getment syscall command 75
getmntent syscall command 76
getopts function 162
getpass() function 182
getpgrp syscall command 79
getpid syscall command 79
getppid syscall command 79
getpsent syscall command 80
getpwent syscall command 82
getpwnam syscall command 83
getpwuid syscall command 84
getrlimit syscall command 84
getuid syscall command 85
gmtime syscall command 85
GR_GID 71, 72, 73
GR_MEM 71, 72, 73
GR_MEMBERS 71, 72, 73
GR_NAME 71, 72, 73
group 71
 changing file or directory 39
 file or directory 89
 information 72
 symbolic link 89
group database
 retrieving an entry 71
 rewinding 120

group, file or directory 48

H

hard link 90
 removing 151
host command environment
 customizing 13
 SH 4
 input and output 7
 return code 22
 SYSCALL 2, 3
 working directory 19
host command environments
 z/OS UNIX 1

I

immediate commands 11
input and output
 syscall commands 8
 z/OS UNIX 7
ioctl syscall command 86
IRXEXEC service 16
IRXJCL service 16
isatty syscall command 87
ISPF editor 4

J

JES
 submitting a job to 194

K

keyboard
 navigation 255
 PF keys 255
 shortcut keys 255
kill syscall command 87

L

L_LEN 56, 61, 63
L_PID 56, 61, 63
L_START 56, 61, 63
L_TYPE 56, 61, 63
L_WHENCE 56, 61, 63
lchown syscall command 89
linein() function 182
lineout() function 183
lines() function 184
link
 hard 90, 151
 symbolic 151, 236
 creating to an external name 43
 reading the contents 115
link syscall command 90
linkage 15
LINKLIB 56
locks, file 60, 62
 querying 54
login name 74
LPALIB 56
lseek syscall command 91

lstat syscall command 92

M

macro language, using REXX 15
message catalog 30, 31
message queue
 creating 98
 locating 98
 obtaining status information 102
 receiving messages from 99
 removing 100
 sending messages to 101
 setting the attributes 100
message, finding and returning 31
mkdir syscall command 93
mkfifo syscall command 93
mknod syscall command 94
MNT_ASYNC MOUNT 78
MNT_FILEACTIVE 78
MNT_FILEDEAD 78
MNT_FILEDRAIN 78
MNT_FILEFORCE 78
MNT_FILEIMMED 78
MNT_FILENORM 78
MNT_FILERESET 78
MNT_IMMEDIATE 78
MNT_MODE_AUNMOUNT 77, 97
MNT_MODE_CLIENT 77
MNT_MODE_EXPORT 77
MNT_MODE_NOAUTOMOVE 77, 97
MNT_MODE_NOSEC 77, 97
MNT_MODE_NOSETID 77, 97
MNT_MODE_RDONLY 77, 97
MNT_MODE_RDWR 77, 97
MNT_MODE_SECACL 77
MNT_MOUNTINPROGRESS 78
MNT_QUIESCED 78
MNTE_AGGNAME 76
MNTE_BYTESREADHW 76
MNTE_BYTESREADLW 76
MNTE_BYTESWRITTENHW 76
MNTE_BYTESWRITTENLW 76
MNTE_DD 76
MNTE_DEV 76
MNTE_DIRIBC 76
MNTE_FILETAG 76, 96
MNTE_FROMSYS 76
MNTE_FSNAME 76, 96, 97
MNTE_FSTYPE 76, 96
MNTE_MODE 77, 97
MNTE_PARDEV 77
MNTE_PARM 77, 97
MNTE_PATH 77, 97
MNTE_PFSSTATUSEXCP 77
MNTE_PFSSTATUSNORMAL 77
MNTE_QJOBNAME 77
MNTE_QPID 77
MNTE_QSYSNAME 77
MNTE_READCT 77
MNTE_READIBC 77
MNTE_RFLAGS 77, 97
MNTE_ROOTINO 77
MNTE_ROSECLABEL 77
MNTE_STATUS 78
MNTE_STATUS2 78
MNTE_SUCCESS 78

MNTE_SYSLIST 78, 97
MNTE_SYSNAME 78, 97
MNTE_TYPE 78
MNTE_UID 78
MNTE_WRITECT 78
MNTE_WRITEIBC 78
mode changing 37, 47
mode specifying 19
modification time, set 154
mount syscall command 95
 MNT_MODE_AUNMOUNT 97
 MNT_MODE_NOAUTOMOVE 97
 MNT_MODE_NOSEC 97
 MNT_MODE_NOSETID 97
 MNT_MODE_RDONLY 97
 MNT_MODE_RDWR 97
 MNTE_FILETAG 96
 MNTE_FSNAME 96
 MNTE_FSTYPE 96
 MNTE_MODE 97
 MNTE_PARM 97
 MNTE_PATH 97
 MNTE_RFLAGS 97
 MNTE_SYSLIST 97
 MNTE_SYSNAME 97
mounted file systems 75, 76
mountnt syscall command
 MNTE_FSNAME 97
msgget syscall command 98
msgrcv syscall command 99
msgrmid syscall command 100, 102
msgset syscall command 100
msgsnd syscall command 101
MTM_DRAIN 152
MTM_FORCE 152
MTM_IMMEDIATE 152
MTM_NORMAL 152
MTM_NOSECURITY 96
MTM_NOSUID 96
MTM_RDONLY 96
MTM_RDWR 96
MTM_REMOUNT 152
MTM_RESET 152
MTM_SAMEMODE 152
MTM_SYNCHONLY 96

N

navigation
 keyboard 255
Notices 259
null-terminated parameter string 199
NUMBER OFF 4
numeric GID 122
numeric UID 122
numerics 20

O

O_APPEND 54, 59, 103
O_CREAT 54, 103
O_EXCL 54, 103
O_NOCTTY 54, 103
O_NONBLOCK 54, 59, 103, 108
O_RDONLY 54, 103
O_RDWR 54, 103

- O_SYNC 54, 59, 103
- O_TRUNC 54, 103
- O_WRONLY 54, 103
- octal numbers 253
- offset, file 91
- open syscall command 102
- opendir syscall command 104
- opening
 - file stream 173
 - process stream 174
- outtrap() function 184
- owner
 - changing file or directory 39
 - file or directory 89
 - symbolic link 89
- owner, file or directory 48

P

- parent process ID calling 79
- PARSE EXTERNAL instruction 8, 16
- PARSE SOURCE instruction tokens 9
- passthrough mode, 3270 108
- password
 - prompting for 182
 - returning 182
- path name
 - returning a 181
- pathconf syscall command 105
- pathname
 - specifying 19
 - terminal 149
- pause syscall command 106
- PC_ACL 106
- PC_ACL_MAX 106
- PC_LINK_MAX 58, 106
- PC_MAX_CANON 58, 106
- PC_MAX_INPUT 58, 106
- PC_NAME_MAX 58, 106
- PC_PATH_MAX 58, 106
- PC_PIPE_BUF 58, 106
- PC_POSIX_CHOWN_RESTRICTED 58, 106
- PC_POSIX_NO_TRUNC 58, 106
- PC_POSIX_VDISABLE 58, 106
- pending signals 134
- performance 13
- permissions
 - octal 253
 - setting 253, 254
- pfscctl syscall command 107
- pipe syscall command 107
- pipe, creating 108
- portable filename character set 19
- POSIX epoch time
 - converting timestamp to 179
- predefined variables
 - data type and numeric value 241
 - effect of PROCEDURE 7
 - using 21
- PROCEDURE instruction 7
- process
 - creating 137
 - retrieving information about 185
 - signal actions 132
 - spawning a child 137, 140
 - starting time 86

- process (*continued*)
 - status 80
 - wait
 - signals-enabled 190
- process group ID 79
 - assigning 121
- process ID 79
- process stream
 - opening 174
- processor time used 147
- procinfo() function 185
- program
 - executing from LINKLIB, LPALIB, or STEPLIB 56
 - run a REXX program 4
- programming services 15
- PS_CHILD 81
- PS_CMD 80
- PS_CONTTY 80
- PS_EGID 80
- PS_EUID 80
- PS_FGPID 80
- PS_FORK 81
- PS_FREEZE 81
- PS_MAXVNODES 80
- PS_MSGRCV 81
- PS_MSGSND 81
- PS_PATH 80
- PS_PAUSE 81
- PS_PGPID 80
- PS_PID 80
- PS_PPID 80
- PS_QUIESCE 81
- PS_RGID 80
- PS_RUID 80
- PS_RUN 81
- PS_SEMWT 81
- PS_SERVERFLAGS 80
- PS_SERVERNAME 80
- PS_SERVERTYPE 80
- PS_SGID 80
- PS_SID 80
- PS_SIZE 80
- PS_SLEEP 81
- PS_STARTTIME 80
- PS_STAT 80
- PS_STATE 81
- PS_SUID 81
- PS_SYSTIME 81
- PS_USERTIME 81
- PS_VNODECOUNT 81
- PS_WAITC 81
- PS_WAITF 81
- PS_WAITO 81
- PS_ZOMBIE 81
- PS_ZOMBIE2 81
- pt3270 syscall command 108
- PW_DIR 82, 83, 84
- PW_GID 82, 83, 84
- PW_NAME 82, 83, 84
- PW_SHELL 82, 83, 84
- PW_UID 82, 83, 84

Q

- quiesce syscall command 109

R

- R_OK 22
- RACF (Resource Access Control Facility)
 - BPX.SUPERUSER FACILITY 14
- RC 21
 - REXX variable 6
- rddir syscall command 110
- read syscall command 111
- readdir syscall command 113
- readfile syscall command 114
- readlink syscall command 114
- real group ID calling process 70, 119
- real user ID calling process 85, 126
- realpath syscall command 115
- relative pathname working directory 19
- remove a directory
 - rmdir 117
- remove a file
 - unlink 151
- rename syscall command 115
- reserved variables 21
- Resource Access Control Facility (RACF) 14
- resource limits 84, 123
- return code
 - from a REXX program 9
 - SH host command environment 22
 - TSO/E 21
 - z/OS shell 22
- RETVAL 21
- rewinddir syscall command 116
- REXX environment, customizing 13
- REXX external function parameter list 197
- REXX functions 173
 - bpxwunix() 174
 - charin() 176
 - charout() 177
 - chars() 178
 - chmod() 178
 - convd2e() 179
 - directory() 180
 - environment() 180
 - exists() 181
 - getpass() 182
 - linein() 182
 - lineout() 183
 - lines() 184
 - outtrap() 184
 - procinfo() 185
 - rexxopt() 189
 - sleep() 190
 - stream() 190
 - submit() 194
 - syscalls 194
- REXX program
 - exit status 9
 - from TSO/E or batch 10
 - moving from TSO/E to a z/OS shell 11
 - querying options 189
 - run from a program 9
 - run from a z/OS shell or a program 4
 - run from the z/OS shell 9
 - run from TSO/E or batch 2
 - setting options 189

REXX programming services 15
 REXX service
 IRXEXEC 16
 IRXJCL 16
 rexxopt() function 189
 RLIM_INFINITY 123
 RLIMIT_AS 85, 124
 RLIMIT_CORE 85, 124
 RLIMIT_CPU 85, 124
 RLIMIT_FSIZE 85, 124
 RLIMIT_NOFILE 85, 124
 rmdir syscall command
 remove or delete a directory 117
 routine, signal interface 3

S

S_FFBINARY 67, 142
 S_FFCR 67, 142
 S_FFCRLF 67, 142
 S_FFLF 67, 142
 S_FFLFCR 67, 142
 S_FFNA 67, 142
 S_FFNL 67, 142
 S_ISCHR 67, 142
 S_ISDIR 67, 142
 S_ISFIFO 67, 142
 S_ISREG 67, 142
 S_ISSYM 67, 142
 SA_NOCLDSTOP 133
 saved set group ID calling process 119
 saved set user ID calling process 126
 SAY 16
 SAY instruction 8
 SC_2_CHAR_TERM 147
 SC_ARG_MAX 146
 SC_CHILD_MAX 146
 SC_CLK_TCK 146
 SC_JOB_CONTROL 146
 SC_NGROUPS_MAX 146
 SC_OPEN_MAX 146
 SC_SAVED_IDS 146
 SC_THREAD_TASKS_MAX_NP 146
 SC_THREADS_MAX_NP 147
 SC_TZNAME_MAX 147
 SC_VERSION 147
 scope, variable 7
 seconds since Epoch 147
 security
 file system server 215
 Resource Access Control Facility
 (RACF) 14
 SEEK_CUR 56, 61, 63, 91
 SEEK_END 56, 61, 63, 91
 SEEK_SET 56, 61, 63, 91
 sending comments to IBM xvii
 sequence numbers, ISPF 4
 set the supplemental group list 120
 setegid syscall command 118
 seteuid syscall command 118
 setgid syscall command 119
 setgrent syscall command 120
 setgroups syscall command 120
 setpgid syscall command 121
 setpwent syscall command 121
 setregid syscall command 122
 effective GID 122

setregid syscall command (*continued*)
 real GID 122
 setreuid syscall command 122
 effective UID 122
 real UID 122
 setrlimit syscall command 123
 RLIM_INFINITY 123
 setsid syscall command 125
 setting, supplemental group list 120
 setuid syscall command 125
 SH environment 4
 input and output 7
 shared memory lock
 initializing 129
 obtaining 129
 releasing 130
 removing 128
 shared memory segment
 attaching 127
 creating 128
 detaching 127
 locating 128
 obtaining status information for 131
 removing 130
 setting attributes for 131
 shell command, running 174
 shmat syscall command 127
 shmdt syscall command 127
 shmget syscall command 127
 shmldestroysyscall command 128
 shmldinit syscall command 129
 shmldobtain syscall command 129
 shmldrelease syscall command 130
 shmldmid syscall command 130
 shmldset syscall command 131
 shmldstat syscall command 131
 shortcut keys 255
 SIG_BLOCK 135
 SIG_CAT 133
 SIG_DFL 133
 SIG_IGN 133
 SIG_QRY 133
 SIG_SETMASK 135
 SIG_UNBLOCK 135
 SIGABND 88, 133
 SIGABRT 88, 133
 sigaction syscall command 132
 SIG_QRY 133
 SIGALRM 88, 133
 SIGALRM, generating 30
 SIGBUS 88, 133
 SIGCHLD 88, 133
 SIGCONT 88, 133
 SIGDANGER 133
 SIGFPE 88, 133
 SIGHUP 88, 133
 SIGILL 88, 133
 SIGINT 88, 133
 SIGIO 88, 133
 SIGIOERR 88, 133
 SIGKILL 88, 133
 signal
 catcher 10
 interface routine (SIR) 3
 mask 134, 136
 pausing for delivery 106
 pending 134

signal (*continued*)
 REXX environment 16
 services 10
 SIGALRM 30
 signal action 132
 signal interface routine
 establish or delete 194
 sigpending syscall command 134
 SIGPIPE 88, 133
 SIGPOLL 88, 133
 sigprocmask syscall command 134
 SIGPROF 88, 133
 SIGQUIT 88, 133
 SIGSEGV 88, 133
 SIGSTOP 88, 133
 sigsuspend syscall command 135
 SIGSYS 88, 133
 SIGTERM 88, 133
 SIGTRAP 88, 133
 SIGTSTP 88, 133
 SIGTTIN 88, 133
 SIGTTOU 88, 133
 SIGURG 88, 133
 SIGUSR1 88, 133
 SIGUSR2 88, 133
 SIGVTALRM 88, 133
 SIGWINCH 133
 SIGXCPU 88, 133
 SIGXFSZ 88, 133
 sleep syscall command 136
 sleep() function 190
 spawn syscall command 137
 spawnp syscall command 140
 special file, character 94
 special file, FIFO 93
 ST_AAUDIT 34, 44, 67, 142
 ST_ACCESSACL 67, 142
 ST_ETIME 34, 44, 67, 142
 ST_AUDITID 67, 142
 ST_BLKSIZE 67, 142
 ST_BLOCKS 67, 142
 ST_CCSID 34, 44, 67, 142
 ST_CRTIME 67, 142
 ST_CTIME 34, 44, 67, 142
 ST_DEV 67, 142
 ST_DMODELACL 67, 142
 ST_EXTLINK 67, 142
 ST_FID 67, 142
 ST_FILEFMT 67, 142
 ST_FMODELACL 67, 142
 ST_GENVALUE 34, 44, 67, 142
 ST_GID 67, 142
 ST_INO 67, 142
 ST_MAJOR 67, 142
 ST_MINOR 67, 142
 ST_MODE 34, 44, 67, 142
 ST_MTIME 34, 44, 67, 142
 ST_NLINK 67, 142
 ST_RTIME 34, 44, 67, 142
 ST_SETGID 34, 44, 67, 142
 ST_SETUID 34, 44, 67, 142
 ST_SIZE 34, 44, 67, 142
 ST_STICKY 34, 44, 67, 142
 ST_TYPE 67, 142
 ST_UAUDIT 34, 44, 67, 142
 ST_UID 34, 44, 67, 142
 starting time 86

- stat syscall command 140
- statfs syscall command 142, 143
- status
 - file 140
 - file system 142
 - process 80
- statvfs syscall command 143
- STDERR 8
- STDIN 8
- STDOUT 8
- stem, specifying 19
- STEPLIB 56
- STFS_AVAIL 68, 143, 144
- STFS_BFREE 68, 143, 144
- STFS_BLOCKSIZE 68, 143, 144
- STFS_FAVAIL 68, 143, 144
- STFS_FFEE 68, 143, 144
- STFS_FILES 68, 143, 144
- STFS_FRSIZE 68, 143, 144
- STFS_FSID 68, 143, 144
- STFS_INUSE 68, 143, 144
- STFS_INVARSEC 68, 143, 144
- STFS_NAMEMAX 68, 143, 144
- STFS_NOSEC 143, 144
- STFS_NOSUID 68, 143, 144
- STFS_RDONLY 68, 143, 144
- STFS_TOTAL 68, 143, 144
- stream
 - characters
 - returning a number of 177
 - returning a string of 176
 - returning the remaining 178
 - checking for data 184
 - file 173
 - opening implicitly 173
 - process 174
 - reading a line 182
 - returning the state of 191
 - writing a line 183
- stream()
 - commands 191
- stream() function 190
- strerror syscall command 144
- strings, specifying 20
- su shell command 14
- submit() function 194
- subroutines 5
- Summary of changes xv
- superuser 14
- supplementary group ID calling
 - process 73
- supplementary group ID user 74
- symbolic link
 - creating 145
 - creating to an external name 43
 - owner 89
 - reading the contents 115
 - removing 151
 - resolving 236
- symlink syscall command 145
- syntax diagrams
 - how to read xi
- syscall commands 1
 - specifying 19
 - VFS server 215
- SYSCALL commands
 - for input and output 8

- SYSCALL environment 2
 - ending the 3
 - establish or end 194
- syscalls function 194
- sysconf syscall command 146
- sysplex
 - getmntent syscall command 76
 - mount syscall command 95

T

- tag, file 64
- terminal pathname 149
- time
 - access and modification 154
 - processor 147
- time syscall command 147
- times syscall command 147
- timestamp
 - converting to POSIX epoch time 179
- TM_HOUR 86
- TM_ISDST 86
- TM_MDAY 86
- TM_MIN 86
- TM_MON 86
- TM_SEC 86
- TM_WDAY 86
- TM_YDAY 86
- TM_YEAR 86
- TMS_CSTIME 148
- TMS_CUTIME 148
- TMS_STIME 148
- TMS_UTIME 148
- tokens
 - file system server 215
 - returned from PARSE SOURCE
 - instruction 9
- trunc syscall command 148
- TSO
 - command environment 5
- TSO command environment 5
- TSO commands
 - examples 6
- TSO/E, running a REXX program 2
- tsocmd command 7
- ttyname syscall command 149

U

- U_MACHINE 151
- U_NODENAME 151
- U_RELEASE 151
- U_SYSNAME 151
- U_VERSION 151
- uname syscall command 150
- unlink syscall command
 - remove or delete a file 151
 - unlink a file 151
- unmount syscall command 152
- unquiesce syscall command 153
- user
 - identified by user ID 84
 - identified by user name 83
 - login name 74
 - supplementary group IDs 74

- user database
 - entry retrieval 82
 - rewinding the 121
- user interface
 - ISPF 255
 - TSO/E 255
- utime syscall command 153

V

- v_close command 215
- v_create syscall command 216
- v_fstatfs syscall command 217
- v_get syscall command 218
- v_getattr syscall command 219
- v_link syscall command 219
- v_lockctl syscall command 220
- v_lookup syscall command 224
- v_mkdir syscall command 226
- v_open syscall command 225
- v_read syscall command 227
- v_readdir syscall command 228
- v_readlink syscall command 229
- v_reg syscall command 229
- v_rel syscall command 230
- v_remove syscall command 231
- v_rename syscall command 232
- v_rmdir syscall command 233
- v_rpn syscall command 233
 - MNT_MODE_AUNMOUNT 77
 - MNT_MODE_CLIENT 77
 - MNT_MODE_EXPORT 77
 - MNT_MODE_NOAUTOMOVE 77
 - MNT_MODE_NOSEC 77
 - MNT_MODE_NOSETID 77
 - MNT_MODE_SECACL 77
 - MNTE_BYTESREADHW 76
 - MNTE_BYTESREADLW 76
 - MNTE_BYTESWRITTENHW 76
 - MNTE_BYTESWRITTENLW 76
 - MNTE_DIRIBC 76
 - MNTE_FROMSYS 76
 - MNTE_QSYSNAME 77
 - MNTE_READCT 77
 - MNTE_READIBC 77
 - MNTE_RFLAGS 77
 - MNTE_STATUS2 78
 - MNTE_SUCCESS 78
 - MNTE_SYSLIST 78
 - MNTE_SYSNAME 78
 - MNTE_WRITECT 78
 - MNTE_WRITEIBC 78
- v_setattr syscall command 234
- v_setattro syscall command 235
- v_symlink syscall command 236
- v_write syscall command 237
- variable
 - predefined 241
 - reserved 21
 - scope 7
 - specifying 20
- VFS 215
- VFS server, syscall commands 215
- vnode 215

W

W_CONTINUED 154
W_EXITSTATUS 154
w_getmntent syscall command
 MNT_MODE_AUNMOUNT 77
 MNT_MODE_CLIENT 77
 MNT_MODE_EXPORT 77
 MNT_MODE_NOAUTOMOVE 77
 MNT_MODE_NOSEC 77
 MNT_MODE_NOSETID 77
 MNT_MODE_SECACL 77
 MNTE_BYTESREADHW 76
 MNTE_BYTESREADLW 76
 MNTE_BYTESWRITTENHW 76
 MNTE_BYTESWRITTENLW 76
 MNTE_DIRIBC 76
 MNTE_FROMSYS 76
 MNTE_QSYSNAME 77
 MNTE_READCT 77
 MNTE_READIBC 77
 MNTE_RFLAGS 77
 MNTE_STATUS2 78
 MNTE_SUCCESS 78
 MNTE_SYSLIST 78
 MNTE_SYSNAME 78
 MNTE_WRITECT 78
 MNTE_WRITEIBC 78
W_IFEXITED 154
W_IFSIGNALED 154
W_IFSTOPPED 154
W_OK 22
W_STAT3 155
W_STAT4 155
W_STOPSIG 155
W_TERMSIG 155
wait syscall command 154
waitpid syscall command 155
working directory 19
 changing 37
 pathname 69
write syscall command 157
writefile syscall command 159

X

X_OK 22

Z

z/OS shell
 return code 22
 run a REXX program from 4
z/OS UNIX
 host command environments 1
 REXX functions 173
z/OS UNIX processing
 input and output 7
 using TSO/E REXX 1



Product Number: 5650-ZOS

Printed in USA

SA23-2283-00

