sa — Command

Print a summary of process accounting
sa [-abcjlmnrstu] [-v *N*] **[***file***]**

One of the accounting mechanisms available on the COHERENT system is *process accounting* (also called *shell accounting*), which records the commands executed by each user. The command **accton** enables or disables shell accounting.

The command **sa** scans the accounting information in *file* and prints a summary. If *file* is omitted, it reads the file **/usr/adm/acct** by default. For each command executed, **sa** prints the number of calls made, the total CPU time (user and system), and the total real time. The output is ordered by decreasing CPU time.

sa recognizes the following options:

a Place commands executed only once and command names with unprintable characters in the category “***other”.

b Sort by average CPU time per call.

c Also print CPU time as a percentage of all CPU time used.

j Print average times per call rather than totals.

l Separate user and system time.

m Accumulate information for each user rather for each command.

n Sort by number of calls.

r Reverse the order of the sort.

s After scanning, condense the accounting file and merge it into the summary files.

t Also print the CPU time as a percentage of real time.

u Print the user and command for each accounting record; this option overrides all others.

v *N* For commands called no more than *N* times, where *N* is a digit, **sa** asks whether to place the command in the category “**junk**”.

sa uses the summary files **/usr/adm/savacct** and **/usr/adm/usracct** to lessen disk usage.

Files

/usr/adm/acct — Default account data
/usr/adm/savacct — Summary
/usr/adm/usracct — Summary

See Also

ac, acct(), acct.h, accton, commands

Notes

The file **/usr/adm/acct** can become very large; therefore, you should truncate it periodically. Special care should be taken if process accounting is enabled on a COHERENT system with limited disk space.

savelog — Command

Save a mail log
/usr/lib/mail/savelog [-c *cycle***] [-g** *group***] [-l]**
[-m *mode***] [-u** *user***] [-t]** *file ...*

The script **savelog** archives each *file*. This script normally is used to save copies of **smail**'s log files, and of other files that grow relentlessly. It copies each *file* into a special archiving directory, and gives the copy a name that reflects how recently it was created. Unless you request otherwise, it also compresses each *file*.

When it saves *file*, **savelog** copies it into directory **$PWD/OLD**, where **$PWD** represents the directory within which the file normally resides. If sub-directory **OLD** does not exist, **savelog** creates it, and gives it mode 0755.

As you probably will invoke **savelog** periodically to save a log file, this directory can hold an indefinite number of archives of *file*, each created at a different time in the past. To help you distinguish among these archives, **savelog** names them as follows:

> *file***.***number***[.***compression_suffix***]**

number represents the order in which the archives were created, zero being the newest; and *compression_suffix* indicates the suffix that the compression program gives the file — **.Z** if the archive is compressed with **compress** (which **savelog** uses by default), or **.gz** if compressed with **gzip**. Note that archive '0' is never compressed, on the off chance that a process still has its corresponding file opened for input.

If *file* does not exist or has zero length, **savelog** performs no further processing. To override this behavior, use option **-t**.

When *file* exists and has a length greater than zero, **savelog** performs the following actions:

- First, it increases by one the version number of each existing copy of *file*. For example, if you are saving file **foo** for the seventh time, then **savelog** moves file **foo.6** to **foo.7**; then moves **foo.5** to **foo.6**; and so on. **savelog** does this regardless of whether an archive is compressed, or whether you used option **-t** on the command line. By default, **savelog** keeps only seven versions of a given *file*, and throws away those versions that exceed that limit. To increase or decrease this limit, use command-line option **-c**, described below.
- If you did *not* use command-line option **-t**, **savelog** next compresses the new *file***.1**. It also changes this file, subject to the command-line options **-m**, **-u**, and **-g** (described below).
- It moves *file* to **OLD/***file.0*.
- If you use any of the command-line options **-m**, **-u**, **-g**, or **-t**, **savelog** re-creates *file*, subject to the given flags.
- Finally, **savelog** modifies the newly created file **OLD/***file***.0**, subject to the settings of command-line options **-m**, **-u**, and **-g**.

Command-line Options

savelog recognizes the following command-line options:

-c *cycle* Save no more than *cycle* versions of *file*. The default is seven, numbered '0' through '6'. *cycle* must be no less than two. Note that because numbering begins with zero, version number *cycle* of *file* is never created.

-g *group*
Use the command **chgrp** to give *group* the group ownership of *file* and its archives.

-l Do not compress any log files.

-m *mode*
Invoke the command **chmod** to set permissions on the log files to *mode*.

-t Touch *file*— that is, create a new, empty copy of *file* after archiving it. This lets you ensure that the log file is re-created with correct permissions.

-u *user* Invoke the command **chown** to make *user* the owner of the archives of *file*.

See Also

commands, mail [overview], uulog

Notes

If you do not use any of the command-line options **-m**, **-u**, or **-g**, **savelog** does not re-create *file* after archiving it.

Copyright © 1987, 1988 Ronald S. Karr and Landon Curt Noll. Copyright © 1992 Ronald S. Karr.

For details on the distribution rights and restrictions associated with this software, see file **COPYING**, which is included with the source code to the **smail** system; or type the command: **smail -bc**.

sbrk() — General Function (libc)

Increase a program's data space
#include <unistd.h>
char *sbrk(*increment***) unsigned int** *increment***;**

sbrk() increases a program's data space by *increment* bytes. **malloc()** calls **sbrk()** should you attempt to allocate more space than is available in the program's data space.

If all goes well, **sbrk()** returns the old break value. Otherwise, if an error occurs, **sbrk()** returns -1 and sets **errno** to an appropriate value.

See Also

brk(), libc, malloc()

Notes

sbrk() will not increase the size of the program data area if the physical memory requested exceeds the physical memory allocated by COHERENT. **sbrk()** does not keep track of how space is used; therefore, memory seized with **sbrk()** cannot be freed. *Caveat utilitor.*

scanf() — STDIO Function (libc)

Accept and format input
#include <stdio.h>
int scanf(*format, arg1, … argN***)**
char **format***; [data type]** **arg1, … *argN***;**

scanf() reads the standard input, and uses the string *format* to specify a format for each *arg1* through *argN*, each of which must be a pointer.

scanf() reads one character at a time from *format*; white space characters are ignored. The percent sign character '%' marks the beginning of a conversion specification. '%' may be followed by characters that indicate the width of the input field and the type of conversion to be done.

scanf() reads the standard input until the return key is pressed. Inappropriate characters are thrown away; e.g., it will not try to write an alphabetic character into an **int**.

scanf() returns the number of arguments filled. It returns EOF if no arguments can be filled or if an error occurs.

Modifiers

The following modifiers can be used within the conversion string:

1. An asterisk '*', which tells **scanf** to skip the next conversion; that is, read the next token but do not write it into the corresponding argument.
2. A decimal integer, which tells **scanf** the maximum width of the next field being read. How the field width is used varies among conversion specifier. See the table of specifiers below for more information.
3. One of the three modifiers **h**, **l**, or **L**, whose use is described below.

Modifiers

The following three modifiers may be used before a conversion specifier:

h When used before the conversion specifiers **d**, **i**, **o**, **u**, **x**, or **X**, it specifies that the corresponding argument points to a **short int** or an **unsigned short int**. When used before **n**, it indicates that the corresponding argument points to a **short int**. In implementations where **short int** and **int** are synonymous, it is not needed. However, it is useful in writing portable code.

l When used before the conversion specifiers **e**, **E**, **f**, **F**, or **G**, it indicates that the corresponding argument points to a **double** rather than a **float**.

L When used before the conversion specifiers **e**, **E**, **f**, **F**, or **G**, it indicates that the corresponding argument points to a **long double** rather than a **float**.

Conversion Specifiers

scanf() recognizes the following conversion specifiers:

c Assign the next input character to the next *arg*, which should be of type **char** *. The field width specifies the number of characters (default, one). **scanf()** does not write a null character at the end of the array it creates. This specifier forces **scanf()** to read and store white-space characters and numerals, as well as letters.

d Convert the token to a decimal integer. The format should be equivalent to that expected by the function **strtol()** with a base argument of ten. The corresponding argument should point to an **int**.

D Assign the decimal integer from the next input field to the next *arg*, which should be of type **long** *.

e Convert the token to a floating-point number. The format of the token should be that expected by the function **strtod()** for a floating-point number that uses exponential notation. The corresponding argument should point to a **float** if no modifiers are present, to a **double** if the **l** modifier is present, or to a **long double** if the **L** modifier is present.

E Same as **e**. Prior to release 4.2 of COHERENT, this conversion specifier converted the token to a **double**. This change has been made to conform to the ANSI Standard, and may require that some code be rewritten.

f Convert the token to a floating-point number. The format of the token should be that expected by the function **strtod()** for a floating-point number that uses decimal notation. The corresponding argument should point to a **double**.

g Convert the token to a floating-point number. The format of the token should of that expected by the function **strtod()** for a floating-point number that uses either exponential notation or decimal notation. The corresponding argument should point to a **float** if no modifiers are present, to a **double** if the **l** modifier is present, or to a **long double** if the **L** modifier is present.

G Same as **g**.

i Convert the token to a decimal integer. The format should be equivalent to that expected by the function **strtol()** with a base argument of zero. The corresponding argument should point to an **int**.

n Do not read any text. Write into the corresponding argument the number of characters that **scanf()** has read up to this point. The corresponding argument should point to an **int**.

o Assign the octal integer from the next input field to the next *arg*, which should be of type **int** *.

O Assign the octal integer from the next input field to the next *arg*, which should be of type **long** *.

p The ANSI standard states that the behavior of the **%p** conversion specificer is implementation-specific. Under COHERENT, **%p** converts a strings of digits in hexadecimal notation into an address. For example, in the code

```
char buf[] = "0x7FFFFDBC";
char *foo;
...
sscanf(buf, "%p", &foo);
```

the **%p** specifier reads the contents of **buf** and turns them into an address, which it then uses to initialize the pointer **foo**. You can use the **%p** specifier to turn back into an address the output of **printf()**'s **%p** specifier. Please note that abuse of this specifier can create all manner of fascinating bugs within your programs: *Caveat utilitor.*

r The next argument points to an array of new arguments that may be used recursively. The first argument of the list is a **char** * that contains a new format string. When the list is exhausted, the routine continues from where it left off in the original format string.

s Assign the string from the next input field to the next *arg*, which should be of type **char** *. The array to which the **char** * points should be long enough to accept the string and a terminating null character.

u Convert the token to an unsigned integer. The format should be equivalent to that expected by the function **strtoul()** with a base argument of ten. See **strtoul** for more information. The corresponding argument should point to an **unsigned int**.

x Convert the token from hexadecimal notation to a signed integer. The format should be equivalent to that expected by the function **strtol** with a base argument of 16. See the Lexicon entry for **strtol()** for more information. The corresponding argument should point to an **unsigned int**.

X Same as **x**. Prior to release 4.2 of COHERENT, **X** meant the same as the current **lx**; that is, the corresponding argument points to a **long** instead of an **int**. This has been changed to conform to the ANSI Standard, and may require that some code be rewritten.

It is important to remember that **scanf()** reads up, but not through, the newline character: the newline remains in the standard input device's buffer until you dispose of it somehow. Programmers have been known to forget to empty the buffer before calling **scanf()** a second time, which leads to unexpected results.

Example

The following example uses **scanf()** in a brief dialogue with the user.

```
#include <stdio.h>

main()
{
	int left, right;

	printf("No. of fingers on your left hand:  ");
	/* force message to appear on screen */
	fflush(stdout);
	scanf("%d", &left);

	/* eat newline char */
	while(getchar() != '\n')
		;

	printf("No. of fingers on your right hand:  ");
	fflush(stdout);
	scanf("%d", &right);

	/* again, eat newline */
	while(getchar() != '\n')
		;

	printf("You've %d left fingers, %d right, & %d total\n",
		left, right, left+right);
}
```

See Also

fscanf(), libc, sscanf()
ANSI Standard, §7.9.6.4
POSIX Standard, §8.1

Notes

Because C does not perform type checking, it is essential that an argument match its specification. For that reason, **scanf()** is best used to process only data that you are certain are in the correct data format. Rather than use **scanf()** to obtain a string from the keyboard: we recommend that you use **gets()** to obtain the string, and use **strtok()** or **sscanf()** to parse it.

scat — Command

Print text files one screenful at a time
scat [[*option ...***] [***file* **...]] ...**

scat prints each *file* on the standard output, one screenful (24 lines) at a time if the output is a screen. **scat** reads and prints the standard input if no *file* is named.

The text is processed to allow convenient viewing on a screen; the options described below select the nature of the processing. Options begin with '-' and may be interspersed with file names.

scat scans two argument lists. The first is in the environmental **SCAT**. It should consist of arguments separated by white space (space, tab, or newline characters), with no quoting or shell metacharacters. This string is a useful place to set terminal-dependent parameters (such as page width and length) and to place invocation lists (see below). The second argument list is supplied on the command line.

scat recognizes the following options:

-1 Do not stop at EOF if exactly one file was specified on the command line.

-b*n* Begin output at input line *n.*

-c Represent all control characters unambiguously. With this option, **scat** prints control characters in the range 0-037 as a character in the range 0100-0137 prefixed by a carat '^'; for example, SOH appears as "^A" and DEL as "^?" It prints mark-parity characters (in the range of 0200-0377) with '~'; for example, mark-parity 'A' and SOH appear as "~A" and "~^A", respectively. It also prefixes the characters '^', '~', and '\' with a '\'. This option overrides the option **-t**.

-cs Like **-c,** but map space ' ' to underscore '_' and prefix underscore '_' with '\'.

-ct Like **-c,** but map tabs to spaces, not "^I".

-i*n* Shift the display window right *n* columns into the text field. This is useful for viewing long lines.

-l*n* Set the display window length to *n* lines. The default is 24 normally, 34 for the Tek 4012.

-n Number input lines; wrapped lines are not numbered.

-r Remote; the output is not paged.

-s Skip empty lines.

-S*n* Seek *n* bytes into input before processing.

-t Truncate long lines. Normally, **scat** wraps each long line, with the interrupted portion delimited by a '\'.

-w*n* Set the display window width to *n* columns. The default is 80 normally, 72 for the Tek 4012.

-x Expand tabs.

-. *suffix* Invoke options by file-name suffix. If a file name ends with *.suffix*, then **scat** scans the argument sublist starting immediately after the invocation flag. New options will apply to the invoking file only. A sublist is terminated by the end of the argument list, by a file name, by the "--" flag, or by another "-." (invocation lists do not nest).

-- Terminate a sublist (see previous option).

Numbers may begin with **0** to indicate octal, and may end in **b** or **k** to be scaled by 512 or 1,024, respectively.

If the output is being paged, **scat** waits for a user response, which may be one of the following:

newline	Display next page
/	Display next half-page
space	Display next line
f	Print current file name and line number
n	**scat** next file
q	Quit

Example

The following shows how to use the environment argument list, invocation lists, and sublists:

```
SCAT="-l24 -.c -n -.s -b5"
export SCAT
scat *.c *.s
```

After processing the **SCAT** argument list, **scat** processes the command line argument list "***.c *.s**" with the page length at 24 lines. If a file is a C source ("***.c**") the invoke option in the **SCAT** argument list numbers the output lines. If a file is an assembly source ("***.s**") **scat** skips the first four lines.

See Also

cat, commands, pr

sched.h — Header File

Define constants used with scheduling
#include <sys/sched.h>

sched.h defines constants and structures that are used by routines that perform scheduling.

See Also

header files

script — Command

Capture a terminal session into a file
script [-l *logfile***] [***command***]**

The COHERENT command **script** executes *command* while copying all terminal output to *logfile*. *logfile* defaults to file **Log.***pid* in your current directory, where *pid* is the number of the recording process. *command* must specify a full path name. If the terminal echoes keyboard input, **script** records these keystrokes in *logfile*.

If no *command* is specified, **script** executes the command specified by environmental variable **SHELL** by default. If **SHELL** is not defined, **script** assumes **/bin/sh**.

To exit from **script**, just type **exit** from a command prompt.

See Also

commands

Notes

script is intended to capture what you type for purposes of debugging. What it captures cannot be replayed into the shell.

sdevice — System Administration

Configure drivers included within kernel
/etc/conf/sdevice

File **sdevice** configures the drivers that can be included within the COHERENT kernel. Command **idmkcoh** reads this file when it builds a new COHERENT kernel, and uses the information within it to configure the suite of drivers it links into the kernel.

There is one line within the file for each type of hardware device; if a driver manipulates more than one type of device, then it has one entry for each type of device it manipulates. A driver's entry within file **/etc/conf/mdevice** indicates how many entries a driver can have with **sdevice**: if field 3 contains flag 'o', the device can have only one entry; whereas if field 3 does not contain this flag, it can have more than one entry (although it is not required to do so). An entry that begins with a pound sign '#' is a comment, and is ignored by **idmkcoh**.

Each entry within **sdevice** consists of ten fields, as follows:

1. *Name*
This gives the name of driver, and must match the name given in **mdevice**. It cannot exceed eight characters.

2. *Included in Kernel?*
This field indicates whether the driver is to be linked into the kernel: 'Y' indicates that it is, 'N' that it is not.

3. *Number of Units*
The number of the hardware units that this driver can manipulate. Under COHERENT, this is always set to zero.

4. *Interrupt Priority*
The device's interrupt priority. This must be a value between 0 and 8: zero indicates that this device is not interrupt driven, whereas a value from 1 to 8 gives the interrupt priority.

5. *Interrupt Type*
The type of interrupt for this device. The legal values are as follows:

0 This device is not interrupt driven.

1 The device is interrupt driven. If the driver controls more than one device, each requires a separate interrupt.

2 The device is interrupt driven. If the driver supports more than one device, all share the same interrupt.

3 The device requires an interrupt line. If the driver supports more than one device, all share the same interrupt. Multiple device drivers that the same interrupt priority can share this interrupt; however, this requires special hardware support.

6. *Interrupt Vector*
The interrupt vector used by the device. If field 5 is set to zero, this must be also.

7. *Low I/O Address*
The low I/O address through which the driver communicates with the device. Set this field to zero if it is not used.

8. *High I/O Address*
The high I/O address through which the driver communicates with the device. Set this field to zero if it is not used.

9. *Low Memory Address*
The low address of memory within the controller of the device being manipulated. Set this field to zero if it is not used.

10. *High Memory Address*
The high address of memory within the controller of the device being manipulated. Set this field to zero if it is not used.

Note that all COHERENT drivers current set fields 7 through 10 to zero.

For examples of settings for this, read the file itself. For an example of modifying this file to add a new driver, see the Lexicon entry for **device drivers**.

See Also

Administering COHERENT, device drivers, mdevice, mtune, stune

sdiv() — Multiple-Precision Mathematics (libmp)

Divide multiple-precision integers
#include <mprec.h>
void sdiv(*a, n, q, ip***)**
mint **a*, **q*; **int** *n*, **ip*;

sdiv() divides the multiple-precision integer (or **mint**) pointed to by *a* with the integer *n*, which is in the range 1 <= *n* <= 128. It writes the quotient into the **mint** pointed to by *q* and the remainder into the integer pointed to by *ip*.

See Also

libmp

SECONDS — Environmental Variable

Number of seconds since current shell started

The Korn shell stores in environmental variable **SECONDS** the number of seconds since the current shell was started.

See Also

environmental variables, ksh

security — System Administration

Because COHERENT is a multi-user, multi-tasking operating system which can support users from remote terminals, steps must be taken to ensure that the system is secure. Sensitive information that is stored on the system must be protected from being read or copied by unauthorized persons; files must be protected against vandalization by intruders. Unless a reasonable degree can be guaranteed, no multi-user operating system can be trusted to archive important information.

In one sense, it is easy to achieve perfect security in a computer system. As Grampp and Morris have noted, "It is easy to run a secure computer system. You merely disconnect all dial-up connections, put the machine and its terminals in a shielded room, and post a guard at the door." For practical uses, however, security means balancing ease of access against restrictiveness: users should have easy access to what is properly theirs, and should be barred from system facilities that do not belong to them.

The COHERENT system has the following tools to assist with security.

Passwords Every user account can be "locked" with a password. Each user can assign her own password, and the system administrator can set passwords for the superusers **root** and **bin**.

Passwords should be changed frequently. A password should have at least six characters, should *not* be a common name or word, and preferably should include a mixture of upper- and lower-case letters, to prevent decryption by brute-force methods.

Passwords should be guarded jealously. In particular, the password for the superuser **root** should be kept secret, as she can read every file and execute every program throughout the system.

Permissions Execution of system-level programs, such as **mount**, is restricted to the superuser **root**. This prevents intruders from seizing superuser permissions through unauthorized manipulation of system services. Ordinary users are also restricted from directly access system devices, for the same reason.

One potential hole in security is the setting the **setuid** bit on programs that are owned by the superuser **root**. Setting this bit grant superuser privileges to whoever runs the program. Two commands often have this bit set: **/etc/enable** and **/etc/disable**. This is done to permit users, in particular user **uucp**, to enable and disable a port. This, however, permits any user to enable or disable a device — including the console device; which means that a cracker who breaks into your system could lock you out of it if she wished.

The lesson is that you should not set the **setuid** bit on any program that is owned by **root** unless you have an excellent reason to do so.

Encryption The command **crypt** performs rotary encryption, similar to that used by the German Enigma machine. Files of sensitive information should be encrypted, to protect them against being read by unauthorized persons. Note that encryption is the only true defense against unauthorized reading: not even the superuser can read an encrypted file unless she has the encryption key.

Many COHERENT systems have only one user and are not networked; for such installations, the normal level of security may be an annoyance. Passwords can be turned off by using the command **passwd** to set the password to **<return>**. The command **chmod** can be used to widen access to devices and system-level utilities; see the Lexicon entry for **chmod** for more information on file access.

Security ultimately is a system-wide responsibility. To quote Grampp and Morris, "By far, the greatest security hazard for a system ... is the set of people who use it. If the people who use a machine are naive about security issues, the machine will be vulnerable regardless of what is done by the local management. This applies particularly to the system's administrators, but ordinary users should also take heed."

See Also

Administering COHERENT, chmod, crypt, login, passwd
Grampp, F.T., Morris, R.H.: UNIX operating system security. *AT&T Bell Lab Tech J* 1984;8:1649-1672.

sed — Command

Stream editor
sed [-n] [-e *command***] [-f** *script***]** *... file ...*

sed is a non-interactive text editor. It reads input from each *file*, or from the standard input if no file is named. It edits the input according to commands given in the *commands* argument and the *script* files. It then writes the edited text onto the standard output.

sed resembles the interactive editor **ed**, but its operation is fundamentally different. **sed** normally edits one line at a time, so it may be used to edit very large files. After it constructs a list of commands from its *commands* and *script* arguments, **sed** reads the input one line at a time into a *work area*. Then **sed** executes each command that applies to the line, as explained below. Finally, it copies the work area to the standard output (unless the **-n** option is specified), erases the work area, and reads the next input line.

Line Identifiers

sed identifies input lines by integer line numbers, beginning with one for the first line of the first *file* and continuing through each successive *file*. The following special forms identify lines:

n — A decimal number *n* addresses the *n*th line of the input.

. — A period '.' addresses the current input line.

**$** — A dollar sign '$' addresses the last line of input.

/*pattern***/** — A *pattern* enclosed within slashes addresses the next input line that contains *pattern*. Patterns, also called *regular expressions*, are described in detail below.

Commands

Each command must be on a separate line. Most commands may be optionally preceded by a line identifier (abbreviated as *[n]* in the command summary below) or by two-line identifiers separated by a comma (abbreviated as *[n[,m]]*). If no line identifier precedes a command, **sed** applies the command to every input line. If one line identifier precedes a command, **sed** applies the command to each input line selected by the identifier. If two-line identifiers precede a command, **sed** begins to apply the command when an input line is selected by the first, and continues applying it through an input line selected by the second.

sed recognizes the following commands:

*[n]***=** — Output the current input line number.

*[n[,m]]***!***command*
Apply *command* to each line *not* identified by *[n[,m]]***.**

*[n[,m]]***{***command...***}**
Execute each enclosed *command* on the given lines.

:*label* — Define *label* for use in branch or test command.

*[n]***a** — Append new text after given line. New text is terminated by any line not ending in '\\'.

b *[label]* — Branch to *label*, which must be defined in a ':' command. If *label* is omitted, branch to end of command script.

*[n[,m]]***c** — Change specified lines to new text and proceed with next input line. New text is terminated by any line not ending in '\\'.

*[n[,m]]***d** — Delete specified lines and proceed with next input line.

*[n[,m]]***D** — Delete first line in work area and proceed with next input line.

*[n[,m]]***g** — Copy secondary work area to work area, destroying previous contents.

*[n[,m]]***G** — Append secondary work area to work area.

*[n[,m]]***h** — Copy work area to secondary work area, destroying previous contents.

*[n[,m]]***H** — Append work area to secondary work area.

*[n]***i** — Insert new text before given line. New text is terminated by any line not ending in '\\'.

*[n[,m]]***l** — Print selected lines, interpreting non-graphic characters.

*[n[,m]]***n** — Print the work area and replace it with the next input line.

*[n[,m]]***N** — Append next input line preceded by a newline to work area.

*[n[,m]]***p** — Print work area.

*[n[,m]]***P** — Print first line of work area.

*[n]***q** — Quit without reading any more input.

*[n]***r** *file* — Copy *file* to output.

*[n[,m]]***s***[k]/pattern1/pattern2/***[g][p][w** *file]*
: Search for *pattern1* and substitute *pattern2* for *k*th occurrence (default, first). If optional **g** is given, substitute all occurrences. If optional **p** is given, print the resulting line. If optional **w** is given, append the resulting line to *file*. Patterns are described in detail below.

*[n[,m]]***t***[label]*
: Test if substitutions have been made. If so, branch to *label*, which must be defined in a ':' command. If *label* is omitted, branch to end of command script.

*[n[,m]]***w** *file* Append lines to *file.*

[n[,m]] **x** Exchange the work area and the secondary work area.

*[n[,m]]***y***/chars/replacements/*
: Translate characters in *chars* to the corresponding characters in *replacements.*

Patterns

Substitution commands and search specifications may include *patterns*, also called *regular expressions*. Pattern specifications are identical to those of **ed**, except that the special characters '**\n**' match a newline character in the input.

A non-special character in a pattern matches itself. Special characters include the following:

^ Match beginning of line, unless it appears immediately after '[' (see below).

$ Match end of line.

\n Match the newline character.

. Match any character except newline.

* Match zero or more repetitions of preceding character.

[*chars***]** Match any one of the enclosed *chars*. Ranges of letters or digits may be indicated using '-'.

[^*chars***]** Match any character *except* one of the enclosed *chars*. Ranges of letters or digits may be indicated using '-'.

*****c* Disregard special meaning of character *c*.

\\(*pattern***\\)** Delimit substring *pattern*; for use with *****d*, described below.

In addition, the replacement part *pattern2* of the substitute command may also use the following:

& Insert characters matched by *pattern1*.

*****d* Insert substring delimited by *d*th occurrence of delimiters '\(' and '\)', where *d* is a digit.

Options

sed recognizes the following options:

-e Next argument gives a **sed** command. **sed**'s command line can have more than one **-e** option.

-f Next argument gives file name of command script.

-n Output lines only when explicit **p** or **P** commands are given.

Limits

The COHERENT implementation of **sed** sets the following limits on input and output:

Characters per input record	512
Characters per output record	512
Characters per field	512

See Also

commands, ed, elvis, ex, me, vi
Introduction to the sed Stream Editor

seed48() — Random-Number Function (libc)

Initialize values from which 48-bit random numbers are computed
unsigned short *seed48(*param***)**
unsigned short *param***[3];**

Computation of 48-bit pseudo-random numbers uses two 48-bit integers and one 16-bit integer. One of the 48-bit values holds the "seed" value from which the 48-bit pseudo-random value is computed. This seed can be set explicitly, or is the previously computed pseudo-random number. The other 48-bit integer holds the multiplier from which the pseudo-random number is computed; and the 16-bit integer gives holds the addend.

Function **seed48()** initializes the "seed" from which a 48-bit pseudo-random number is computed. *param* is an array of three unsigned short integers that together comprise the new 48-bit seed value.

seed48() returns a pointer to an array of three unsigned short integers that holds the old seed.

See Also

libc, srand48()

seekdir() — General Function (libc)

Reset the position within a directory stream
void seekdir (*dirp***,** *loc***)**
DIR **dirp***;**
off_t *loc***;**

The function **seekdir()** is one of a set of COHERENT routines that manipulate directories in a device-independent manner. It resets the current position within the directory stream pointed to by *dirp* to *loc*. *loc* must be a position indicator returned by a previous call to **telldir()**.

If an error occurs, **seekdir()** exits and sets **errno** to an appropriate value.

See Also

closedir(), dirent.h, getdents(), libc, opendir(), readdir(), rewinddir(), telldir()

Notes

telldir() and **seekdir()** are unreliable when the directory stream has been closed and reopened. It is best to avoid using **telldir()** and **seekdir()** altogether.

Because directory entries can dynamically appear and disappear, and because directory contents are buffered by these routines, an application may need to continually rescan a directory to maintain an accurate picture of its active entries.

The COHERENT implementation of the **dirent** routines was written by D. Gwynn.

seg.h — Header File

Definitions used with segmentation
#include <seg.h>

seg.h defines structures and constants used by routines that handle memory segmentation.

See Also

header files

select() — General Function (libsocket)

Check if devices are ready for activity
#include <sys/types.h>
#include <sys/time.h>
#include <sys/select.h>
#include <unistd.h>
int select(*nfds***,** *readfds***,** *writefds***,** *exceptfds***,** *timeout***)**
int *nfds***;**
fd_set **readfds***,** **writefds***,** **exceptfds***;**
struct timeval **timeout***;**

The function **select()** examines file descriptors, and tells you which are ready for a given type of activity. **select()** can be used with descriptors for sockets, pipes, and most character devices including the console, serial terminals connected via the **asy** driver, and pseudoterminals using the **pty** driver.

readfds, *writefds*, and *exceptfds* each gives the address of a a bit-map whose bits correspond to the file descriptors of the sockets that interest you. Respectively, these arguments identify the sockets that may have data to be read, those that to which you wish to write data, and those that may have an exception condition pending. (What an "error condition" may be, is described below.) **select()** examines descriptors zero through *nfds* in each set and checks whether the corresponding socket is ready for the activity in question. If the socket is not ready, **select()** flips off the bits that correspond to that socket.

Please note that although *readfds*, *writefds*, and *exceptfds* each is pointer to **int**, the bit-map it points to can be longer than 32 bits. You can, for example, declare that these pointers points to an array of **int**s. The number of file descriptors you can ask **select()** to examine limited by the manifest constant **FD_SETSIZE**, which is defined in header file **<sys/select.h>**. COHERENT sets this constant to 256; thus, if you set *nfds* to a value greater than 256, only the first 256 file descriptors will be examined.

If you are not interested in a given activity, set the corresponding pointer to NULL. For example, if you are interested only in reading and writing, but not in exception handling, set *exceptfds* to NULL.

timeout gives the address a **timeval** structure that holds the maximum time you are willing to wait for the selection to complete. If it is NULL, **select()** waits indefinitely.

By manipulating the value of *timeout*, you can perform some useful tricks. For example, if you set to zero the fields **tv_sec** and **tv_usec** within the **timeval** structure to which *timeout* points, **select()** performs a nonblocking poll of the indicated devices; this is demonstrated below. Another trick is to set field **tv_usec** within *timeout* to a nonzero value, but set *nfds* to zero. This tells **select()** to examine no sockets, but to wait the specified number of microseconds while not doing it. This lets you "sleep" for an interval shorter than is possible through the system call **sleep()**, whose minimum delay is one second.

If all goes well, **select()** returns the number of sockets that are ready. If the time limit expires, it returns zero. If an error occurs, it leaves all three bit maps unmodified, returns -1, and sets **errno** to one of the following values:

EBADF A descriptor set specifies an invalid descriptor. For example, this error occurs if one of the file descriptors does not describes an ordinary file instead of a socket.

EINTR **select()** received a signal before the time limit expired and before it could finish examining the sockets.

EINVAL
The time structure to which *timeout* points contains invalid data: one of its components is negative or too large.

The following example code demonstrates how to set up a socket and examine it with **select()**. is taken from a program written by Jon Dhuse (jdhuse@sedona.intel.com), and was slightly modified for clarity. The entire program appears in the Lexicon entry **libsocket**:

```
int sd[2], rdfds[2], wrtfds[2], i;
struct timeval timeout;
      ...

/* create socket */
sd = socket(AF_UNIX, SOCK_STREAM, 0);
      ...

/* initialize the arrays of ints */
for (i = 1; i < 2; i++)
      rdfs[i] = 0;
      wrtfds[i] = 0;
}
      ...
```

```
/* Check whether socket is ready */
rdfds[0] = 1 << sd;     /* initialize bit map to check for reading */
wrtfds[0] = 1 << sd;    /* initialize bit map to check for writing */
timeout.tv_sec = 0;
timeout.tv_usec = 0;
i = select(sd+1, rdfds, wrtfds, (int *)NULL, &timeout);
if (i < 0)
        printf("select() returned error %d\n", errno);
else {
      if (rdfds & (1 << sd)) /* check if socket has data */
            printf("socket has data to be read\n");
      if (wrtfds & (1 << sd)) /* check if socket can be written to */
            printf("data can be written to socket\n");
}
```

Associated Macros

The header file **<sys/select.h>** defines the following macros, which are meant to help you manipulate sets of file descriptors:

FD_ZERO (&*fdset*)
Initialize the bit map *fdset* to zero.

FD_SET (*fd*, &*fdset*)
Turn on bit *fd* within the bit map *fdset*.

FD_CLR (*fd*, &*fdset*)
Turn off bit *fd* within the bit map *fdset*.

FD_ISSET (*fd*, *fdset*)
This macro evaluates to a non-zero value if bit *fd* is turned on within *fdset*; otherwise, it evaluates to zero.

The behavior of these macros is undefined if a descriptor's value is less than zero or greater than or equal to **FD_SETSIZE**.

Exception Conditions

As noted above, the bit map *exceptfds* identifies sockets that may have an exception condition pending. As of this writing, COHERENT defines an "exception condition" to be one of the following:

POLLHUP
A hangup has occurred, i.e., loss of carrier on a modem line or closure of the associated master device when **select()** queries a slave pseudo-tty.

POLLNVAL
The file descriptor does not correspond to an open device.

See Also

accept(), connect(), libsocket, poll(), read(), write()

Notes

The system call **poll()** uses a different calling sequence to do much the same work as **socket()**.

sem — Kernel Module

Kernel module for semaphores

The kernel module **sem** enables System V-style semaphores. It is called a *kernel module* because you can link it into your kernel or exclude it, as you wish, just like a device driver; yet it is not a true device devicer because it does not perform I/O with a peripheral device.

See Also

device drivers, kernel, semctl()

sem.h — Header File

Definitions used by semaphore facility
#include <sys/sem.h>

sem.h defines constants and structures used by the COHERENT semaphore facility.

See Also

header files, semget()

semctl() — General Function (libc)

Control semaphore operations
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>
int semctl(*id***,** *number***,** *command***,** *arg***)**
int *id***,** *command***,** *number***;**
union semun {
    int *value***;**
    struct semid_ds **buffer***;**
    unsigned short *array***[];**
} *arg***;**

The function **semctl()** controls the COHERENT system's semaphore facility.

A set of semaphores consists of a copy of structure **semid_ds**, which is defined in header file **<sys/sem.h>**. This structure points to the set of semaphores, notes how many semaphores are in the set, and gives information on who can manipulate it, and how. The semaphores themselves consist of an array of structures of type **sem**, which is also defined in **sem.h**. When the function **semget()** creates a set of semaphores, it assigns to that set an identification number and returns that number to the calling process. For details on this process, see the Lexicon entry for **semget()**

id identifies the set of semaphores to be manipulated. This value must have been returned by a call to **semget()**. *number* gives the offset within the set identified by *id* of the semaphore that interests you. *arg* gives information to be passed to, or received from, the semaphore in question. *command* names the operation that you want **semctl()** to perform.

The following *command*s manipulate semaphore *number* within the set identified by *id*:

GETVAL Return the value of **semval**, which is the field in structure **sem** that gives the address of the semaphore's text map.

SETVAL Set **semval** to *arg.value*. If an "adjust value" had been created for this semaphore (by changing or setting a semaphore through **semop()** with the flag **SEM_UNDO** set), it is erased.

GETPID Return the value of **sempid**, which is the field in **sem** that identifies the last process to have manipulated this semaphore.

GETNCNT Return the value of **semncnt**, which gives the number of processes that await an increase in field **sem.semval**.

GETZCNT Return the value of **semzcnt**, which gives the number of processes that are waiting for the value of **sem.semval** to become zero.

The following *command*s return or set field **semval** within every semaphore in the set identified by *id*:

GETALL Write every **semval** into *arg.array*.

SETALL Initialize every **semval** to the corresponding value within *arg.array*. All "adjust values" for this semaphores are erased.

semctl() also recognizes the following *command*s:

IPC_STAT Copy the value of each semaphore in the set identified by *id* into the structure pointed to by *arg.buffer*.

IPC_SET Copy fields **sem_perm.uid**, **sem_perm.gid**, and **sem_perm.mode** (low nine bits only) from the **ipc_perm** associated with *id* into that pointed to *arg.buffer*. Only the superuser **root** or the user whose effective user ID matches the value of field **uid** in the data structure identified by *id* can invoke this command.

IPC_RMID Destroy the **semid_ds** structure identified by *id*, plus its array of semaphores. Only the superuser **root** or the user whose effective user ID matches the value of field **uid** can invoke this command.

semctl() fails if one or more of the following is true:

- *id* is not a valid semaphore identifier. **semctl()** sets the global variable **errno** to **EINVAL**.
- *number* is less than zero or greater than field **sem_nsems** in structure **semid_ds**, which gives the number of semaphores in the set identified by *id* (**EINVAL**).
- *command* is not a valid command (**EINVAL**).
- The calling process is denied operation permission (**EACCES**).
- *command* is **SETVAL** or **SETALL**, but the value of **semval** exceeds the system-imposed maximum (**ERANGE**).
- *command* is **IPC_RMID** or **IPC_SET**, but the calling process is owned neither by **root** nor by the user who created the set of semaphores being manipulated (**EPERM**).
- *arg.buffer* points to an illegal address (**EFAULT**).

semctl() returns the following values upon successful completion of their following commands:

Command	*Return Value*
GETVAL	Value of **semval**
GETPID	Value of **sempid**
GETNCNT	Value of **semncnt**
GETZCNT	Value of **semzcnt**

For all other commands, **semctl()** returns zero to indicate successful completion.

If it could not execute a command successfully, **semctl()** returns -1 and sets **errno** to an appropriate value.

Files

/usr/include/sys/ipc.h
/usr/include/sys/sem.h

See Also

libc, semget(), semop()

Notes

For information on other methods of interprocess communication, see the Lexicon entries for **msgctl()** and **shmctl()**.

semget() — General Function (libc)

Create or get a set of semaphores
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>
semget(*semkey*, *number*, *flag***)**
key_t *semkey*; **int** *number*, *flag*;

semget() creates a set of semaphores plus its associated data structure and identifier, links them to the identifier *semkey*, and returns the identifier that it has associated with *semkey*.

semkey is an identifier that your application generates to identify its semaphores.

number gives the of semaphores you want **shmget()** to create.

flag can be bitwise OR'd to include the following constants:

IPC_ALLOC This process already has a set of semaphores; please fetch it.

IPC_CREAT If this process does not have a set of semaphores, please create one.

IPC_EXCL Fail if this process already has a set of semaphores.

IPC_NOWAIT Fail if the process must wait to obtain a set of semaphores.

When it creates a set of semaphores, **semget()** also creates a copy of structure **semid_ds**, which the header file **<sys/sem.h>** defines as follows:

```
struct semid_ds {
       struct ipc_perm sem_perm;         /* operation permission struct */
       struct sem *sem_base;             /* pointer to first semaphore in set */
       unsigned short sem_nsems;         /* # of semaphores in set */
       time_t sem_otime;                 /* last semop time */
       time_t sem_ctime;                 /* last change time */
};
```

Field **sem_base** points the semaphores themselves. Each semaphore is a structure of type **sem**, which header file **<sys/sem.h>** defines as follows:

```
struct sem {
       unsigned short semval;            /* semaphore text map address */
       short sempid;                     /* pid of last operation */
       unsigned short semncnt;           /* # awaiting semval > cval */
       unsigned short semzcnt;           /* # awaiting semval = 0 */
};
```

Field **sem_perm** is a structure of type **ipc_perm**, which header file **<sys/ipc.h>** defines as follows:

```
struct ipc_perm {
       unsigned short uid;               /* owner's user id */
       unsigned short gid;               /* owner's group id */
       unsigned short cuid;              /* creator's user id */
       unsigned short cgid;              /* creator's group id */
       unsigned short mode;              /* access modes */
       unsigned short seq;               /* slot usage sequence number */
       key_t key;                        /* key */
};
```

semget() initializes **semid_ds** as follows:

- It sets the fields **sem_perm.cuid**, **sem_perm.uid**, **sem_perm.cgid**, and **sem_perm.gid** to, respectively, the effective user and group identifiers of the calling process.
- It sets the low-order nine bits of **sem_perm.mode** to the low-order nine bits of *flag*. These nine bits define access permissions: the top three bits give the owner's access permissions (read, write, execute), the middle three bits the owning group's access permissions, and the low three bits access permissions for others.
- It sets **sem_nsems** to *number*. This gives the number of semaphores to which **sem_base** points.
- It sets field **sem_otime** to zero, and field **sem_ctime** to the current time.

semget() fails if any of the following are true:

- *number* is less than one and the set of semaphores identified by *semkey* does not exist. **semget()** sets **errno** to **EINVAL**.
- *number* exceeds the system-imposed limit (**EINVAL**).
- A semaphore identifier exists for *semkey*, but permission, as specified **flag**'s low-order nine bits, is not granted (**EACCES**).
- A semaphore identifier exists for *semkey*, but the number of semaphores in its set is less than *number*, and *number* does not equal zero (**EINVAL**).
- A semaphore identifier does not exist for *semkey* and (*flag* & **IPC_CREAT**) is false (**ENOENT**).
- **semget()** tried to create a set of semaphores, but could not because the maximum number of sets allowable by the system always exists (**ENOSPC**).

- A semaphore identifier already exists for *semkey* but *flag* requests that **semget()** create an exclusive set for it — i.e.

    ((*flag* & IPC_CREAT) && (*flag* & IPC_EXCL))

  is true (**EEXIST**).

If all goes well, **semget()** returns a semaphore identifier, which is always a non-negative integer. Otherwise, it returns -1 and sets **errno** to an appropriate value.

Files

/usr/include/sys/ipc.h
/usr/include/sys/sem.h

See Also

ftok(), ipcrm, ipcs, libc, libsocket, semctl(), semop()

Notes

Prior to release 4.2, COHERENT implemented semaphores through the driver **sem**. In release 4.2, and subsequent releases, COHERENT has implemented semaphores as a set of functions that conform in large part to the UNIX System-V standard.

The kernel variables **SEMMNI** and **SHMMNS** set, respectively, the maximum number of identifiers that can exist at any given time and the maximum number of semaphores that a set can hold. Daredevil system operators who have large amounts of memory at their disposal may wish to change these variables to increase the system-defined limits. For details on how to do so, see the Lexicon entry **mtune**.

semop() — General Function (libc)

Perform semaphore operations
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>
int semop(*id***,** *operation***,** *nops***)**
int *id***,** *nops***; struct sembuf** *operation***[];**

semop() performs semaphore operations.

id identifies the set of semaphores to be manipulated. It must have been returned by a call to **semget()**.

nops gives the number of structures in the array pointed to by *operation*.

operation points to an array of structures of type **sembuf**, which the header file **sem.h** defines as follows:

```
struct sembuf {
        unsigned short sem_num;                 /* semaphore # */
        short sem_op;                           /* semaphore operation */
        short sem_flg;                          /* operation flags */
};
```

Each **sembuf** describes a semaphore operation. Field **sem_op** identifies the operation to perform on the semaphore in the set identified by *id* and with offset *sem_num*. **sem_op** specifies one of three semaphore operations, as follows:

1. If **sem_op** is negative, one of the following occurs:

   A. If **semval** in the semaphore structure identified by *id* is greater than or equal to the absolute value of **sem_op**, **semop()** subtracts the absolute value of **sem_op** from **semval**.

   B. If **semval** is less than the absolute value of **sem_op** and (**sem_flg** & **IPC_NOWAIT**) is true, **semop()** sets **errno** to **EGAIN** and immediately returns -1.

   C. If **semval** is less than the absolute value of **sem_op** and (**sem_flg** & **IPC_NOWAIT**) is false, then **semop()** increments the **semncnt** associated with the specified semaphore and suspends execution of the calling process until one of the following occurs:

      a. **semval** equals or exceeds the absolute value of **sem_op**. When this occurs, **semop()** decrements the value of **semncnt** associated with the specified semaphore, and subtracts the absolute value of **sem_op** from **semval**.

   - **b.** The *id* for which the calling process is awaiting action is removed from the system.
   - **c.** The calling process receives a signal. When this occurs, **semop()** decrements the field **semncnt** in the **sem** structure that *id* identifies, and the calling process resumes execution in the manner defined by the signal. (See the Lexicon entry for **signal()** for details of what behavior each signal initiates.)

2. If **sem_op** is positive, **semop()** adds **sem_op** to **semval**.

3. If **sem_op** is zero, one of the following occurs:

- **A.** If **semval** is zero, **semop()** returns immediately.
- **B.** If **semval** does not equal zero and (**sem_flg** & **IPC_NOWAIT**) is true, **semop()** sets **errno** to **EGAIN**, and immediately returns -1.
- **C.** If **semval** does not equal zero and (**sem_flg** & **IPC_NOWAIT**) is false, **semop()** increments the **semzcnt** associated with the specified semaphore and suspends execution of the calling process until one of the following occurs:
   - **a.** **semval** becomes zero. **semop()** decrements the value of the field **semzcnt** associated with the specified semaphore.
   - **b.** The set of semaphores identified by *id* is removed from the system.
   - **c.** The calling process receives a signal. **semop()** then decrements the value of the **semzcnt** associated with the specified semaphore, and the calling process resumes execution in the manner prescribed by the signal.

If field **sem_flg** in a **sembuf** structure contains value **SEM_UNDO** (i.e., expression (**sem_flg** & **SEM_UNDO**) is true) then the system stores an *adjust value* for this semaphore operation for this semaphore and links it to the process that has invoked **semop()**. The adjust value is the inversion of this semaphore operation; when the process dies, the system executes these adjust values, to undo each of these semaphore operations. If you use the function **semctl()** to change the value of a semaphore or a set of semaphores, then the system erases all adjust values for those semaphores.

semop() returns -1 and sets **errno** to the value in parentheses if any of the following error conditions occurs:

- *id* is not a valid semaphore identifier (**EINVAL**).
- **sem_num** is less than zero or greater than or equal to the number of semaphores in the set associated with *id* (**EFBIG**).
- *nops* exceeds the system-imposed maximum (**E2BIG**).
- Permission is denied to the calling process (**EACCES**).
- *operation* would suspend the calling process but (**sem_flg** & **IPC_NOWAIT**) is true (**EAGAIN**).
- *operation* would cause **semval** to overflow the system-imposed limit (**ERANGE**).
- *operation* points to an illegal address (**EFAULT**).
- The calling processing receives a signal (**EINTR**).
- The set of semaphores identified by *id* has been removed from the system (**EDOM**).

If all goes well, **semop()** sets the **sempid** of each semaphore specified in the array pointed to by *operation* to the process identifier of the calling process. It then returns the value that **semval** had had at the time that the last operation in the array pointed to by *operation* was executed.

Files

/usr/include/sys/ipc.h
/usr/include/sys/sem.h

See Also

libc, semctl(), semget()

Notes

The COHERENT implementation of semaphores does not permit a process to lock or unlock a semaphore unless it can gain access to all of the semaphores that it requests. This is to prevent the situation in which two processes have each locked semaphores that the other wants, and each has **IPC_NOWAIT** set to false — thus suspending each other forever.

send() — Sockets Function (libsocket)

Send a message to a socket
#include <sys/socket.h>
#include <sys/types.h>
int send(*socket***,** *message***,** *length***,** *flags***)**
int *socket***;**
char **message***;**
int *length***,** *flags***;**

The function **send()** sends a message to a socket.

socket is the socket to which the messages are sent. It must have been created by the function **socket()**, and connected by the function **connect()**. *buffer* points to the chunk of memory that holds the message to be sent; *length* gives the amount of allocated memory to which *buffer* points.

flags ORs together either or both of the following flags:

MSG_OOB
Send "out-of-band" data on sockets that support this notion. The underlying protocol must also support out-of-band data.

MSG_DONTROUTE
The socket turned on for the duration of the operation. It is used only by diagnostic or routing programs.

If all goes well, **send()** returns the number of bytes it sent. If something goes wrong, it returns -1 and sets **errno** to one of the following values:

EAGAIN
If *socket* has no buffer space available, **send()** normally waits until space becomes available (which is a blocking operation). *socket*, however, is marked as non-blocking.

EBADF *socket* does not identify a valid socket.

EINTR A signal interrupted **send()** before it could send any data.

EMSGSIZE
socket requires that message be sent atomically, and the message was too long.

ENOMEM
Insufficient user memory was available to complete the operation.

ENOTSOCK
socket describes a file, not a socket.

EPROTO
A protocol error has occurred.

See Also

connect(), libsocket, recv(), sendto(), socket()

sendto() — Sockets Function (libsocket)

Send a message to a socket
#include <sys/socket.h>
#include <sys/types.h>
int send(*socket***,** *message***,** *length***,** *flags***,** *addr***,** *alen***)**
int *socket***;** char **message***; int** *length***,** *flags***;**
sockaddr_t **addr***; int** *alen***;**

The function **sendto()** sends a message to a socket. Unlike the related function **sendto()**, it works regardless of whether the socket is connected.

socket is the socket to which the messages are sent. It must have been created by the function **socket()**. *buffer* points to the chunk of memory into which the message is to be written; *length* gives the amount of allocated memory to which *buffer* points. If *from* is not NULL, **sendto()** initializes it to the the source address of the message. It initializes *alen* to the size of the buffer associated with *address*, and modifies it upon return to the size of the address stored there.

flags ORs together either or both of the following flags:

MSG_OOB
: Send "out-of-band" data on sockets that support this notion. The underlying protocol must also support out-of-band data.

MSG_DONTROUTE
: The socket turned on for the duration of the operation. It is used only by diagnostic or routing programs.

If all goes well, **sendto()** returns the number of bytes it sent. If something goes wrong, it returns -1 and sets **errno** to one of the following values:

EAGAIN
: If *socket* has no buffer space available, **sendto()** normally waits until space becomes available (which is a blocking operation). *socket*, however, is marked as non-blocking.

EBADF *socket* does not identify a valid socket.

EINTR A signal interrupted **sendto()** before it could send any data.

EMSGSIZE
: *socket* requires that message be sent atomically, and the message was too long.

ENOMEM
: Insufficient user memory was available to complete the operation.

ENOTSOCK
: *socket* describes a file, not a socket.

EPROTO
: A protocol error has occurred.

See Also

connect(), libsocket, recv(), send(), socket()

Notes

At present, the COHERENT implementation of **sendto()** always behaves as if *address* were initialized to NULL.

serialno — System Administration

Hold the serial number of your system
/etc/serialno

The file **/etc/serialno** holds your system's serial number. This is the number assigned to your system when you installed COHERENT onto your computer. You need this number when you update your COHERENT system, or when you contact Mark Williams Company.

See Also

Administering COHERENT

services — System Administration

List supported TCP/IP services
/etc/services

The file **/etc/services** names the services provided by TCP/IP and related protocols.

Each line within this file describes one services. A line consists of four fields, which respectively give the official service name, well-known port number by which it is accessed, the name of its protocol, and any aliases by which it is known. For example:

```
smtp            25/tcp          mail
time            37/tcp          timserver
time            37/udp          timserver
```

Fields are separated by white space, with the exception of fields that give the port and the protocol name; these are separated by virgule '/'. The fourth, aliases field is optional. A pound-sign character '#' introduces a comment; all text from that character to the end of the line is ignored.

See Also

Administering COHERENT, hosts, hosts.equiv, inetd.conf, networks, protocols

set — Command

Set shell option flags and positional parameters
set [-ceiknstuvx [*name ...***]]** (Bourne shell)
set [[+-]aefhkmnuvx] [[+-]o *name***]** (Korn shell)

set changes the options of the current shell and optionally sets the values of positional parameters. This command is used implemented by both the Bourne and Korn shells; however, its syntax and options vary from one shell to the other.

Bourne Shell

The shell variable '$-' contains the currently set shell flags. If the optional *name* list is given, **set** assigns the positional parameters **$1**, **$2** ... to the given shell variables.

set recognizes the following options:

-c *string*
Read shell commands from *string.*

-e Exit on any error (command not found or command returning nonzero status) if the shell is not interactive.

-i The shell is interactive, even if the terminal is not attached to it; print prompt strings. For a shell reading a script, ignore signals **SIGTERM** and **SIGINT.**

-k Place all keyword arguments into the environment. Normally, the shell places only assignments to variables preceding the command into the environment.

-n Read commands but do not execute them.

-s Read commands from the standard input and write shell output to the standard error.

-t Read and execute one command rather than the entire file.

-u If the actual value of a shell variable is blank, report an error rather than substituting the null string.

-v Print each line as it is read.

-x Print each command and its arguments as it is executed.

- Cancel the **-x -v** options.

The shell executes **set** directly.

Korn Shell

set recognizes the following options. Preceding an option with '-' turns on the option; preceding it with '+' turns it off.

-a **allexport**: Automatically export all new variables.

-e **errexit**: Exit from the shell when non-zero status is received.

-f **noglob**: Do not expand file names. This globally turns off the special meaning of characters '*' and '?'.

-h **trackall**: Automatically add all commands to the shell's hash table.

-k **keyword**: Recognize variable assignments anywhere in a command.

-m **monitor**: Enable job control. See the Lexicon article on **ksh** for details on job control and how to use it.

-n **noexec**: Compile an input command, but do not execute it.

-o *option*
Set *option*. **set** recognizes the following *options*:

allexport	Same as **-a** option, above.
emacs	Turn on MicroEMACS-style editing of command lines.
errexit	Same as **-e** option, above.
ignoreeof	Tell the shell not to exit when reading EOF: must use **exit** command to exit from the shell.
keyword	Same as **-k** option, above.
monitor	Same as **-m** option, above.
noexec	Same as **-n** option, above.
noglob	Same as **-f** option, above.
trackall	Same as **-h** option, above.
nounset	Same as **-u** option, below.
verbose	Same as **-v** option, below.
xtrace	Same as **-x** option, below.

-u **nounset**: Treat dollar-sign expansion of an unset variable as an error.

-v **verbose**: When compiling a command, echo its compiled (i.e., expanded) version on the standard output before executing it.

-x **xtrace**: Echo simple commands while executing.

The shells execute **set** directly.

See Also

commands, ksh, sh, unset

setbuf() — STDIO Function (libc)

Set alternative stream buffer
#include <stdio.h>
void setbuf(*fp*, *buffer***)**
FILE **fp***; char** **buffer***;**

The standard I/O library STDIO automatically buffers all data read and written in streams, with the exception of streams to terminal devices. STDIO normally uses **malloc()** to allocate the buffer, which is a **char** array **BUFSIZ** characters long; **BUFSIZ** is a manifest constant defined in the header file **stdio.h**.

setbuf()'s arguments are the file stream *fp* and the *buffer* to be associated with the stream. The call should be issued after the stream has been opened, but before any input or output request has been issued. If *buffer* is NULL, the stream will be unbuffered. If *buffer* is not NULL, the arena of memory it points to must contain at least **BUFSIZ** bytes.

setbuf() returns nothing.

See Also

fopen(), libc, malloc(), setvbuf()
ANSI Standard, §7.9.5.5
POSIX Standard, §8.1

setgid() — System Call (libc)

Set group id and user id
#include <unistd.h>
int setgid(*id***) int** *id***;**

The *group identifier* is the number that identifies the user group that “owns” a given file. File **/etc/group** establishes the set of groups that your COHERENT system recognizes. (For details on how this file is laid out, see the Lexicon entry for **group**). When a file is executable, the executing process inherits its group identifier (and thus, its group-level permissions) from the file in which it resides on disk. For example, the program **troff** resides in file **/bin/troff**. This file is “owned” by group **bin**; thus, when you execute **troff**, its group-level permissions are those of group **bin**.

The group identifier comes in three forms:

real This is the group identifier of the user who is running the process.

effective
This is the group identifier that determines the access rights of the process. These rights are the same as those of the real group identifier unless they have been altered by executing a file whose **setgid** bit is set. For example, the program **troff** does *not* have the setgid bit set; thus, when you execute **troff**, the group permissions of the **troff** process remain those of your group, not those of the group **bin**. On the other hand, the program **/usr/lib/uucp/uucico** does have the setgid bit set; thus, when you invoke **uucico**, the **uucico** process uses the permissions of **uucico**'s group (that is, of group **uucp**), instead of your group.

saved effective
This permits a process to step back and forth between its real and effective group identifiers. If you return from an effective group identifier to your real one, the system saves the previously effective identifier so you can revert to it if need be.

The system call **setgid()** lets you set the real and effective group identifiers of the calling process to the group identifier *gid*. The behavior of **setgid()** varies depending upon the following:

1. If **setgid()** is invoked by a user whose effective user identifier is that of the superuser **root**, **setgid()** sets the real, effective, and saved effective group identifiers to *gid*.
2. If **setgid()** is invoked by a user whose real group identifier is the same as *gid*, **setgid()** sets the effective group identifier to *gid*.
3. If **setgid()** is invoked by a user whose saved effective group identifier is same as *gid*, **setgid()** sets the effective group identifier to *gid*.

If all goes well, **setgid()** returns zero. If a problem arises, it returns -1.

See Also

execution, getuid(), libc, login, setuid(), unistd.h
POSIX Standard, §4.2.2

setgrent() — General Function (libc)

Rewind group file
#include <grp.h>
void setgrent();

setgrent() rewinds the file **/etc/group**. It returns nothing.

Files

/etc/group
<grp.h>

See Also

group, libc

setgroups() — System Call (libc)

Set the supplemental group-access list
#include <unistd.h>
int setgroups(*ngroups***,** *grouplist***)**
int *ngroups***; const gid_t** **grouplist***;**

The "supplemental group-access list" is the list of group identifiers that are used in addition to the effective group identifier when determining the level of access that a process has to a file. **setgroups()** fills the calling process's supplemental group-access list with the group identifiers in the array to which *grouplist* points. *ngroups* gives the number of identifiers in the array, and cannot exceed **NGROUPS_MAX**.

If all goes well, **setgroups()** returns zero. It fails and returns -1 if any of the following occur:

- The value of *ngroups* exceeds **NGROUPS_MAX**. **setgroups** sets **errno** to **EINVAL**.
- The effective user identifier is not that of the super-user **root**. **setgroups()** sets **errno** to **EPERM**.

- *grouplist* contains an illegal address. **setgroups()** sets **errno** to **EFAULT**.

See Also

getgroups(), initgroups(), libc, limits.h, unistd.h

Notes

This function may be invoked only by the superuser **root**.

sethostent() — Sockets Function (libsocket)

Open and rewind file /etc/hosts
#include <netdb.h>
void sethostent(*stayopen***)**
int *stayopen***;**

The function **sethostent()** is one of a set of functions that interrogate the file **/etc/hosts** to look up information about a remote host on a network. It opens and rewinds **/etc/hosts**.

Flag *stayopen* indicates whether **/etc/hosts** is to stay open after it has been interrogated by **gethostbyaddr()** or **gethostbyname()**: if it is zero, then **/etc/hosts** is closed after it is interrogated; if it is nonzero, then **/etc/hosts** remains open.

See Also

endhostent(), gethostbyaddr(), gethostbyname(), libsocket

setjmp() — General Function (libc)

Save machine state for non-local goto
#include <setjmp.h>
int setjmp(*env***) jmp_buf** *env***;**

The function call is the only mechanism that C provides to transfer control between functions. This mechanism, however, is inadequate for some purposes, such as handling unexpected errors or interrupts at lower levels of a program. To answer this need, **setjmp** helps to provide a non-local *goto* facility. **setjmp()** saves a stack context in *env*, and returns value zero. The stack context can be restored with the function **longjmp()**. The type declaration for **jmp_buf** is in the header file **setjmp.h**. The context saved includes the program counter, stack pointer, and stack frame.

Example

The following gives a simple example of **setjmp()** and **longjmp()**.

```
#include <setjmp.h>
#include <stdio.h>

jmp_buf env;      /* place for setjmp to store its environment */

main()
{
      int rc;

      if(rc = setjmp(env)) { /* we come here on return */
            printf("First char was %c\n", rc);
            exit(EXIT_SUCCESS);
      }
      subfun();   /* this never returns */
}

subfun()
{
      char buf[80];

      do {
            printf("Enter some data\n");
            gets(buf);  /* get data */
      } while(!buf[0]); /* retry on null line */

      longjmp(env, buf[0]);   /* buf[0] must be non zero */
}
```

See Also

getenv(), libc, longjmp(), sigsetjmp()
ANSI Standard, §7.6.1.1
POSIX Standard, §8.1

Notes

Programmers should note that many user-level routines cannot be interrupted and reentered safely. For that reason, improper use of **setjmp()** and **longjmp()** can create mysterious and irreproducible bugs. The use of **longjmp()** to exit interrupt exception or signal handlers is particularly hazardous.

setjmp.h — Header File

Define setjmp() and longjmp()
#include <setjmp.h>

setjmp.h defines the structure **jmp_buf** for a **setjmp()** environment.

See Also

header file, longjmp(), setjmp()
ANSI Standard, §7.6

setnetent() — Sockets Function (libsocket)

Open and rewind file /etc/networks

#include <netdb.h> int setnetent(*stayopen***) int** *stayopen***;**

Function **setnetent()** opens or rewinds file **/etc/networks**, which describes all entities on your local network. If flag *stayopen* is set to a non-zero value, **/etc/networks** is not closed after each call to **getnetbyaddr()** or **getnetbyname()**.

See Also

endnetent(), getnetbyaddr(), getnetbyname(), getnetent(), libsocket, netdb.h

setpgid() — System Call (libc)

Set the process-group identifier
#include <sys/types.h>
#include <unistd.h>
int setpgid(*pid***,** *pgid***)**
pid_t *pid***,** *pgid***;**

setpgid() sets to *pgid* the process-group identifier of the process with identifier *pid*. If *pgid* equals *pid*, the process becomes a process-group leader. If *pgid* does not equal *pid*, the process becomes a member of an existing process group.

If *pid* equals zero, **setpgid()** uses the process identifier of the calling process. If *pgid* equals zero, the process specified by *pid* becomes a process-group leader.

If all goes well, **setpgid()** returns a value of zero. Otherwise, it returns -1 and sets **errno** to an appropriate value. **setpgid()** if any of the following are true:

- *pid* matches the process identifier of a child process of the calling process, and that child process has successfully executed an **exec()** function. **setpgid()** sets **errno** to **EACCES**.
- *pgid* is less than zero or greater than or equal to **PID_MAX**. **setpgid()** sets **errno** to **EINVAL**.
- The calling process has a controlling terminal that does not support job control. **setpgid()** sets **errno** to **EINVAL**.
- The process identified by *pid* argument is a session leader. **setpgid()** sets **errno** to **EPERM**.
- *pid* equals the process identifier of a child process of the calling process, and the child process is not in the same session as the calling process. **setpgid()** sets **errno** to **EPERM**.
- *pgid* does not match the process identifier of the process indicated by *pid*, and the call process's session has no process with a process-group identifier that equals *pgid*. **setpgid()** sets **errno** to **EPERM**.

- *pid* does not match the process identifier of the calling process or of a child process of the calling process. **setpgid()** sets **errno** to **ESRCH**.

See Also

libc, unistd.h
POSIX Standard, §4.3.3

setpgrp() — System Call (libc)

Make a process a process-group leader
int setpgrp()

setpgrp() sets the requesting process's process-group identifier to its own process identifier. This detaches the process from its parent group and makes it the leader of its own processing group. If the process is not already a process-group leader, it is detached from its controlling terminal.

setpgrp() returns the new process-group identifier.

See Also

getpgrp(), libc

Notes

This function is obsolete, and is being phased out in favor of the function **setsid()**.

setprotoent() — Sockets Function (libsocket)

Open the protocols file

#include <netdb.h> int setprotoent(*stayopen***) int** *stayopen***;**

Function **setprotoent()** opens or rewinds file **/etc/protocols**, which describes all protocols recognized on your local network. If flag *stayopen* is set to a non-zero value, **/etc/protocols** is not closed after each call to **getprotobyaddr()** or **getprotobyname()**.

See Also

getprotobyaddr(), getprotobyname(), getprotoent(), endprotoent(), libsocket, netdb.h

setpwent() — General Function (libc)

Rewind password file
#include <pwd.h>
setpwent()

The COHERENT system has five routines that search the file **/etc/passwd**, which contains information about every user of the system. **setpwent()** rewinds the password file, which allows searches to begin from the beginning of the file. Please note that this function does not return a meaningful value.

Example

For an example of this function, see the entry for **getpwent()**.

Files

/etc/passwd
pwd.h

See Also

libc

setservent() — Sockets Function (libsocket)

Open the services file

#include <netdb.h> int setservent(*stayopen***) int** *stayopen***;**

Function **setservent()** opens or rewinds file **/etc/services**, which describes the services offered by TCP/IP on your local network. If flag *stayopen* is set to a non-zero value, **/etc/services** is not closed after each call to **getservbyport()** or **getservbyname()**.

See Also

getservbyname(), getservbyport(), getservent(), endservent(), libsocket, netdb.h

setsid() — System Call (libc)

Set session identifier
#include <sys/types.h>
#include <unistd.h>
pid_t setsid ( );

If the calling process is not a process-group leader, **setsid()** sets its process-group and session identifiers to its process identifier, and releases the its controlling terminal.

If all goes well, **setsid()** returns the calling process's session identifier. If the calling process is already a process-group leader, or if process-group identifier of another process equals that of the calling process, **setsid()** returns -1 and sets **errno** to **EPERM**.

See Also

libc, unistd.h
POSIX Standard, §4.3.2

Notes

If the calling process is the last member of a pipeline started by a job-control shell, the shell may make the calling process a process-group leader. The other processes of the pipeline become members of that process group. If this happens, the call to **setsid()** fails.

For this reason, a process that calls **setsid()** and expects to be part of a pipeline should first fork: the parent should exit and the child should call **setsid()**. This will ensure that the process works reliably when started by both job-control shells and non-job-control shells.

setsockopt() — Sockets Function (libsocket)

Set a socket option
#include <sys/types.h>
#include <sys/socket.h>
int setsockopt(*socket***,** *level***,** *option***,** *buffer***,** *length***)**
int *socket***,** *level***,** *option***,** *length***;**
char **buffer***;**

Function **setsockopt()** sets options on a socket.

socket gives the identifier of the socket, as returned by the function **socket()**.

level gives the level at which the options are set. To retrieve options set on the socket level, set *level* to **SOL_SOCKET** whereas to retrieve options set the TCP level, set *level* to the number of the TCP protocol.

option gives the number of the option to set. A list of options that are recognized at the socket level appears below. Options at other levels are set by their respective protocols.

buffer gives the address of the buffer that holds the option. *length* gives the length of *buffer*, in bytes.

The following options are recognized at the socket level. They are set in header file **<sys/socket.h>**:

SO_BROADCAST
Toggle permission to transmit broadcast messages.

SO_KEEPALIVE
Toggle whether to keep a connection alive by periodically transmitting messages. If the connected party fails to respond to a message, the connection is considered broken and processes that use the socket are notified via the signal **SIGPIPE**.

SO_LINGER
Control the action taken when a socket is closed but contains unsent messages. If **SO_LINGER** is set and the socket promises reliable delivery of data, the system blocks the process that is attempting to close *socket* until *socket* can transmit its data or its attempts to do so time out. If **SO_LINGER** is not enabled, the socket is closed immediately and the unsent messages are thrown away.

SO_OOBINLINE
Toggle whether a band can receive out-of-band data. Such data can then be read by the function **recv()** or sent by the function **send()**, when invoked with the flag **MSG_OOB**.

SO_RCVBUF
SO_SNDBUF
Set the size of the receive or send buffer, respectively. You can increase the size of a buffer to speed high-volume connections, or decrease it to limit the amount of data that are backlogged. The system places an absolute limit on these values.

SO_REUSEADDR
Toggle whether local addresses can be reused.

If all goes well, **setsockopt()** returns zero. If something goes wrong, it returns -1 and set **errno** to one of the following values:

EBADF *socket* does not identify a valid socket.

ENOMEM
The available user memory was insufficient to complete the operation.

ENOPROTOOPT
option gives an unknown option.

ENOTSOCK
socket identifies a file, not a socket.

See Also

getsockopt(), libsocket

setspent() — General Function (libc)

Rewind the shadow-password file
#include <shadow.h>
setspent()

The COHERENT system has four routines that search the file **/etc/shadow**, which contains the password of every user of the system. **setspent()** rewinds the password file — that is, it resets the seek pointer so that subsequent searches of the file start at the beginning of the file. This function does not return a meaningful value.

See Also

endspent(), getspent(), libc, shadow, shadow.h

setuid() — System Call (libc)

Set user identifier
#include <unistd.h>
int setuid(*id***)**
int *id***;**

The *user identifier* is the number that identifies the user who “owns” a given file. The suite of users is defined in file **/etc/passwd**. When a file is executable, the executing process inherits its user identifier (and thus, its user-level permissions) from the file where it lives on disk. The user identifier comes in three forms:

real This is the identifier of the user who is running the process.

effective
This is the user identifier that determines the access rights of the process. These rights are the same as those of the real user identifier unless they have been altered by executing a file whose **setuid** bit is set.

saved effective
This permits a process to step back and forth between its real and effective user identifiers. If you return from an effective user identifier to your real one, the previously effective id is saved so you can revert to it if need be.

The system call **setuid()** lets you alter the real and effective user identifiers of the calling process to the user identifier *uid*. The behavior of **setuid()** varies depending upon the following:

1. If the effective user identifier is that of the superuser, **setuid()** sets the real, effective, and saved effective user identifiers to *uid*.
2. If the real user identifier is the same as *uid*, **setuid()** sets the effective user identifier to *uid*.
3. If the saved effective user identifier is same as *uid*, **setuid()** sets the effective user identifier to *uid*.

To **setuid** an existing executable file, use the command **chmod**.

See Also

chmod, execution, getuid(), libc, login, setgid(), unistd.h
POSIX Standard, §4.2.2

Diagnostics

setuid() returns zero on success, or -1 on failure.

Notes

For more information on the user id, see the Lexicon entry for **execution**.

setupterm() — terminfo Function

Initialize a terminal
#include <curses.h>
setupterm(*term***,***fd***,***errret***)**
char **term***;**
int *fd***,** **errret***;**

COHERENT comes with a set of functions that let you use **terminfo** descriptions to manipulate a terminal. **setupterm()** initializes terminal capabilities for terminal type *term*, which is accessed via file-descriptor *fd*. It inhales all capabilities at once, and performs all other system-dependent initialization — which is one reason why **terminfo** is much faster than **termcap**.

If *term* is initialized to NULL, **setupterm()** uses the contents of the environmental variable **TERM** as a default.

errret points to an integer into which **setupterm()** writes the terminal's status: zero if there is no such terminal type, one if all went well, or -1 if something has gone wrong. If *erret* is NULL, **setupterm()** prints an error message and exits if the terminal cannot be found.

See Also

terminfo

setutent() — General Function (libc)

Rewind the input stream for a login logging file
#include <utmp.h>
void setutent()

Function **setutent()** rewinds the input stream that is reads the file that records login events. This lets you search this file multiple times without having to close and reopen it.

By default, **setutent()** manipulates a stream that reads file **/etc/utmp**. If you wish to manipulate another logging file, use the function **utmpname()**.

See Also

libc, utmp.h

setvbuf() — STDIO Function (libc)

Set alternative file-stream buffer
#include <stdio.h>
int setvbuf(*fp***,** *buffer***,** *mode***,** *size***)**
FILE **fp***;** **char** **buffer***;** **int** *mode***;** **size_t** *size***;**

When the functions **fopen()** and **freopen()** open a stream, they automatically establish a buffer for it. The buffer is **BUFSIZ** bytes long. **BUFSIZ** is a manifest constant that is defined in the header **stdio.h**.

The function **setvbuf()** alters the buffer used with the stream pointed to by *fp* from its default buffer to *buffer*. Unlike the related function **setbuf()**, it also allows you set the size of the new buffer as well as the form of buffering.

buffer is the address of the new buffer. *size* is its size, in bytes. *mode* is the manner in which you wish the stream to be buffered, as follows:

_IOFBF	Fully buffered
_IOLBF	Line-buffered
_IONBF	No buffering

These constants are defined in the header **stdio.h**.

You should call **setvbuf()** after a stream has been opened but before any data have been written to or read from the stream. For example, the following give *fp* a 50-byte buffer that is line-buffered:

```
char buffer[50];
FILE *fp;

fopen(fp, "r");
setvbuf(fp, buffer, _IOLBF, sizeof(buffer));
```

On the other hand, the following turns off buffering for the standard output stream:

```
setvbuf(stdout, NULL, _IONBF, 0);
```

setvbuf() returns zero if the new buffer could be established correctly. It returns a value other than zero if something went wrong or if an invalid parameter is given for *mode* or *size*.

Example

This example uses **setvbuf()** to turn off buffering and echo.

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

main(void)
{
      int c;

      if(setvbuf(stdin,  NULL, _IONBF, 0))
            fprintf(stderr, "Couldn't turn off stdin buffer\n");

      if(setvbuf(stdout,  NULL, _IONBF, 0))
            fprintf(stderr, "Couldn't turn off stdout buffer\n");

      while((c = getchar()) != EOF)
            putchar(c);
}
```

See Also

fclose(), fflush(), fopen(), freopen(), libc, setbuf()
ANSI Standard, §7.9.5.6

Notes

setvbuf() affects the buffering of an I/O stream but does not affect any buffering that performed by the device upon which the text is typed. Some devices (e.g., **/dev/tty**) are buffered by default. To turn off the buffering of what a user types, you must both turn off buffering on the input stream and turn off buffering on the device itself. For example, to turn off buffering on a terminal device, you must both call **setvbuf()** to change the size of the input buffering to zero, and call **stty()** to put the terminal device into raw mode.

sgtty — Device Driver

General terminal interface

COHERENT uses two method for controlling terminals: **sgtty** and **termio**. To use **sgtty**, simply include the statement **#include <sgtty.h>** in your sources. To use **termio**, include the statement **#include <termio.h>**.

The rest of this article discusses the **sgtty** method of controlling terminals.

When a terminal file is opened, it normally causes the process to wait until a connection is established. In practice, users' programs seldom open these files; they are opened by the program *getty* and become a user's standard input, output, and error files. The very first terminal file opened by the process group leader of a terminal file not already associated with a process group becomes the *controlling terminal* for that process group. The controlling terminal plays a special role in handling **quit** and **interrupt** signals, as discussed below. The controlling terminal is inherited by a child process during a call to **fork**. A process can break this association by changing its process group using **setpgrp**.

A terminal associated with one of these files ordinarily operates in full-duplex mode. Characters can be typed at any time, even while output is occurring, and are only lost when the system's input buffers become completely full, which is rare, or when the user has accumulated the maximum allowed number of input characters that have not yet been read by some program. Currently, this limit is 256 characters. When the input limit is reached, the sytems throws away all the saved characters without notice.

Normally, terminal input is processed in units of lines. A line is delimited by a newline character (ASCII LF) or an end-of-file character (ASCII EOT). Unless otherwise directed, a program attempting to read will be suspended until an entire line has been typed. Also, no matter how many characters are requested in the read call, at most one line will be returned. It is not, however, necessary to read a whole line at once; any number of characters may be requested in a read, even one, without losing information.

During input, the system normally processes **erase** and **kill** characters. By default, the backspace character erases the last character typed, except that it will not erase beyond the beginning of the line. By default, the **<ctrl-U>** kills (deletes) the entire input line, and optionally outputs a newline character. Both these characters operate on a keystroke-by-keystroke basis, independently of any backspacing or tabbing which may have been done. Both the erase and kill characters may be entered literally by preceding them with the escape character (\). In this case, the escape character is not read. You may change the erase and kill characters via command **stty**.

Certain characters have special functions on input. These functions and their default character values are summarized as follows:

INTR (**<ctrl-C>** or ASCII ETX) generates an *interrupt* signal that is sent to all processes associated with the controlling terminal. Normally, each such process is forced to terminate, but arrangements may be made either to ignore the signal or to receive a trap to an agreed-upon location; see the Lexicon entry for **signal**.

QUIT (Control-\ or ASCII ES) generates a *quit* signal. Its treatment is identical to that of the interrupt signal except that, unless a receiving process has made other arrangements, it will not only be terminated but a core image file (called **core**) will be created in the current working directory.

ERASE (**<backspace>** or ASCII BS) erases the preceding character. It will not erase beyond the start of a line, as delimited by a newline or **EOF** character.

KILL (**<ctrl-U>** or ASCII NAK) deletes the entire line, as delimited by a newline or **EOF** character.

EOF (**<ctrl-D>** or ASCII EOT) generates an end-of-file character from a terminal. When received, all the characters waiting to be read are immediately passed to the program without waiting for a newline, and the **EOF** is discarded. Thus, if no characters are waiting, which is to say the **EOF** occurred at the beginning of a line. zero characters will be passed back, which is the standard end-of-file indication.

NL (ASCII LF) is the normal line delimiter. It cannot be changed or escaped.

STOP (**<ctrl-S>** or ASCII DC3) can be used to suspend output. It is useful with CRT terminals to prevent output from disappearing before it can be read. While output is suspended, STOP characters are ignored and not read.

START (**<ctrl-Q>** or ASCII DC1) resumes output that has been suspended by a STOP character. While output is not suspended, START characters are ignored and not read. The start/stop characters can be changed via command **stty**, or via special **ioctl()** calls described below.

The character values for INTR, QUIT, ERASE, **EOF**, and KILL may be changed to suit individual tastes. The ERASE, KILL, and **EOF** character may be escaped by a preceding \ character, in which case the system ignores its special meaning. See the Lexicon article on **stty** for information on how to change these settings dynamically.

When using a "modem control" serial line, loss of carrier from the data-set (modem) causes a *hangup* signal to be sent to all processes that have this terminal as the controlling terminal. Unless other arrangements have been made, this signal causes the process to terminate. If the hangup signal is ignored, any subsequent read returns with an end-of-file indication. Thus programs that read a terminal and test for end-of-file can terminate appropriately when hung up on.

When one or more characters are written, they are transmitted to the terminal as soon as previously written characters have finished typing. Input characters are echoed by putting them into the output queue as they arrive. If a process produces characters more rapidly than they can be printed, it will be suspended when its output queue exceeds some limit, known as the “high water mark”. When the queue has “drained” down to some threshold, the program resumes.

The header file **<sgtty.h>** declares structures and manifest constants to control the **sgtty** interface. Of interest to users are the constants that define baud rates for terminal ports; these are as follows.

B50	50 baud
B75	75 baud
B110	110 baud
B134	134 baud
B150	150 baud
B200	200 baud
B300	300 baud
B600	600 baud
B1200	1200 baud
B1800	1800 baud
B2400	2400 baud
B4800	4800 baud
B9600	9600 baud
B19200	19,200 baud
B38400	38,400 baud

Terminal ioctl() Functions

Header file **<sgtty.h>** defines the following data structures used by the various device drivers to convey terminal specific information. These structures are used in conjunction with special terminal or device driver symbolic constants as part of **ioctl()** requests.

The **sgttyb** structure contains information related to line discipline, such as serial line speed, if appropriate, the “erase” and “kill” characters, and a series of flags which set the mode of the line.

```
/*
 * Structure for TIOCSETP/TIOCGETP
 */
struct sgttyb {
	char	sg_ispeed;	/* Input speed */
	char	sg_ospeed;	/* Output speed */
	char	sg_erase;	/* Character erase */
	char	sg_kill;	/* Line kill character */
	int	sg_flags;	/* Flags */
};
```

The following symbolic constants are used to access bit positions of member **sg_flags** in data structure **sgttyb**:

CBREAK Each input character causes wakeup (i.e., forces a return from a **read()** system call).

CRMOD Map the carriage return characters '\r' to the newline character '\n'.

CRT Use CRT-style character erase.

ECHO Echo input characters.

EVENP Select even parity. If used in conjunction with **ODDP**, allow either parity.

LCASE Lowercase mapping on input.

ODDP Select odd parity. If used in conjunction with **EVENP**, allow either parity.

RAW Raw mode. Same as **RAWIN** plus **RAWOUT**.

RAWIN Input is treated as 8-bit characters and not interpreted.

RAWOUT Output is treated as 8-bit characters and not interpreted.

TANDEM Use X-ON/X-OFF flow control protocol to remote device.

XTABS Expand tabs to spaces.

Data structure **tchars** specifies additional special terminal characters such as the “interrupt” and “quit” characters, the “start” and “stop” characters used for flow control, and the “end-of-file” character.

```
/*
 * Structure for TIOCSETC/TIOCGETC
 */
struct tchars {
	char	t_intrc;	/* Interrupt */
	char	t_quitc;	/* Quit */
	char	t_startc;	/* Start output */
	char	t_stopc;	/* Stop output */
	char	t_eofc;		/* End of file */
	char	t_brkc;		/* Input delimiter */
};
```

The following symbolic constants are used to access various device functions via **ioctl()** calls, as defined in header file **<sgtty.h>**. Note that not all functions are appropriate for all classes of devices.

TIOCCBRK Clear a BREAK condition on a serial line (i.e., “mark” the line). This request cancels a previously issued **TIOCSBRK** request.

TIOCCDTR Clear modem control signal Data Terminal Ready (DTR) on a serial line.

TIOCCHPCL Do not force a hangup on “last close” on a modem line. The normal mode of operation for serial lines is to drop modem signal Data Terminal Ready (DTR) when the last **close()** operation is performed, thus requesting the attached modem to drop the connection.

TIOCCRTS Clear the Request To Send (RTS) signal on a serial line. Modem control signal RTS is often used for hardware flow control.

TIOCEXCL Set device access as exclusive use. This request requires the process to have **root** privileges.

TIOCFLUSH Flush the input queue, discarding any pending input characters, and wait for the output queue to “drain”.

TIOCGETC Get current values of the special terminal characters, as defined by data structure **tchars**.

TIOCGETF Get current console keyboard function key bindings. This request is specific to the **nkb** console keyboard device driver. See Lexicon article **nkb** for further details.

TIOCGETKBT Get current console keyboard key mapping table. This request is specific to the **nkb** console keyboard device driver. See Lexicon article **nkb** for further details.

TIOCGETP Get current terminal line settings, as defined by data structure **sgttyb**.

TIOCGETTF Get current value of the terminal flags, as defined by field **t_flags** in the TTY structure.

TIOCHPCL Set hangup on “last close”. See **TIOCCHPCL** for further details.

TIOCRMSR Get the current value of the Modem Status Register (MSR) for the specified serial line. This request is device driver specific and is currently supported only in the **al** device driver. Symbolic constants **MSRCTS**, **MSRDSR**, **MSRRI**, and **MSRRLSD** correspond to the Clear To Send, Data Set Ready, Ring Indicator and Receive Line Status Detect (i.e. Carrier Detect) signals, respectively, in the MSR.

TIOCNXCL Set this device or port as non-exclusive use. See **TIOCEXCL** for further details.

TIOCQUERY Query the number of characters currently waiting in the input queue.

TIOCSBRK Assert BREAK (i.e., “space the line”) on the given serial port. This is often used during login to signal a remote system to “hunt” to the next baud rate in a sequence. See **TIOCCBRK** for further details.

TIOCSDTR Assert modem control signal Data Terminal Ready (DTR) on a serial line.

TIOCSETC Wait for output to "drain", then set the terminal control characters for this device, as specified by data structure **tchars**.

TIOCSETF Set console keyboard function key mapping. This request is specific to the **nkb** console keyboard device driver. See Lexicon article **nkb** for further details.

TIOCSETKBT Set console keyboard key mapping table. This request is specific to the **nkb** console keyboard device driver. See Lexicon article **nkb** for further details.

TIOCSETN Set terminal line settings, as defined by data structure **sgttyb**. Do not flush the input queue prior to using the new settings.

TIOCSETP Same as request **TIOCSETN**, but also flush the input queue.

TIOCSRTS Assert the Request To Send (RTS) signal on a serial line. Modem control signal RTS is often used for hardware flow control.

Examples

The following code fragment gets the current terminal settings and turns off echo.

```
#include <sgtty.h>
static struct sgttyb new, orig;
. . .
/*
 * Get the existing terminal parameters for the terminal
 * device associated with file descriptor 0 (stdin),
 * turn off echo, turn on CBREAK (break on every input character)
 * and set the new parameters.
 */
ioctl(0, TIOCGETP, &orig);
new = orig;
new.sg_flags &= ~ECHO;           /* Turn off echo */
new.sg_flags |= CBREAK;          /* Turn on CBREAK mode */
ioctl(0, TIOCSETN, &new);
```

The following line uses the previously saved terminal mode to return the terminal mode to its prior state:

```
ioctl(0, TIOCSETN, &orig);
```

See Also

device drivers, gtty(), ioctl(), sgtty.h, stty, stty(), terminal, termio

sgtty.h — Header File

Definitions used to control terminal I/O
#include <sgtty.h>

sgtty.h defines structures, constants, and macros used by routines that use the **sgtty** method to control terminal I/O.

See Also

header files, sgtty

sh — Command

The Bourne shell
sh [-ceiknstuvx] *token ...*

The COHERENT system offers two command interpreters: **ksh**, the Korn shell; and **sh**, the Bourne shell. **sh** is the default COHERENT command interpreter. The tutorial included in this manual describes the Bourne shell in detail.

As you will see from the following description, a shell is both a command interpreter and a programming language in its own right. It would be worth your while to spend some time in learning the rudiment's of the shell's programming language; doing so will help you to use your COHERENT system to best advantage.

Commands

A *command* consists of one or more tokens. A *token* is a string of text characters (i.e., one or more alphabetic characters, punctuation marks, and numerals) delineated by spaces, tabs, or newlines.

A *simple command* consists of the command's name, followed by zero or more tokens that represent arguments to the command, names of files, or shell operators. A *complex command* uses shell constructs to execute one or more commands conditionally. In effect, a complex command is a mini-program that you write in the shell's programming language and that **sh** interprets.

Shell Operators

sh recognizes a number of operators that form pipes, that redirect input and output to commands, and that let you define the conditions under which a given command are executed.

command **|** *command*
The *pipe* operator: let the output of one command serve as the input to a second. You can combine commands with '|' to form *pipelines*. A pipeline passes the standard output of the first (leftmost) command to the standard input of the second command. For example, in the pipeline

```
sort customers | uniq | more
```

sh invokes **sort** to sort the contents of file **customers**. It then pipes the output of **sort** into the command **uniq**, which outputs one unique copy of the text that is input into it. **sh** then pipes the output of **uniq** to the command **more**, which displays it on your terminal one screenful at a time. Note that under COHERENT, unlike MS-DOS, pipes are executed concurrently: that is, **sort** does not have to finish its work before **uniq** and **more** can begin to receive input and go to work.

command **;** *command*
Execute commands on a command line sequentially. The command to the left of the ';' executes to completion; then the command to the right of it executes. For example, in the command line

```
a | b ; c | d
```

first executes the pipeline **a | b** then, when **a** and **b** are finished, executes the pipeline **c | d**.

command **&**
Execute a command in the background. This operator must follow the command, not precede it. It prints the process identifier of the command on the standard output, so you can use the **kill** command to kill that process should something go wrong. This operator lets you execute more than one command simultaneously. For example, the command

```
fdformat -v /dev/fha0 &
```

formats a high-density, 5.25-inch floppy disk in drive 0 (that is, drive A); but while the disk is being formatted, **sh** returns the command line prompt so you can immediately enter another command and begin to work. If you did not use the '&' in this command, you would have to wait until formatting was finished before you could enter another command.

command **&&** *command*
Execute a command upon success. **sh** executes the command that follows the token '&&' only if the command that precedes it returns a zero exit status, which signifies success. For example, the command

```
cd /etc
fdformat -v /dev/fha0 && badscan -o proto /dev/fha0 2400
```

formats a floppy disk, as described above. If the format was successful, it then invokes the command **badscan** to scan the disk for bad blocks; if it was not successful, however, it does nothing.

command **||** *command*
Execute a command upon failure. This is identical to operator '&&', except that the second command is executed if the first returns a non-zero status, which signifies failure. For example, the command

```
/etc/fdformat -v /dev/fha0 || echo "Format failed!"
```

formats a floppy disk. If formatting failed, it echoes the message **Format failed!** on your terminal; however, if formatting succeeds, it does nothing.

Note that the tokens newline, ';' and '&' bind less tightly than '&&' and '||'. **sh** parses command lines from left to right if separators bind equally.

>*file* Redirect into *file* all text written to the standard output, which normally is written onto your screen. For example, the command

```
sort customers >customers.sort
```

sorts file **customers** and writes the sorted output into file **customers.sort**. **sh** creates **customers.sort** if it does not exist, and destroys its previous contents if it does exist.

>>*file* Append onto *file* all text written to the standard output, which normally is written onto your screen. If *file* does not exist, **sh** creates it; however, if the file already exists, **sh** appends the output onto its contents rather than destroying them. For example, the command

```
sort customers.now | uniq >>customers.all
```

sorts file **customers.now**, pipes its output to command **uniq**, which throws away duplicate lines of input, and appends the results onto file **customers.all**.

<*file* Redirect input. Here, **sh** reads the contents of a file and processes them as if you had typed them from your keyboard. For example, the command

```
ed textfile <edit.script
```

invokes the line editor **ed** to edit **textfile**; however, instead of reading editing commands from your keyboard, **sh** passes to **ed** the contents of file **edit.script**. This command would let you prepare an editing script that you could execute repeatedly upon files rather than having to type the same commands over and over.

<< *token*
Prepare a “here document”. This operator tells **sh** to accept standard input from the shell input until it reads the next line that contains only *token*. For example, the command

```
cat >FOO <<\!
      Here is some text.
!
```

redirects all text between ‘<<\!’ and ‘!’ to the **cat** command. The operator ‘>’ in turn redirects the output of **cat** into file **FOO**. **sh** performs parameter substitution on the here document unless the leading *token* is quoted; parameter substitution and quoting are described below.

command **2>** *file*
Redirect into *file* all text written to the standard error, which normally is written onto your screen. For example, the command

```
nroff -ms textfile >textfile.p 2>textfile.err
```

invokes the command **nroff** to format the contents of **textfile**. It redirects the output of **nroff** (i.e., the standard output) into **textfile.p**; it also redirects any error messages that **nroff** may generate into file **textfile.err**.

Please note that a command may use up to 20 streams. By default, stream 0 is the standard input; stream 1 is the standard output; and stream 2 is the standard error. **sh** lets you redirect any of these streams individually into a file, or combine streams into each other.

<&*n* **sh** can redirect the standard input and output to duplicate other file descriptors. (See the Lexicon article **file descriptor** for details on what these are.) This operator duplicates the standard input from file descriptor *n*.

>&*n* Merge one output stream with another. For example,

```
2>&1
```

merges the output of file descriptor 2 (the standard error) with that file descriptor 1 (the standard output).

<&- Close the standard input.

>&- Close the standard output.

When you execute a command in the foreground, that command inherits the file descriptors and signal traps (described below) of the invoking shell, modified by any specified redirection. When you execute a command in the background, it receives its input from the null device **/dev/null** (unless you redirect its input and output), and ignores all interrupt and quit signals.

File-Name Patterns

If a token contains any of the characters '?', '*', or '[', **sh** interprets it as being a file-name *pattern*. **sh** "expands" a pattern into the names of zero or more files in the current directory. These characters are sometimes called "wildcards," because each can represent any of several values, depending upon how you use them:

? Match any single character except newline. For example, the command

```
ls name?
```

prints the name of any file that consists of the string **name** plus any one character. If B name is followed by no characters, or is followed by two or more characters, it will not be printed.

***** Match a string of zero or more characters, other than newline. For example, the command

```
ls name*
```

prints the name of any file that begins with the string **name**, followed by zero or more other characters. Likewise, the command

```
ls name?*
```

prints the name of any file that consists of the string **name** followed by at least one character. Unlike **name***, the token **name?*** insists that be followed by at least one character before it will be printed.

[!*xyz***]**
Exclude characters *xyz* from the string search. For example, the command

```
ls [!abc]*
```

prints all files in the current directory except those that begin with **a**, **b**, or **c**.

[*C-c***]**
Enclose alternatives to match a single character. A hyphen '-' indicates a range of characters. For example, the command

```
ls name[ABC]
```

prints the names of files **nameA**, **nameB**, and **nameC** (assuming, of course, that those files exist in the current directory). The command

```
ls name[A-K]
```

prints the names of files **nameA** through **nameK** (again, assuming that they exist in the current directory).

When **sh** reads a token that contains one of the above characters, it replaces the token in the command line with an alphabetized list of file names that match the pattern. If it finds no matches, it passes the token unchanged to the command. For example, when you enter the command

```
ls name[ABC]
```

sh replaces the token **name[ABC]** with **nameA**, **nameB**, and **nameC** (again, if they exist in the current directory), so the command now reads:

```
ls nameA nameB nameC
```

It then passes this second, transformed version of the command line to the command **ls**.

Note that the slash '/' and leading period '.' must be matched explicitly in a pattern. The slash, of course, separates the elements of a path name; whereas a period at the begin of a file name usually (but not always) indicates that that file has special significance.

Pattern Matching in Prefixes and Suffices

sh recognizes special constructs that let you match patterns in the prefices and suffices of a string:

{#*parameter***}**
This operator gives the number of characters in *parameter*. For example, the command

```
foo=BAZZ ; echo ${#foo}
```

prints '4' on your screen, which is the number of characters that comprise variable **foo**.

{*string1*%*string2*}

This returns the longest string for which the beginning of *string1* matches *string2*. For example, if variable **xyzzy** is initialized to string **usr/bin/cpio**, then the command

```
echo ${xyzzy%/*}
```

echoes the string **usr/bin**.

{*string1*%%*string2*}

This returns the shortest string for which the beginning of *string1* matches *string2*. For example, if variable **xyzzy** is initialized to **usr/bin/cpio**, then the command

```
echo ${xyzzy%/*}
```

echoes the string **usr**.

{*string1*#*string2*}

This returns the longest string for which the end of *string1* matches *string2*. For example, if variable **plugh** is initialized to the string **usr/bin/cpio ,** the command

```
echo ${plugh#*/}
```

echoes **bin/cpio**.

{*string1*##*string2*}

This returns the shortest string for which the end of *string1* matches *string2*. For example, given that **plugh=usr/bin/cpio**, the command

```
echo ${plugh##*/}
```

echoes **cpio**.

The following shows how to use these expressions to implement the command **basename**:

```
basename () {
      set $(echo ${1##*/}) $2
      echo ${1%$2}
}
```

Quoting Text

From time to time, you will want to "turn off" the special meaning of characters. For example, you may wish to pass a token that contains a literal asterisk to a command; to do so, you need a way to tell **sh** not to expand the token into a list of file names. Therefore, **sh** recognizes the **quotation operators** '\', '"', and '''. These "turn off" (or *quote*) the special meaning of operators.

The backslash '\' quotes the following character. For example, the command

```
ls name\*
```

lists a file named **name***, and no other.

The shell ignores a backslash immediately followed by a newline, called a *concealed newline*. This lets you give more arguments to a command than will fit on one line. For example, the command

```
cc -o output file1.c file2.c file3.c \
      file4.c file5.c file19.c
```

invokes the C compiler **cc** to compile a set of C source files, the names of which extend over more than one line of input. You will find this to be extremely helpful, especially when you write scripts and **makefile**s, to help you write neat, easily read commands.

A pair of apostrophes ' ' prevents interpretation of any enclosed special characters. For example, the command

```
find . -name '*.c' -print
```

finds and prints the name of any C-source file in the current directory and any subdirectory. The command **find** interprets the '*' internally; therefore, you want to suppress the shell's expansion of that operator, which is accomplished by enclosing that token between apostrophes.

A pair of quotation marks " " has the same effect. Unlike apostrophes, however, **sh** performs parameter substitution and command-output substitution (described below) between quotation marks.

Environmental Variables

Environmental variables are names that can be assigned string values on a command line, in the form

name=value

name must begin with a letter, and can contain letters, digits, and the underscore character '_'. In shell input, '$*name*' or '${*name*}' represents the value of the variable. Consider, for example, the commands:

```
TEXT=mytext
nroff -ms $TEXT >$TEXT.out
```

Here, **sh** expands **$TEXT** before it executes the command **fBnroff**. This technique is very useful in large, complex scripts: by using variables, you can change the behavior of the script by editing one line, rather than having to edit numerous variables throughout the script.

Note that if an assignment precedes a command on the same command line, the effect of the assignment is local to that command; otherwise, the effect is permanent. For example,

```
kp=one testproc
```

assigns variable **kp** the value **one** only for the execution of the script **testproc**.

sh sets the following environmental variables by default:

The number of actual positional parameters given to the current command.

@ The list of positional parameters "$1 $2 ...".

***** The list of positional parameters "$1" "$2" ... (the same as '$@' unless quoted).

- Options set in the invocation of the shell or by the **set** command.

? The exit status returned by the last command.

! The process number of the last command invoked with '&'.

$ The process number of the current shell.

sh also references the following variables:

CWD Current working directory: this is the name of the directory in which you are now working. Note that this variable is not common to other implementations of **sh**. Code that uses it may not be portable to other operating systems.

HOME Initial working directory; usually specified in the password file **/etc/passwd**.

IFS Delimiters for tokens; usually space, tab and newline.

LASTERROR
Name of last command returning nonzero exit status.

MAIL Checked at the end of each command. If file specified in this variable is new since last command, the shell prints "You have mail." on the user's terminal.

PATH Colon-separated list of directories searched for commands.

PS1 First prompt string. By default, this is '$'.

PS2 Second prompt string. By default, this is '>'. **sh** prints it when it expects more input, such as when an open quotation-mark has been typed but a close quotation-mark has not been typed, or within a shell construct.

Beginning with release 4.2, the COHERENT implementation of **sh** performs word-expansion on the values of the variables **PS1** and **PS2**. For example, setting the variables

```
export SITE=$(uname -s)
PS1="$SITE $USER $(pwd) > "
```

create a prompt that consists of your site name, your login identifier, and your current working directory.

The special forms '${*nameCtoken*}' perform conditional parameter substition: *C* is one of the characters '-', '=', '+', or '?'. **sh** replaces the form '${*name-token*}' by the value of *name* if it is set, and by *token* otherwise. It handles the '=' form in the same way, but also sets the value of *name* to *token* if it was not set previously. **sh** replaces the '+' form by *token* if the given *name* is set. It replaces the '?' form by the value of *name* if set, and otherwise prints *token* and exits from the shell.

To unset an environmental variable, use the command **unset**. The command **unset -f** undefines a shell function (described below).

Command Output Substitution

sh can use the output of a command as shell input (e.g., as command arguments) by enclosing the command between grave characters ` `. For example, to list the contents of the directories named in file **dirs**, use the command

```
ls -l `cat dirs`
```

Constructs

sh lets you control execution of commands by the constructs **break**, **case**, **continue**, **for**, **if**, **until**, and **while**. It recognizes each reserved word only if it occurs unquoted as the first token of a command. This implies that a separator must precede each reserved word in the following constructs; for example, newline or ';' must precede **do** in the **for** construct.

The following describes each shell construct:

break *[n]*
Exit from a **for**, **until**, or **while** loop. If *n* is given, exit from the preceding *n* levels of looping.

case *token* **in [** *pattern [* | *pattern] ...*) *sequence*;; **] ... esac**
Check *token* against each *pattern*, and execute *sequence* associated with the first matching *pattern*.

continue *[n]*
Branch to the end of the *n*th enclosing **for**, **until**, or **while** construct.

for *name [* **in** *token ...]* **do** *sequence* **done**
Execute *sequence* once for each *token*. On each iteration, *name* takes the value of the next *token*. If the **in** clause is omitted, **$@** is assumed. For example, to list all files ending with **.c:**

```
for i in *.c
do
      cat $i
done
```

if *seq1* **then** *seq2 [* **elif** *seq3* **then** *seq4] ... [* **else** *seq5]* **fi**
Execute *seq1*. If the exit status is zero, execute *seq2*; if not, execute the optional *seq3* if given. If the exit status of *seq3* is zero, then execute *seq4*, and so on. If the exit status of all tested sequences is nonzero, execute *seq5*.

until *sequence1 [* **do** *sequence2]* **done**
Execute *sequence2* until the execution of *sequence1* results in an exit status of zero.

while *sequence1 [* **do** *sequence2]* **done**
Execute *sequence2* as long as the execution of *sequence1* results in an exit status of zero.

(*sequence***)**
Execute *sequence* within a subshell. This allows *sequence* to change the current directory, for example, and not affect the enclosing environment.

$(( )) Perform arithmetic expansion, as described in the POSIX Standard. The expression syntax is that of C, but the only values are signed integers, and there are no side effects (i.e., no increment, decrement, or assignment operators). The expression given inside this form is first processed for further expansions, then evaluated according to the C rules for arithmetic; the result is placed on the output command line. This is most useful when used with **return** and **exit** to form return values from functions and scripts.

To use **$(())** with the shell's logical operators and statements, you must use some indirection. For example:

```
val () {
      return $((! ($*)))
}
```

Or:

```
val $(($# < 5)) && {
      echo Not enough arguments
      exit 1
}
```

Or:

```
val $((${#foo} > 8)) {
      echo $foo is too long; use 8 characters, maximum.
      exit 2
}
```

{*sequence*
} Braces simply enclose a *sequence*. Note that the closing '}' must appear on the line that follows *sequence*.

Special Commands

sh usually executes commands with the **fork** system call, which creates another process. However, **sh** executes the commands in this section either directly or with an **exec** system call. See the Lexicon articles on **fork()** and **exec** for details on these calls.

. *script* Read and execute commands from *script*. Positional parameters are not allowed. **sh** searches the directories named in the environmental variable **PATH** to find the given *script*.

: *[token ...]*
A colon ':' indicates a "partial comment". **sh** normally ignores all commands on a line that begins with a colon, except for redirection and such symbols as **$**, **{**, **?**, etc.

A complete comment: if **#** is the first character on a line, **sh** ignores all text that follows on that line.

cd *dir* Change the working directory to *dir.* If no argument is given, change to the home directory.

command *command* **[***arguments***]**
When you issue a command, **sh** by default looks for that command among its set of built-in functions; if it cannot find it there, it looks for the command in the directories set in the environmental variable **PATH**. Thus, if you have given a shell function the same name as an executable command, **sh** will never find the executable.

The command **command** tells **sh** to look for *command* in the directories named in your **PATH**, and ignore any shell function with that name.

dirs **sh** lets you maintain a "directory stack", or stack of names of directories. You can push, pop, and otherwise manipulate the contents of this stack, which you can use for any purpose for which you need to access a number of directory names quickly. The command **dirs** prints the contents of the directory stack. The commands **pushd** and **popd** also manipulate the directory stack.

Please note that these commands are unique to the COHERENT implementation of **sh**, and are not portable to other shells. *Caveat utilitor.*

eval *[token ...]*
Evaluate each *token* and treat the result as shell input.

exec *[command]*
Execute *command* directly rather than performing **fork**. This terminates the current shell.

exit *[status]*
Set the exit status to *status*, if given; otherwise, the previous status is unchanged. If the shell is not interactive, terminate it.

export *[name ...]*
sh executes each command in an *environment*, which is a set of shell-variable names and their corresponding values. When you invoke **sh**, it inherits all environmental variables that are currently set; and it, in turn, normally passes those variables to each command it invokes. **export** specifies that the shell should pass the modified value of each given *name* to the environment of subsequent commands. When no *name* is given, **sh** prints the name and value of each variable marked for export.

getopts *optstring name [arg ...]*
Parse the *arg*s to *command.* See the Lexicon entry for **getopts** for details.

popd *[N ...]*
Pop the directory stack. When used without an argument, it pops the stack once. When used with one or more numeric arguments, **popd** pops the specified items from the stack; item 0 is the top of the stack. (For information on the directory stack, see the entry for the command **dirs**, above.)

pushd *[dir0 ... dirN]*
Push *dir0* through *dirN* onto the directory stack, and change the current directory to the last directory pushed onto the stack. When called without an argument, **pushd** exchanges the two top stack elements. (For information on the directory stack, see the entry for the command **dirs**, above.)

read *name ...*
Read a line from the standard input and assign each token of the input to the corresponding shell variable *name*. If the input contains fewer tokens than the *name* list, assign the null string to extra variables. If the input contains more tokens, assign the last *name* the remainder of the input.

readonly *[name ...]*
Mark each shell variable *name* as a read only variable. Subsequent assignments to read only variables will not be permitted. With no arguments, print the name and value of each read only variable.

set [-ceiknstuvx *[name ...]]*
Set listed flag. If *name* list is provided, set shell variables *name* to values of positional parameters beginning with **$1**.

shift Reset positional parameter **1** to the value **$2**, reset positional parameter **2** to the value **$3**, and so on. The original value of positional parameter **1** is thrown away.

times Print the total user and system times for all executed processes.

trap *[command] [n ...]*
Execute *command* if **sh** receives signal *n.* If *command* is omitted, reset traps to original values. To ignore a signal, pass null string as *command.* With *n* zero, execute *command* when the shell exits. With no arguments, print a description of the traps that have already been set.

umask *[nnn]*
Set user file creation mask to *nnn.* If no argument is given, print the current file creation mask.

wait *[pid]*
Delay execution of further commands until the process that has process identifier *pid* terminates. If *pid* is omitted, delay until every child process has finished executing. If no child process is active, this command finishes immediately.

Shell Functions

Beginning with COHERENT release 4.2, **sh** lets you declare and use functions. In effect, a function is a script that you load permanently into memory.

A function takes the form

```
function() {
      command $1 $2
      other_function $3 $4
}
```

A function begins with its name. A pair of parentheses after the name tell **sh** that this is a function.

The body of the function is enclosed within braces. A function can call any command, plus any other function. Arguments are passed into the function using the syntax **$1**, **$2**, etc., just as with a shell script.

sh keeps an internal list of the functions that you have declared. When it reads an identifier, it first searches its list of functions; if the identifier is not a function, **sh** then assumes that the identifier names a command, and it attempts to find that command in the directories you have named in your environmental variable **PATH**. Thus, if you give a function the same name as that of an existing command, **sh** will always use the function and never find the command. To suppress this behavior, use the command **command**, described above.

The following example function copies one file into another:

```
copyfile(){
      if [ $# -eq 1 ]; then
            cat $1
      else
            cp $1 $2
      fi
}
```

Shell Library

The file **/usr/lib/shell_lib.sh** holds a library of shell functions. You can import these library with the '.' command.

This library holds the following functions:

basename "*pathname*" **[** "*suffix*" "*prefix*" **]**
: This function behaves the same as the command **basename**, except that you can include an option *prefix* to strip as well as a *suffix*.

file_exists "*filename*"
: Return **true** if file *filename* exists.

find_file "*filename*" "*path*" **"path" ...**
: Seek *file* in every directory named in a *path*.

has_prefix "*string*" "*prefix*"
: Return **true** if *string* is prefixed with the string *prefix*.

is_empty "*arg*"
: Return true if *arg* is null.

is_equal "*arg1*" "*arg2*"
: Return true if *arg1* and *arg2* are equal.

is_numeric "*argument*"
: Return **true** if *argument* is numeric, or **false** if it is not.

is_yes "*arg*"
: Return true if *arg* matches 'Y', 'y', "YES", or "yes"; one if the argument matches 'N', 'n', "NO", or "no"; two if it matches anything else.

parent_of "*file*" **[** "*prefix*" "*suffix*" **]**
: By default, write the path name of *file*. *prefix* and *suffix* are the prefix and suffix of a command run with the output path name.

read_input "*prompt*" "*variable*" "*default*" "*validate*"
: Echo *prompt* onto the screen. Write what the user types into *variable*. If the user does not respond, set *variable* to *default*. The optional argument *validate* names a function that **read_input** calls to evaluate what the user types; often, this is the shell-library function **require_yes_or_no**.

require_yes_or_no "*arg*"
: This is the validation function for **read_input**. Check whether *arg* is properly affirmative or negative.

source_path "*script*" **[** "*command* " **]**
: Echo the name of the directory that contains *script*. Normally, this function is called with the **$0** of *script*. It pipes its output into *command* if this argument is set; if it is not, it writes to the standard output.

split_path "*string*" "*prefix*" "*suffix*"
: This function dissects *string*, which must consist of colon-separated elements, such as a **PATH** string. *prefix* and *suffix* give, respectively, the prefix and suffix of the command that is run for every component of *string*.

val "*expression*"
: Return the negated value of *expression*. You can use this function to turn shell arithmetic expressions into test results.

Scripts

Shell commands can be stored in a file, or *script*. The command

```
sh script [ parameter ... ]
```

executes the commands in *script* with a new subshell **sh**. Each *parameter* is a value for a positional parameter, as described below. If you have used the command **chmod** to make *script* executable, you may omit the **sh** command.

To ensure that a script is executed by **sh**, begin the script with the line:

```
#!/bin/sh
```

Parameters of the form '$*n*' represent command-line arguments within a script. *n* can range from zero through nine; **$0** always gives the name of the script. These parameters are also called *positional parameters*.

If no corresponding parameter is given on the command line, the shell substitutes the null string for that parameter. For example, if the script **format** contains the following line:

```
nroff -ms $1 >$1.out
```

then invoking **format** with the command

```
format mytext
```

invokes the command **nroff** to format the contents of **mytext**, and writes the output into file **mytext.out**. If, however, you invoke this command with the command line

```
format mytext yourtext
```

the script formats **mytext** but ignores **yourtext** altogether.

Reference **$*** represents all command-line arguments. If, for example, we change the contents of script **format** to read

```
nroff -ms $* >$1.out
```

then the command

```
format mytext yourtext
```

invoke **nroff** to format the contents of **mytext** and **yourtext**, and write the output into file **mytext.out**.

Commands in a script can also be executed with the . (dot) command. It resembles the **sh** command, but the current shell executes the script commands without creating a new subshell or a new environment; therefore, you cannot use command-line arguments.

Command-line Options

-c *string*
Read shell commands from *string.*

-e Exit on any error (command not found or command returning nonzero status) if the shell is not interactive.

-i The shell is interactive, even if the terminal is not attached to it; print prompt strings. For a shell reading a script, ignore the signals **SIGTERM** and **SIGINT**.

-k Place all keyword arguments into the environment. Normally, **sh** places only assignments to variables preceding the command into the environment.

-n Read commands but do not execute them.

-s Read commands from the standard input and write shell output to the standard error.

-t Read and execute one command rather than the entire file.

-u If the actual value of a shell variable is blank, report an error rather than substituting the null string.

-v Print each line as it is read.

-x Print each command and its arguments as it is executed.

- Cancel the **-x** and **-v** options.

If the first character of argument 0 is '-', **sh** reads and executes the scripts **/etc/profile** and **$HOME/.profile** before reading the standard input. **/etc/profile** is a convenient place for initializing system-wide variables, such as **TIMEZONE**.

Files

/etc/profile — System-wide initial commands
$HOME/.profile — User-specific initial commands
/dev/null — For background input
/tmp/sh* — Temporary files
/usr/lib/shell_lib.sh — Library of shell functions

See Also

commands, dup(), environ, exec, fork(), getopts, ksh, login, newgrp, set, signal(), test, Using COHERENT, vsh
Introduction to sh, the Bourne Shell, tutorial

For a list of all commands associated with **sh**, see the section **Shell Commands** in the **commands** Lexicon article.

Diagnostics

sh notes on the standard error all syntax errors in commands, and all commands that it cannot find. Syntax errors cause a noninteractive shell to exit. It gives error messages if I/O redirection is incorrect. **sh** returns the exit status of the last command executed or the status specified by an **exit** command.

shadow — System Administration

File that holds restricted passwords
/etc/shadow

COHERENT stores information in file **/etc/passwd**. This file identifies each user, gives her home directory, default shell, and base group. It must be universally readable so that it can be used by programs like **ls**, which must translate user-identification numbers into login identifiers.

In general, this system works well; however, it does create a hole in system security. If users' encrypted passwords are kept in **/etc/passwd**, which is universally readable, a "cracker" can read the passwords, decypher some of them with brute-force methods, and then log in as the users whose passwords she cracked.

To plug that hole in system security, UNIX implemented the method of "shadow" passwords. In this scheme, a user's login information is still kept in **/etc/passwd**; however, the encrypted passwords (plus supplemental information) is kept in the file **/etc/shadow**, which can be read only by a process with root-level permissions.

The shadow password file contains one entry per user. Each user identified in **/etc/shadow** must have an entry in **/etc/passwd**. The opposite is not true, but a user described in **/etc/passwd** who does not have an entry in **/etc/shadow** cannot log into your system. Each entry in **/etc/shadow** is laid out exactly the same as file **/etc/passwd.** At present, the COHERENT implementation of **login** uses only the *name* and *password* fields. For details, see the Lexicon entry for **passwd**.

Reading /etc/shadow

COHERENT includes four functions with which a program can read the shadow-password file **/etc/shadow**:

endspent()
Close **/etc/shadow** after reading from it.

getspent()
Read the next record from **/etc/shadow**. If a process has not yet read **/etc/shadow**, it returns the first record.

getspnam()
Return the first record for the user with a given login identifier.

setspent()
Return the seek pointer for **/etc/shadow** to the beginning of the file.

Functions **getspent()** and **getspnam()** return a pointer to an object with structure **spwd**, which gives an analogue for each field in **/etc/shadow**. This structure is defined in header file **<shadow.h>**. For details on this structure, see the Lexicon entry for **shadow.h**.

See Also

Administering COHERENT, login, shadow.h

Notes

For details of how the program **login** uses shadow passwords, see its entry in the Lexicon.

shadow.h — Header File

Definitions used with shadow passwords
#include <shadow.h>

The header file **<shadow.h>** declares and defines the functions, macros, structures, and constants used to manipulate shadow passwords.

<shadow.h> holds defines the structure **spwd**, which describes the records that are stored in file **/etc/shadow**. **<shadow.h>** gives two definitions **spwd**: one implements the structure used by UNIX System V, release 4; and the other implements the structure used by UNIX System V, release 3.

The following gives the form of **spwd** that is used by some releases of UNIX System V, release 4:

```
struct spwd {
      char *sp_namp;    /* User Name */
      char *sp_pwdp;    /* Encrypted password */
      long sp_lstchg;   /* Last changed date */
      long sp_min;
      long sp_max;
      long sp_warn;
      long sp_inact;
      long sp_expire;   /* Acct expiration date. */
      unsigned long sp_flag;
};
```

The following gives the version of **spwd** used by UNIX System V, release 3:

```
struct      spwd {
      char *sp_name; /* User name */
      char *sp_passwd; /* Encrypted password - non-POSIX */
      int sp_uid; /* User id */
      int sp_gid; /* Group id */
      int sp_quota; /* File space quota  - non-POSIX*/
      char *sp_comment; /* Comments - non-POSIX */
      char *sp_gecos; /* Gecos box number - non-POSIX */
      char *sp_dir; /* Working directory */
      char *sp_shell; /* Shell */
};
```

By default, COHERENT uses the System V, release 3, version of **spwd**.

For information on how to select a given version of **spwd**, see the discussion of compilation environments in the Lexicon article **header files**.

See Also

header files, endspent(), getspent(), getspnam(), libc, setspent(), shadow

SHELL — Environmental Variable

Name the default shell
SHELL=*shell*

The environmental variable **SHELL** names the shell that COHERENT invokes when you log in. The default is **SHELL=/bin/sh**, which invokes the Bourne shell.

See Also

environmental variables, sh

shellsort() — General Function (libc)

Sort arrays in memory
void shellsort(*data***,** *n***,** *size***,** *comp***)**
char ****data*; int** *n***,** *size***; int (****comp***)();**

shellsort() is a generalized algorithm for sorting arrays of data in memory, using D. L. Shell's sorting method. **shellsort()** works with a sequential array of memory called *data*, which is divided into *n* parts of *size* bytes each. In practice, *data* is usually an array of pointers or structures, and *size* is the **sizeof** the pointer or structure.

Each routine compares pairs of items and exchanges them as required. The user-supplied routine to which *comp* points performs the comparison. It is called repeatedly, as follows:

```
(*comp)(p1, p2)
char *p1, *p2;
```

Here, *p1* and *p2* each point to a block of *size* bytes in the *data* array. In practice, they are usually pointers to pointers or pointers to structures. The comparison routine must return a negative, zero, or positive result, depending on whether *p1* is less than, equal to, or greater than *p2*, respectively.

Example

For an example of how to use this routine, see the entry for **string**.

See Also

libc, qsort()
The Art of Computer Programming, vol. 3, pp. 84*ff*, 114*ff*

Notes

For a discussion of how the **shellsort** algorithm differs from that used by **qsort()**, see the Lexicon entry for **qsort()**.

shift — Command

Shift positional parameters
shift

Commands to the shell can be stored in a file, or *script*. Positional parameters pass command-line variables to a script.

shift changes the values of positional parameters. The old parameter values **$2**, **$3**, ... become the new parameter values **$1**, **$2** .... **shift** also reduces the value of **$#**, which gives the number of positional parameters, by one.

The shell executes **shift** directly.

See Also

commands, ksh, sh

shm — Kernel Module

Kernel module for shared memory

The kernel module **shm** enables System V-style shared memory. It is called a *kernel module* because you can link it into your kernel or exclude it, as you wish, just like a device driver; yet it is not a true device driver because it does not perform I/O with a peripheral device.

See Also

device drivers, kernel, shmget()

shm.h — Header File

Definitions used with shared memory
#include <sys/shm.h>

shm.h defines constants and macros used by routines that implement the COHERENT shared-memory facility.

See Also

header files, shmget()

shmat() — General Function (libc)

Attach a shared-memory segment to a process
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
char *shmat (*shmid***,** *shmaddr***,** *shmflg***)**
int *shmid***,** *shmflag***; char ****shmaddr***;**

shmat() attaches the shared-memory segment associated with the identifier *shmid* with the **.data** segment of the calling process.

shmat() selects the address at which to attach the shared-memory segment. If

shmflg `& SHM_RDONLY`

is true, the attached memory is read-only; otherwise, it is read-write.

shmat() fails if any of the following is true:

- *shmid* is not a valid shared-memory identifier. **shmat()** sets **errno** to **EINVAL**.
- The calling process lacks appropriate permission (**EACCES**).
- Not enough memory is available to hold the shared-memory segment (**ENOMEM**).
- The process already has the maximum number of shared-memory segments attached to it (**EMFILE**).

You can attach more than one shared-memory segment to a process, up to a maximum of six. COHERENT assigns each segment its own address.

If all went well, **shmat()** returns the address of the newly attached shared-memory segment; otherwise, it returns -1 and sets **errno** to an appropriate value.

Example

For an example of this function, see the Lexicon entry for **shmget()**.

See Also

libc, shmctl(), shmdt(), shmget()

Notes

The COHERENT implementation of shared memory does not yet support attaching a shared-memory segment to a user-defined address. Therefore, you should always set *shmaddr* to zero.

shmctl() — General Function (libc)

Manipulate shared memory
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
shmctl(*shmid***,** *command***,** *buf***)**
int *shmid***,** *command***; struct shmid_ds ***buf***;**

shmctl() controls the COHERENT system's shared-memory facility. Note that shared memory consists of the segment of memory being shared, plus a copy of structure **shmid_ds**, which is defined in header file **<sys/shm.h>**. This structure describes the shared-memory segment and identifies who can manipulate it, and how.

command names the operation that you want **shmctl()** to perform, as follows:

IPC_RMID Remove the system identifier *shmid* and destroy its associated shared memory segment and **shmid_ds** structure. Only the superuser **root** or the user whose effective user ID matches the value of field **uid** can invoke this command.

IPC_SET Copy fields **shm_perm.uid**, **shm_perm.gid**, and **shm_perm.mode** (low nine bits only) from the **ipc_perm** associated with *buf* into *shmid*. Only the superuser **root** or the user who created this shared-memory segment can invoke this command.

IPC_STAT Copy every element of the **shmid_ds** associated with *shmid* into the one pointed to by *buf*.

SHM_LOCK Lock the shared-memory segment *shmid*, to keep it from being paged out of memory. Only the superuser **root** can invoke this command. Because COHERENT does not support paging, this command present does nothing.

SHM_UNLOCK Unlock the shared-memory segment *shmid*, to permit it to be paged out of memory. Only the superuser **root** can invoke this command. Because COHERENT does not support paging, this command present does nothing.

shmctl() fails if any of the following is true:

- *shmid* is not a valid shared-memory identifier. **shmget()** sets **errno** to **EINVAL**.
- *command* is not a valid command (**EINVAL**).
- *command* equals **IPC_STAT** but the owner of the calling process lacks permission (**EACCES**).
- *command* equals **IPC_RMID** or **IPC_SET** but the owner of the calling process lacks permission (**EPERM**).
- *buf* points to an illegal address (**EFAULT**).

shmctl() returns zero if all went well; otherwise, it returns -1 and sets **errno** to an appropriate value.

Example

For an example of this function, see the Lexicon entry for **shmget()**.

Files

/usr/include/sys/ipc.h
/usr/include/sys/shm.h

See Also

libc, shmat(), shmdt(), shmget()

Notes

For information on other methods of interprocess communication, see the Lexicon entries for **semctl()** and **msgctl()**.

shmdt() — General Function (libc)

Detach a shared-memory segment from a process

#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
char *shmdt (*shmaddr***)**
char ****shmaddr*;**

shmdt() detaches the shared-memory segment at address *shmaddr* from the calling process. If all went well, **shmdt()** returns returns zero; otherwise, it returns -1 and sets **errno** to an appropriate value. In particular, it sets **errno** to **EINVAL** if *shmaddr* does not point to the beginning of a shared-memory segment.

Example

For an example of this function, see the Lexicon entry for **shmget()**.

See Also

libc, shmctl(), shmdt(), shmget()

Notes

The COHERENT implementation of shared memory does not yet support attaching a shared-memory segment to a user-defined address. Therefore, you should always set *shmaddr* to zero.

shmget() — General Function (libc)

Create or get shared-memory segment

#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
int shmget(*memkey***,** *size***,** *flag***)**
key_t *memkey***; int** *size***,** *flag***;**

shmget() creates a shared-memory identifier, associated data structure, and shared-memory segment, links them to the identifier *memkey*, and returns the shared-memory identifier that it has associated with *memkey*.

memkey is an identifier that your application generates to identify its shared-memory segments. To guarantee that each key is unique, you should use the function call **ftok()** to generate keys.

size gives the size, in bytes, of the shared-memory segment that you want **shmget()** to create.

flag can be bitwise OR'd to include the following constants:

IPC_ALLOC This process already has a shared-memory segment; please fetch it.

IPC_CREAT If this process does not already have a shared-memory segment, please create one.

IPC_EXCL Fail if a shared-memory segment already exists for this process.

IPC_NOWAIT Fail if the process must wait to obtain a shared-memory segment.

When it creates a shared-memory segment, **shmget()** also creates a copy of structure **shmid_ds**, which is defined in header file **<sys/shm.h>**, and which describes the shared-memory segment. It is defined as follows:

```
struct shmid_ds {
        struct ipc_perm                     shm_perm;/* operation permission struct */
        int shm_segsz;                      /* segment size */
        char *__unused;                     /* for binary compatibility */
        char __pad [4];                     /* for binary compatibility */
        pid_t shm_lpid;                     /* pid of last shmop */
        pid_t shm_cpid;                     /* pid of creator */
        unsigned short shm_nattch;          /* current # attached */
        unsigned short shm_cnattach;        /* for binary compatibility */
        time_t shm_atime;                   /* last shmat time */
        time_t shm_dtime;                   /* last shmdt time */
        time_t shm_ctime;                   /* last change time */
};
```

Field **shm_perm** is a structure of type **ipc_perm**, which header file **<sys/ipc.h>** defines as follows:

```
struct ipc_perm {
        unsigned short uid;                 /* owner's user id */
        unsigned short gid;                 /* owner's group id */
        unsigned short cuid;                /* creator's user id */
        unsigned short cgid;                /* creator's group id */
        unsigned short mode;                /* access modes */
        unsigned short seq;                 /* slot usage sequence number */
        key_t key;                          /* key */
};
```

shmget() initializes **shm_id** as follows:

- It sets fields **shm_perm.guid**, **shm_perm.uid**, **shm_perm.cgid**, and **shm_perm.gid** to, respectively, the effective user ID and effective group ID of the calling process.
- It sets the low-order nine bits of field **shm_perm.mode** to the low-order nine bits of *flag*. These nine bits define access permissions: the top three bits give the owner's access permissions (read, write, execute), the middle three bits the owning group's access permissions, and the low three bits access permissions for others.
- It sets field **shm_segsz** equal to *size*.
- It sets fields **shm_atime**, **shm_dtime**, **shm_lpid**, and **shm_nattch** to zero, and field **shm_ctime** to the current time.

shmget() fails if any of the following is true:

- *size* is smaller than one byte, or larger than 0x10000 (the system-imposed maximum). **shmget()** sets **errno** to **EINVAL**.
- A shared-memory identifier exists for *memkey* but permission, as specified by *flag*'s low-order nine bits, is not granted (**EACCES**).
- A shared-memory identifier exists for *memkey*, but the size of its associated segment is less than *size*, and *size* does not equal zero (**EINVAL**).
- A shared-memory identifier does not exist for *memkey* and (*flag* & **IPC_CREAT**) is false (**ENOENT**).
- **shmget()** tried to create a shared-memory segment, but could not because 100 (the COHERENT-defined maximum) already exist (**ENOSPC**).

- **shmget()** tried to create a shared-memory identifier, but could not because not enough physical memory is available (**ENOMEM**).
- A shared-memory identifier already exists for *memkey*, but *flag* requests that **shmget()** create an exclusive segment it — i.e.

  ((*flag* & IPC_CREAT) && (*flag* & IPC_EXCL))

  is true (**EEXIST**).

If all goes well, **shmget()** returns a shared-memory identifier, which is always a non-negative integer. Otherwise, it returns -1 and sets **errno** to an appropriate value.

Example

The following demonstrates how to use COHERENT's shared-memory feature. Please note that this example will *not* work with versions of COHERENT prior to release 4.2.

The example consists of two programs: **writeshm**, which captures input from the keyboard and writes it into a shared-memory segment; and **readshm**, which reads and displays from the shared-memory segment the text that **writeshm** put there. Each program terminates when you type "end".

Note that this example is most effective if you run each program from its own virtual console.

The first program gives the source for **writeshm**:

```
#include <stdio.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <string.h>

main()
{
	int iShmId; /* Segment id */
	char *cpShm; /* Pointer to the segment */
	key_t key; /* Segment key */

	key = ftok("/etc/passwd", 'S');     /* Get a key */

	/* if a shared-memory segment exists, get it; otherwise, create one */
	if ((iShmId = shmget(key, 256, 0644 | IPC_CREAT)) < 0) {
		perror("get");
		exit(1);
	}

	/* Attach segment to process. Use an attach address of zero to
	 * let the system find a correct virtual address to attach.
	 */
	if ((cpShm = shmat(iShmId, 0, 0644)) == (char *) -1) {
		perror("shmat");
		exit(1);
	}
	printf("Server is ready.\n");
	printf("Any message to continue, 'end' to exit\n");

	for (;;) {
		printf("Enter the message -> ");
		gets(cpShm);
		if (!strcmp(cpShm, "end")) {
			puts("Bye");
			shmdt(cpShm);     /* Detach segment */
			break;
		}
	}
}
```

The next program gives the source for **readshm**:

```
#include <stdio.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <string.h>

main()
{
	int iShmId; /* Segment id */
	char *cpShm; /* Pointer to the segment */
	key_t key; /* Segment key */
	char cBuf[16]; /* Read buffer */

	/* Get a key */
	key = ftok("/etc/passwd", 'S');

	/* Get shared memory id. If it does not exist, do *not* create it. */
	if ((iShmId = shmget(key, 256, 0644)) < 0) {
		perror("get");
		exit(1);
	}

	/* attach shared-memory segment to the process */
	if ((cpShm = shmat(iShmId, 0, 0644)) == (char *) -1) {
		perror("shmat");
		exit(1);
	}
	printf("Client is ready\n");

	for (;;) {
		printf("Press enter to read the message -> ");
		gets(cBuf);
		printf("Got: \"%s\"\n", cpShm);

		/* Exit on the 'end': detach and remove segment */
		if (!strcmp(cpShm, "end")) {
			struct shmid_ds   stShmId;

			puts("Bye");
			shmdt(cpShm);
			if (shmctl(iShmId, IPC_RMID, &stShmId)) {
				perror("shmctl");
				exit(1);
			}
			break;
		}
	}
}
```

Files

/usr/include/sys/ipc.h
/usr/include/sys/shm.h

See Also

ftok(), ipcrm, ipcs, libc, libsocket, shmat(), shmctl(), shmdt()

Notes

Prior to release 4.2, COHERENT implemented shared memory through the driver **shm**. In release 4.2, and subsequent releases, COHERENT implements shared memory as a set of functions that conform in large part to the UNIX System-V standard.

The kernel variables **SHMMAX** and **SHMMNI** set, respectively, the maximum size of a shared-memory segment and the number of shared-memory segments that can exist at any given time. Daredevil system operators who have large amounts of memory at their disposal may wish to change these variables to increase the system-defined limits. For details on how to do so, see the Lexicon entry **mtune**.

short — C Keyword

Data type

short is a numeric data type. The ANSI standard states that it cannot be longer than an **int**.

COHERENT defines a **short** to be two bytes long; thus, **sizeof short** equals two **char**s, or 15 bits plus a sign, and can hold any value from -32,768 to 32,767.

A **short** normally is sign extended when cast to a larger data type; however, an **unsigned short** will be zero extended when cast.

See Also

C keywords, data format, data type, int, long
ANSI Standard, §6.1.2.5

shutdown — Command

Shut down the COHERENT system
/etc/shutdown [reboot | halt | single | powerfail] *time*

shutdown shuts down the COHERENT system. You must use this command to shut COHERENT down. *Failure to shut down the system before rebooting or shutting off the computer could damage the* COHERENT *file system and destroy data.*

When you invoke **shutdown**, you must specify the "level" shutdown, and the time to shutdown. The level must be one of the following:

reboot Bring down the system, then reboot it automatically. Use this level if, for example, you are installing a new kernel.

halt Halt the system, but do not reboot it or enter single-user mode. Use this option when you intend to turn off your computer.

single Bring down the system to single-user mode.

powerfail
Bring down the system as quickly as possible. Normally, this level is invoked by a daemon that has received information of a power failure from your system's uninterruptable power supply (UPS).

time is the interval, in minutes, from the time you invoke the command to the time that **shutdown** shuts the system down. Setting *time* to zero shuts down the system immediately. Every minute, **shutdown** displays on every user's terminal the message

```
System going down in N minutes!
```

where *N* is the number of minutes left until shutdown. When time has expired, **shutdown** displays the message

```
System is going down now!
```

at which point users have ten seconds to save their files and exit. Users who have turned off system messages will not, of course, see these messages.

After the system has been halted, you do not need to type **sync**; **shutdown** does that automatically.

If users have not logged off from the system when it comes time to shut the system down, you will see the prompt:

```
Some file systems remain mounted.  Proceed with shutdown ? [y]
```

If you type 'n', the shutdown will be aborted. You should then make sure that the users have logged off, then invoke **/etc/shutdown** again. To lock out new users from logging in while you are trying to shut the system down, create the file **/etc/nologin**. Note that this file is removed automatically when you reboot your system.

If you type 'y', **shutdown** will continue as before. Users will be thrown off the system; any files they might have had opened at that time will not be updated.

See Also

commands, nologin, reboot

Notes

Only the superuser **root** can run **shutdown**.

shutdown can be run from any terminal. When the system reboots, however, control returns to the system console.

shutdown was written by Udo Munk (udo@umunk.GUN.de).

shutdown() — Sockets Function (libsocket)

Replace function to shut down system
int shutdown (*s*, *how***)**
int *s*, *how***;**

Function **shutdown()** does nothing under COHERENT. It is present in its sockets library to ensure that imported code will link.

See Also

libsocket

sigaction() — System Call (libc)

Perform detailed signal management
#include <signal.h>
int sigaction (*signal*, *action*, *old_action***)**
int *signal***; const struct sigaction** **action***; struct sigaction** **old_action***;**

sigaction() lets the calling process specify the action it will take when it receives *signal*.

signal can be any of the signals named in the Lexicon entry for **signal()** except **SIGKILL** and **SIGSTOP**.

action points to a structure that specifies the action to take when *signal* is received. If *action* is set to NULL, the current disposition of the signal is unaffected.

old_action points to a structure that describes the action previously associated with *signal*, and that is to be restored upon return from **sigaction()**.

The structure **sigaction** has the following members:

```
void  (*sa_handler)();
sigset_t    sa_mask;
int   sa_flags;
```

sa_handler gives the disposition of the signal. You can set it to any of the actions described in the article for **signal()**.

sa_mask identifies the signals to be blocked while the signal handler is active. Upon entry to the signal handler, that set of signals is added to the set of signals already being blocked when *signal* was received. *signal* itself is also blocked. Note that you cannot block **SIGSTOP** and **SIGKILL**.

sa_flags specifies a set of flags used to modify the behavior of *signal*. As of this writing, **sigaction()** recognizes only the flag **SA_NOCLDSTOP**: If this is set and *signal* equals **SIGCHLD**, *signal* is not sent to the calling process when its child processes stop or continue.

sigaction() returns zero if all is well. It fails and returns -1 if either of the following is true:

- *signal* does not identify a valid signal. **sigaction()** sets **errno** to **EINVAL**.
- *action* or *old_action* points outside the process's allocated address space. **sigaction()** sets **errno** to **EFAULT**.

See Also

libc, sigaddset(), sigdelset(), sigemptyset(), sigfillset(), sigismember(), signal(), signal.h, sigpending(), sigprocmask(), sigset(), siglongjmp(), sigsetjmp(), sigsuspend()
POSIX Standard, §3.3.4

sigaddset() — Signal Function (libc)

Add a signal to a set of signals
#include <signal.h>
int sigaddset (*set***,** *signo***)**
sigset_t ****set*; int** *signo***;**

sigaddset() is one of a set of signalling functions that manipulate data objects addressable by the application, instead of a set of signals known to the system. It adds the signal *signo* to the set of signals to which *set* points.

If all goes well, **sigaddset()** returns zero. If *signo* is set to an invalid or unsupported value, it returns -1 and sets **errno** to **EINVAL**.

See Also

libc, sigaction(), sigdelset(), sigemptyset(), sigfillset(), sigismember()

Notes

If your program is compiled using the System V Release 4 compilation environment, this is a function that is linked in from **libc**. If not, a macro form is used.

sigdelset() — Signal Function (libc)

Delete a signal from a set
#include <signal.h>
int sigdelset (*set***,** *signo***)**
sigset_t ****set*;**
int *signo***;**

sigdelset() is one of a set of signalling functions that manipulate data objects addressable by the application, instead of a set of signals known to the system. It deletes the signal *signo* from the set of signals to which *set* points.

If all goes well, **sigdelset()** returns zero. If *signo* is set to an invalid or unsupported value, it returns -1 and sets **errno** to **EINVAL**.

See Also

libc, sigaction(), sigaddset(), sigemptyset(), sigfillset(), sigismember()

Notes

If your program is compiled using the System V Release 4 compilation environment, this is a function that is linked in from **libc**. If not, a macro form is used.

sigemptyset() — Signal Function (libc)

Initialize a set of signals
#include <signal.h>
int sigemptyset (*set***)**
sigset_t ****set*;**

sigemptyset() is one of a set of signalling functions that manipulate data objects addressable by the application, instead of a set of signals known to the system. It initializes the set of signals to which *set* points, such that all standard signals are excluded.

sigemptyset() returns zero if all goes well. If a problem occurs, it returns -1 and sets **errno** to an appropriate value.

An application must call either **sigemptyset()** or **sigfillset()** at least once for each object of type **sigset_t** prior to any other object. If such an object is not initialized in this way, but is supplied as an argument to any of the functions **sigaddset()**, **sigdelset()**, **sigismember()**, **sigaction()**, **sigprocmask()**, **sigpending()**, or **sigsuspend()**, the results are undefined.

See Also

libc, sigaction(), sigaddset(), sigdelset(), sigfillset(), sigismember()

Notes

If your program is compiled using the System V Release 4 compilation environment, this is a function that is linked in from **libc**. If not, a macro form is used.

sigfillset() — Signal Function (libc)

Initialize a set of signals
#include <signal.h>
int sigfillset (*set***)**
sigset_t ***set*;**

sigfillset() is one of a set of signalling functions that manipulate data objects addressable by the application, instead of a set of signals known to the system. It initializes the set of signals to which *set* points, such that all standard signals are included.

sigfillset() returns zero if all goes well. If a problem occurs, it returns -1 and sets **errno** to an appropriate value.

Applications must call either **sigemptyset()** or **sigfillset()** at least once for each object of type **sigset_t** prior to any other object. If such an object is not initialized in this way, but is supplied as an argument to any of the functions **sigaddset()**, **sigdelset()**, **sigismember()**, **sigaction()**, **sigprocmask()**, **sigpending()**, or **sigsuspend()**, the results are undefined.

See Also

libc, sigaction(), sigaddset(), sigdelset(), sigemptyset(), sigismember()

Notes

If your program is compiled using the System V Release 4 compilation environment, this is a function that is linked in from **libc**. If not, a macro form is used.

sighold() — System Call (libc)

Place a signal on hold
#include <signal.h>
int sighold (*sigtype***)**
int *sigtype***;**

sighold() is a member of the **sigset()** family of signal-handling system calls. It is equivalent to the system call

```
sigset(sigtype, SIG_HOLD);
```

This, in effect, places the signal *sigtype* “on hold” until the system call **sigrelse()** is invoked for it. Only one “copy” of *sigtype* can be held at a time.

Thus, you can use **sighold()** and **sigrelse()** to defer processing of the signal *sigtype*. This permits you to protect a portion of your application from this signal until it is ready to process it.

See the Lexicon entry for **signal()** for a list of recognized signals. Note that signal **SIGKILL** cannot be held.

sighold() returns zero if all went well. If something went wrong, it returns -1 and sets **errno** to an appropriate value.

See Also

libc, sigignore(), signal(), sigpause(), sigrelse(), sigset()

Notes

For more information on the **sigset()** family of signal-handling system calls, see the Lexicon entry for **sigset()**.

sigignore() — System Call (libc)

Tell the system to ignore a signal
#include <signal.h>
int sigignore (*sigtype***)**
int *sigtype***;**

sigignore() is a member of the **sigset()** family of signal-handling system calls. It is equivalent to the system call

```
sigset(sigtype, SIG_IGN);
```

This, in effect, tells the system to ignore all signals of type **sigtype**.

See the Lexicon entry for **signal()** for a list of recognized signals. Note that signal **SIGKILL** cannot be ignored.

sigignore() returns zero if all went well. If something went wrong, it returns -1 and sets **errno** to an appropriate value.

See Also

libc, sighold(), signal(), sigpause(), sigrelse(), sigset()

Notes

For more information on the **sigset()** family of signal-handling system calls, see the Lexicon entry for **sigset()**.

sigismember() — Signal Function (libc)

Check if a signal is a member of a set
#include <signal.h>
int sigismember (*set***,** *signo***)**
sigset_t **set***;**
int *signo***;**

sigismembeer() is one of a set of signalling functions that manipulate data objects addressable by the application, instead of a set of signals known to the system. It tests whether the signal *signo* is a member of the set of signals to which *set* points.

If *signo* is a member of *set*, **sigismember()** returns zero. If *signo* is set to an invalid or unsupported value, it returns -1 and sets **errno** to **EINVAL**.

See Also

libc, sigaction(), sigaddset(), sigdelset(), sigemptyset(), sigfillset()

Notes

If your program is compiled using the System V Release 4 compilation environment, this is a function that is linked in from **libc**. If not, a macro form is used.

siglongjmp() — General Function (libc)

Perform a non-local goto and restore signal mask
#include <setjmp.h>
void siglongjmp(*environ***,** *value***)**
sigjmp_buf *environ***; int** *val***;**

siglongjmp() behaves like the function **longjmp()**, except that it also restores the signal mask.

environ points to an array of type **sigjmp_buf**, which is declared in header file **setjmp.h**. It must have been initialized by a call to **sigsetjmp()**. *value* is the integer value to be returned to the function that called **sigsetjmp()**.

See Also

libc, sigaction(), sigprocmask(), sigsetjmp(), sigsuspend()
POSIX Standard, §8.3.1

signal() — System Call (libc)

Specify action to take upon receipt of a given signal
#include <signal.h>
int (*signal(*sigtype, function***))()**
int *sigtype***, (****function***)();**

A process can receive a *signal*, or interrupt, from a hardware exception, terminal input, or a **kill()** call made by another process. A hardware exception might be caused by an illegal instruction or a bad machine address. The terminal interrupt character (described in detail in the Lexicon entry **tty**) generates a process interrupt (and in one case a core dump file for debugging purposes).

signal() tells the signal handler what to do when the current process receives signal *sigtype*. *sigtype* is the signal to process, as defined below. *function* points to the routine to execute when *sigtype* is received. This can be a function of your own creation; or you can use one of the following macros, which expand into pointers to system-defined functions:

SIG_DFL
This is the default action. The process terminates just as if it called the function **exit()**. In addition, the system writes a core file in the current working directory if *sigtype* is any of the following: **SIGQUIT**, **SIGSYS**, **SIGTRAP**, or **SIGSEGV**. (Note that this behavior applies only to executables for which you have write permission. If you lack write permission on an executable, then no core file is written.) For more information on core files, see the Lexicon entry **core**.

SIG_IGN
Ignore *sigtype*. The system discards all signals of this type.

signal() returns a pointer to the previous action. If *sigtype* is not a recognized signal, **signal()** returns **(int (*)())-1**.

With the exception of **SIGKILL** and **SIGTRAP**, caught signals are reset to the default action **SIG_DFL**. To catch a signal again, the routine to which *function* points must reissue the call to **signal()**.

The following list names the signals that **signal()** can process, as defined in the header file **signal.h**. Note that the signal **SIGKILL**, which kills a process, can be neither caught nor ignored. Signals marked by an asterisk produce a core dump if the action is **SIG_DFL**:

SIGHUP . . . . Hangup
SIGINT . . . . Interrupt
SIGQUIT* . . . . Quit
SIGILL*. . . . . Illegal instruction
SIGTRAP* . . . . Trace trap
SIGIOT . . . . IOT instruction
SIGABRT* . . . . To be replaced by SIGIOT
SIGEMT . . . . Emulator trap
SIGFPE* . . . . Floating-point exception
SIGKILL . . . . Kill
SIGBUS. . . . . Bus error
SIGSEGV* . . . . Segmentation violation
SIGSYS* . . . . Bad argument to system call
SIGPIPE . . . . Write to pipe with no readers
SIGALRM . . . . Alarm
SIGTERM . . . . Software termination signal
SIGUSR1. . . . . User-defined signal
SIGUSR2. . . . . User-defined signal
SIGCLD. . . . . Death of a child
SIGCHLD. . . . . Death of a child
SIGPWR . . . . Restart
SIGWINCH. . . . . Window change
SIGPOLL. . . . . Polled event in stream

A signal may be caught during a system call that has not yet returned. In this case, the system call appears to fail, with **errno** set to **EINTR.** If desired, such an interrupted system call may be reissued. System calls which may be interrupted in this way include **pause()**, **read()** on a device such as a terminal, **write()** on a pipe, and **wait()**.

Example

The following program demonstrates **signal()**, **kill()**, **getpid()**, and **fork()**.

```
#include <signal.h>

int got_it; /* Each side gets its own copy of all data at the fork */
int errset;

/*
 * Control comes here on SIGTRAP.  Do no I/O in signal function.
 * Reset the signal if you ever want another.
 */

void
sig_ser()
{
      got_it = 1; /* tell the child we got it */
```

```
	if (0 > signal(SIGTRAP, sig_ser))   /* reset the signal */
		errset = 1;
}

main()
{
	int count;
	int child, parent;

	parent = getpid();      /* Both sides will get a copy */

	if (signal(SIGTRAP, sig_ser) < 0) { /* sets for both sides */
		perror("signal set failed");
		exit(0);
	}

	if (child = fork()) {   /* parent gets the child's id */
		for (count = 0; count < 3; count++) {
			kill(child, SIGTRAP);   /* signal the child */

			while(!got_it)          /* wait for signal */
				sleep(1);
			if (errset)
				perror("parent: signal reset failed");

			printf("parent got signal %d\n", count);
			got_it = errset = 0;
		}
		exit(0);
	}

	for (count = 0; count < 3; count++) {
		while(!got_it)                /* wait for signal */
			sleep(1);
		if (errset)
			perror("child: signal reset failed");
		printf("child got signal %d\n", count);   /* show we got it */

		kill(parent, SIGTRAP);        /* signal the parent */
		got_it = errset = 0;
	}
	exit(0);
}
```

See Also

kill, kill(), libc, ptrace(), sh, sigaction(), signame, sigset()
ANSI Standard, §7.7.1.1

Notes

The function **signal()** predates the **sigset()** and **sigaction()** sets of signal-handling functions. *Never* combine **signal()** with any of the **sigset()** or **sigaction()** families of functions: use one or the other, but not both. For a description of how **signal()** differs from **sigset()** and **sigaction()**, see their Lexicon entries.

signal.h — Header Files

Define signals
#include <signal.h>

The header file **signal.h** defines manifest constants that name all of the machine-independent signals that the COHERENT system uses to communicate with its processes.

See Also

header files, kill, signal()
ANSI Standard, §7.7
POSIX Standard, §3.3.1

signame — Global Variable

Array of names of signals
#include <signal.h>
extern char *signame[_SIGNAL_MAX];

When a program terminates abnormally, its parent process receives a byte of termination information from the system call **wait()**. This byte contains a signal number, as defined in the header file **signal.h**. For example, **SIGINT** indicates an interrupt from the terminal.

The array **signame**, indexed by signal number, contains strings that give the meaning of each signal. Thus, **signame[SIGINT]** points to the string "interrupt". For portability reasons, all programs which wait on child processes (such as the shell **sh**) should use **signame**.

Files

<signal.h>

See Also

Programming COHERENT, sh, signal(), wait

Notes

Please note that through revision 10 of the COHERENT manual, the signal numbers in **signame[]** were offset by one. That is erroneous: the signal numbers are not offset at all.

In standard implementations of UNIX, the manifest constant **NSIG** was one larger than the number of signals. Prior to release 4.2, however, COHERENT defined **NSIG** as being equal to the number of signals. Beginning with release 4.2, COHERENT defines **NSIG** to conform to the UNIX usage, and introduces the manifest constant **_SIGNAL_MAX**, which is equal to the number of signals. If your code depends upon the old definition of **NSIG**, you should replace it with **_SIGNAL_MAX**.

Please note that **signame[]** is obsolete, and will be removed from future releases of COHERENT. Please do not incorporate it into new code, and you should try to remove it from existing code.

sigpause() — System Call (libc)

Pause until a given signal is received
#include <signal.h>
int sigpause (*sigtype***)**
int *sigtype***;**

sigpause() is a member of the **sigset()** family of signal-handling system calls. It pauses until the signal *sigtype* is received. If, however, a signal of type *sigtype* had previously been "placed on hold" by a call to **sighold()**, **sigpause()** releases the signal for processing, just as if you had invoked the system call **sigrelse()**.

See the Lexicon entry for **signal()** for a list of recognized signals.

sigpause() returns zero if all went well. If something went wrong, it returns -1 and sets **errno** to an appropriate value.

See Also

libc, sighold(), sigignore(), signal(), sigrelse(), sigset()

Notes

For more information on the **sigset()** family of signal-handling system calls, see the Lexicon entry for **sigset()**.

Note that invoking

```
sigpause(SIGCHLD)
```

with no children pauses forever.

sigpending() — System Call (libc)

Examine signals that are blocked and pending
#include <signal.h>
int sigpending(*stash***)**
sigset_t ****stash*****;**

sigpending() retrieves the signals that have been sent to the calling process but have been blocked by the calling process's signal mask. *stash* points to the area of memory where the retrieved signals are to be stored.

sigpending() returns zero if all goes well. It returns -1 and sets **errno** to **EFAULT** if *stash* points outside the process's allocated address space.

See Also

libc, sigaction(), signal()
POSIX Standard, §3.3.6

sigprocmask() — System Call (libc)

Examine or change the signal mask
#include <signal.h>
int sigprocmask(*how***,** *set***,** *old_set***)**
int *how***; const sigset_t** **set***; sigset_t** **old_set***;**

sigprocmask() examines or changes the calling process's signal mask.

how defines how to modify the mask, as follows:

SIG_BLOCK
Add to the signal mask the set of signals to which *set* points.

SIG_UNBLOCK
Remove from the signal mask the set of signals to which *set* points.

SIG_SETMASK
Replace the current signal mask with the set of signals to which *set* points. If *old_set* is not NULL, **sigprocmask()** stores the old mask in the space to which it points.

If *set* is NULL, **sigprocmask()** ignores the value of *how*; thus, you can use the call to find which signals are in the signal mask.

If any unblocked unblocked signals are pending after you call **sigprocmask()**, at least one of those signals will be delivered before **sigprocmask()** returns.

If all goes well, **sigprocmask()** returns zero. **sigprocmask()** returns -1 if either of the following conditions is true:

- *how* is not set to a recognized value. **sigprocmask()** sets **errno** to **EINVAL**.
- *set* or *old_set* points outside the process's allocated address space. **sigprocmask()** sets **errno** to **EFAULT**.

In either error condition, **sigprocmask()** does not change the signal mask.

See Also

libc, signal(), siglongjmp(), sigsetjmp()
POSIX Standard, §3.3.5

sigrelse() — System Call (libc)

Release a signal for processing
#include <signal.h>
int sigrelse (*sigtype***)**
int *sigtype***;**

sigrelse() is a member of the **sigset()** family of signal-handling system calls. It releases the signal *sigtype*, which had previously been "placed on hold" by the system call **sighold()**. Only one "copy" of *sigtype* can be held at a time. Thus, you can use **sighold()** and **sigrelse()** to defer processing of the signal *sigtype*. This permits you to protect a portion of your application from this signal until it is ready to process it.

When *sigtype* is released, it is processed by the function that had set for it by the system call **sigset()**. If **sigset()**

has not been invoked for *sigtype*, then the system uses the function to which **SIG_DFL** points. **SIG_DFL** terminates the process, just as if it called the function **exit()**. In addition, it dumps core if *sigtype* is any of the following: **SIGQUIT**, **SIGRESET**, **SIGTRAP**, **SIGSEGV**, or **SIGSYS**.

Note that signal **SIGKILL** cannot be held. See the Lexicon entry for **signal()** for a list of recognized signals.

sigrelse() returns zero if all went well. If something went wrong, it returns -1 and sets **errno** to an appropriate value.

See Also

libc, sighold(), sigignore(), signal(), sigpause(), sigset()

Notes

For more information on the **sigset()** family of signal-handling system calls, see the Lexicon entry for **sigset()**.

sigset() — System Call (libc)

Specify action to take upon receipt of a given signal
#include <signal.h>
void (*sigset (*sigtype*, *function*))()
int *sigtype***;**
void (**function*)();

sigset() tells the signal handler what to do when the current process receives signal *sigtype*.

sigtype identifies the signal being sought. For a list of recognized signals, see the Lexicon entry for **signal()**. Note that the signal **SIGKILL**, which kills a process, can be neither caught nor ignored.

function points to the function to be executed when *sigtype* is received. This can be a function of your own creation; or you can use one of the following macros, which expand into pointers to system-defined functions:

SIG_DFL
This is the default action. The process terminates just as if it called the function **exit()**. In addition, the system writes a core file in the current working directory if *sigtype* is any of the following: **SIGQUIT**, **SIGSYS**, **SIGTRAP**, **SIGSEGV**, or **SIGSYS**. For more information on core files, see the Lexicon entry **core**.

SIG_IGN
Ignore *sigtype*. The system discards all signals of this type.

SIG_HOLD
Hold *sigtype*. The signal is held until the process calls **sigrelse()** to release it. Once the signal is released, it is processed as defined by **sigset()**. Only one "copy" of *sigtype* can be held at any given time.

If all goes well, **sigset()** returns a pointer to the routine that had previously been in place to process *sigtype*. If something goes wrong (e.g., *sigtype* is not defined in **signal.h**), **sigset()** returns **SIG_ERR** and sets **errno** to an appropriate value.

sigset() Versus signal()

The COHERENT system also include the system call **signal()**, which also handles signals. **signal()** predates **sigset()** and its related functions **sighold()**, **sigignore()**, **sigpause()**, and **sigrelse()**. You should *never* combine **signal()** with the **sigset()** family of functions: use one or the other, but not both.

The **sigset()** functions differ from **signal()** in the way they handle signals while a signal is being processed: **signal()** automatically invokes **SIG_DFL** for *sigtype* while its *function* is executing; whereas **sigset()** and its related functions invoke **SIG_HOLD**.

Thus, with **signal()**, sending signal *sigtype* to a program while that signal's *function* is already executing will trigger the default action, which in most instances is to exit from the program. The signal-handling function itself can call **signal()** to reset the signal-handler to point to itself or another function; however, there remains a brief interval of vulnerability between the time the signal-processing function is called and the time it calls **signal()** to program the signal handler. With **sigset()**, however, if another *sigtype* is received while its *function* processing, the signal handler holds it, and releases it automatically after *function* returns.

sigset() also differs from **signal()** in the way in which the signal-handler is reset once *sigtype* has been processed. With **signal()**, *function* is automatically reset to **SIG_DFL** just before a signal of type *sigtype* is processed. If you wish *sigtype* always to be processed by *function*, you must explicitly re-invoke **signal()** for *sigtype* within *function*. However, the **sigset()** family of routines always process *sigtype* by the routine to which *function* points until you explicitly change it.

See Also

libc, sighold(), sigignore(), signal(), sigpause(), sigrelse()

Notes

Functions called from within a signal handler should be re-entrant; this includes the standard I/O library. Thus, in general, it is not a good idea to call **printf()** from inside a signal handler. The risk is that a signal will arrive while the main program is updating a static structure, or calling **malloc()**; then the signal handler will run when something is not in a consistent state, with unpredictable results.

sigsetjmp() — General Function (libc)

Save machine state and signal mask for non-local jump
#include <setjmp.h>
int sigsetjmp(*environ, savemask***)**
sigjmp_buf *environ***;**
int *savemask***;**

sigsetjmp() performs the same action as the function **setjmp()**, except that if the value of *savemask* is not zero, it saves the process's signal mask as well as the machine state into the array to which *environ* points.

See Also

libc, sigaction(), siglongjmp(), sigprocmask(), sigsuspend()
POSIX Standard, §8.3.1

sigsuspend() — System Call (libc)

Install a signal mask and suspend process
#include <signal.h>
int sigsuspend(*set***)**
const sigset_t ****set*;**

sigsuspend() replaces the process's signal mask with the set of signals to which *set* points, then suspends the current process until it receives a signal that either terminates the process or invokes a signal-handling function.

If the received signal terminates the process, **sigsuspend()** does not return. If, however, the received signal invokes a signal-handling function, **sigsuspend()** restores the original signal mask.

Because **sigsuspend()** indefinitely suspends execution of the process, there is no return value that indicates successful completion. If something goes wrong, it returns -1 and sets **errno** to an appropriate value. **sigsuspend()** fails if either of the following is true:

- The calling process catches a signal and grabs control from the signal-catching function. **sigsuspend()** sets **errno** to **EINTR**.
- *set* points outside the process's allocated address space. **sigsuspend()** sets **errno** to **EFAULT**.

See Also

libc, siglongjmp(), signal(), sigsetjmp()
POSIX Standard, §3.3.7

sin() — Mathematics function (libm)

Calculate sine
#include <math.h>
double sin(*radian***) double** *radian***;**

sin() calculates the sine of its argument *radian*, which must be in radian measure.

Example

The following example uses the functions **sin()** and **cos()** to paint sine and cosine on the screen. It is by Dmitry Gringauz (dmitry@golem.com).

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

```
#define MAX_X 79 /* X dimension of screen */
#define MAX_Y 23 /* Y dimension of screen */
char screen[MAX_X][MAX_Y]; /* the screen matrix */

main()
{
	double pi = 3.14159, i, result;
	int x = 0, y = 0, mid_x = (MAX_X-1)/2, mid_y = (MAX_Y-1)/2;

	/* blank (dot) out the screen */
	for (y = 0; y < MAX_Y; y++)
		for (x = 0; x < MAX_X; x++)
			screen[x][y] = '.';

	/* build the "axis" */
	for (x=0; x < MAX_X; x++)
		screen[x][mid_y] = '-';
	for (y = 0; y < MAX_Y; y++)
		screen[mid_x][y] = '|';

	/* make center and arrows */
	screen[mid_x][mid_y] = '+';
	screen[mid_x][0] = '^';
	screen[MAX_X-1][mid_y] = '>';

	/* do the sin() and cos() thing */
	for (i = -pi; i <= pi; i = i + 2.0 / (MAX_X)) {
		result = sin(i) ;

		x = i*mid_x/pi + mid_x;
		y = mid_y*(-1.0*result) + mid_y;

		if (x >= MAX_X)
			x = MAX_X - 1;

		if (y >= MAX_Y)
			y = MAX_Y - 1;

		screen[x][y] = '*';
		result = cos(i) ;

		x = i*mid_x/pi + mid_x;
		y = mid_y*(-1.0*result) + mid_y;

		if (x >= MAX_X)
			x = MAX_X - 1;

		if (y >= MAX_Y)
			y = MAX_Y - 1;
		screen[x][y] = '*';
	} /* i */

	/* print the screen */
	for (y = 0; y < MAX_Y; y++) {
		for (x = 0; x < MAX_X; x++)
			printf("%c", screen[x][y]);
		printf("\n");
	} /* y */
}
```

See Also

cos(), cosh(), libm, sinh()
ANSI Standard, §7.5.2.6
POSIX Standard, §8.1

sinh() — Mathematics Function (libm)

Calculate hyperbolic sine
#include <math.h>
double sinh(*radian***) double** *radian***;**

sinh() calculates the hyperbolic sine of *radian*, which is in radian measure.

See Also

libm
ANSI Standard, §7.5.3.2
POSIX Standard, §8.1

size — Command

Print size of an object file
size [*file ...***]**

size prints the sizes, in bytes, of the segments of each *file* (in decimal) and also prints the total size of all the segments (in both decimal and octal). Each *file* must be an object file.

size outputs one line for each file, listing the following segments:

.text
.data
.bss

See Also

coff.h, commands, l.out.h

Notes

size makes no concessions to machines that use hexadecimal.

sizeof — C Keyword

Return size of a data element

sizeof is a C operator that returns a constant **int** that gives the size of any given data element. The element examined can be a data object, a portion of a data object, or a type cast. **sizeof** returns the size of the element in **char**s; for example

```
long foo;
sizeof foo;
```

returns four, because a **long** is as long as four **char**s.

sizeof can also tell you the size of an array. This is especially helpful for use with external arrays, whose size can be set when they are initialized. For example:

```
char *arrayname[] = {
      "COHERENT",
      "COHware volume I",
      "COHERENT Device Driver Kit",
        "GNU C/C++"
};

main()
{
      printf("\"arrayname\" has %d entries\n",
            sizeof(arrayname)/sizeof char*);
}
```

sizeof is especially useful in **malloc()** routines, and when you need to specify byte counts to I/O routines. Using it to set the size of data types instead of using a predetermined value will increase the portability of your code.

See Also

C keywords, data types, operators
ANSI Standard, §6.3.3.4

sleep — Command

Stop executing for a specified time
sleep *seconds*

The command **sleep** suspends execution for a specified number of *seconds*. This routine is especially useful with other commands to the shell. For example, typing

```
(sleep 3600; echo coffee break time) &
```

executes the **echo** command in one hour (3,600 seconds) to indicate an important appointment.

See Also

alarm(), commands, ksh, pause(), sh

sleep() — General Function (libc)

Suspend execution for interval
#include <unistd.h>
sleep(*seconds***)**
unsigned *seconds***;**

sleep() suspends execution for not less than *seconds*.

Example

The following example demonstrates how to use **sleep()**:

```
#include <unistd.h>
main()
{
	printf("Waiting for Godot ...\n");

	for ( ; ; ) {
		sleep(5);   /* sleep for five seconds */
		printf("... still waiting ...\n");
	}
}
```

See Also

libc, nap(), unistd.h
POSIX Standard, §3.4.3

Notes

To make a program sleep for less than one second, use the system calls **nap()** or **poll()**. For an example, see the Lexicon article for **poll()**.

smail — Command

Mail delivery system
smail [*flags* **]** *address ...*

smail is the program that receives and delivers mail. It accepts mail from a source either on your local host or on a remote host, and delivers that mail to its destination — again, either on your local host or another remote host. **smail** does not provide a user interface for typing mail or reading it; to do so, you must use a "mailer" program, such as **mail** or **elm**.

You will rarely, if ever, need to invoke **smail** directly. You may modify one of its configuration files from time to time, but **smail** normally is invoked only by other programs. The rest of this article gives **smail**'s command-line options and describes how it works. You will find this information useful should you wish to reconfigure your mail system, or chase down a bug.

smail can be invoked under a variety of names. Each name indicates the major use to which **smail** will be put, e.g., receiving local mail, receiving remote mail, attempting to deliver undelivered mail, or displaying information about undelivered mail. These names are described below; each also has its own Lexicon entry.

Command-line Options

smail recognizes the following command-line options:

-bc Display the contents of file **COPYING**, which is distributed with the source code for **smail**. This file details what your rights and restrictions the authors of **smail** have set upon their work.

-bd Listen for connection requests on a socket bound in the Internet domain. When a connection occurs, conduct an Simple Mail Transfer Protocol (SMTP) conversation with the peer process on the other system. This option currently is not implemented under COHERENT, as COHERENT does not yet support networking.

-bi Initialize the **aliases** file. The file that it builds depends upon whether you also use option **-oA** on the command line.

By default, **smail** under COHERENT is compiled with the **GDBM** package. **GDBM** is a set of functions that permit a program to build and read a simple hashed data base; for details on how it works, see the Lexicon entry for **libgdbm**. Thus, when you also use option **-oA***file* to name an aliases file, **smail** invokes the command **/usr/lib/mail/newaliases** to compile the contents of *file* into a DBM data base.

-bm *address ...*
Deliver mail to each *address*.

-bP *address*
Assume that each *address* on the command line is a configuration-file variable, and write its value onto the standard output. For example, the command

```
smail -bP hostnames max_message_size
```

produces output of the form:

```
lepanto.com
102400
```

If you also use the flags **-d** or **-v** on the command line, **smail** also displays the variable names. Thus, the command

```
smail -bP -v max_message_size
```

prints something like the following:

```
max_message_size=102400
```

The command

```
smail -bP primary_name
```

prints the primary (or “canonical”) name for the local host that **smail** uses, and command

```
smail -bP config_file
```

prints the name of the primary configuration file. The command

```
smail -bP help
```

prints a verbose listing of every variable plus its type, one variable per line. Finally, command

```
smail -bP all
```

prints a verbose listing of every variable and its value. It is equivalent to the command **smail -bP -v** followed by a list of the name of every configuration variable.

-bp List information about the messages that currently reside in **smail**’s input spool directories. This is **smail**’s default mode of operation when you invoke it under the name **mailq**. When you also use the flags **-v** or **-d**, **smail** displays the transaction-log entries for each message, to show what has happened to the message so far.

-bS Read SMTP commands from the standard input, but do not write SMTP replies onto the standard output. Report failures via mail rather than through reply codes.

This option is suitable for setting up a batched form of SMTP between machines over a remote execution service like UUCP. This is the default mode of operation if you invoke **smail** under the name **rsmtp**.

-bs Read SMTP commands from standard input, and write SMTP replies onto the standard output. The following SMTP commands are implemented:

HELO	**MAIL**	**FROM**	**RCPT**
TO	**DATA**	**RSET**	**NOOP**
VRFY	**EXPN**	**QUIT**	

This is the default mode of operation if you invoke **smail** under the name **smtpd**.

-bt Run **smail** in test-address mode: **smail** reads addresses from standard input, parses them, and writes its result onto the standard output. This is primarily useful for debugging **smail** or debugging new **smail** routers.

-bV See option **-V**, below.

-bv Verify an address. **smail** reads each address you list on its command line, subjects it to aliasing and forwarding expansions, then subjects it to host routing or resolving, and finally prints the resolved address onto the standard output. You can then check whether the resolved address matches what you expect. If **smail** cannot resolve an address, it prints an explanation of why it cannot.

-C *file* Use *file* as the primary configuration file — i.e., the file that holds global attributes. **smail** resets the effective user identifier and group identifier to those of the real user and group, to avoid problems should **smail** be **setuid** to the superuser.

If *file* is '-', then **smail** does not use a primary configuration file. You should use this only for debugging.

-d[*number***]**
Turn on debugging. *number* sets the level of debugging; the default level is one. No white space must separate the option and *number*. Please note that **-d** and **-v** are identical; **smail** recognizes both for historical reasons.

-D *file* Write debugging information into *file*. Normally, using option **-v** or **-d** to generate debugging output also disables background delivery of mail, because programs should not continue to write to the standard error after the mail process exits; however, if you name a debugging-output file, background delivery can continue.

-ee
-oee These options refer to a "berkenet" style of error-processing that **smail** does not support. If used, **smail** mails an error message back to you.

-em
-oem Mail error messages to the sender. This is the default.

-ep Write error messages onto the standard-error device.

-eq If an error occurs, do not notify the sender of it. This only works for mail being delivered locally: an error that occurs on a remote host's mail system still generates a mail message to the sender. To set this behavior on both the local host and a remote host, supply a header that reads:

```
Precedence: junk
```

-ew
-oew Mail errors to the sender, just as with option **-m**. With some mail-delivery programs, this option asks the program to invoke the command **write** to write errors onto the sender's screen, should she be logged in.

-F *fullname*
Set to fullname the full name of the sender for incoming mail. Use this option only if you wish to use smail to receive a single mail message from the standard input.

-f *sender*
Set to *sender* the address for incoming mail. Use this option only if you want **smail** to receive a single mail message from the standard input.

-h *number*
Set to *number* the hop count for a message. If this command-line option is not used, **smail** computes the hop count from the number of **Received:** fields in the message's header. **smail** uses the hop count as a primitive method of detecting an infinite loop: if the hop count is too large, **smail** rejects the mail.

NB, an infinite loop occurs when two sites each think that a given user resides on the other. A message mailed to that user will ping-pong between the sites; unless the message is stopped somehow, its header can grow infinitely large.

-I Use the “hidden-dot” algorithm when reading a message. If a message contains a line that begins with one or more periods, **smail** removes that leading period; a line that consists of a single period terminates the message. This option is always set for messages received via SMTP.

-i A line that consists of a single period does not terminate an incoming message. This is the default if you invoke smail under the name **rmail**.

-m If a user mails a message to an alias list or mailing list that includes her name, send a copy of the message to that user. By default, if the user mails a message to a list that includes her name, **smail** does *not* send a copy of a message back to her.

-N Disable delivery of a message. **smail** performs all other processing, and the transport programs are expected to go through most of the steps involved in delivery. Use this option when you wish to debug smail but do not want to have the messages delivered.

-n Do not process aliases. With this option, **smail** will not expand entries in alias files; however, it will still expand entries in mailing-list files and forwarding files.

-oC *file* See option **-C**, above.

-odb If mail is to be delivered, deliver it in the background. Note that background delivery is not currently supported in the SMTP modes: mail is delivered in the foreground.

-oD *file* Use *file* as the **directors** file, instead of the default **/usr/lib/mail/directors**. **smail** resets the effective user and group identifiers to those of the real user and group, to avoid problems should an installation setuid **smail** to the superuser. If file is ‘-’, **smail** does not read a **directors** file. Use this option only when you are debugging **smail**.

-odf If mail is to be delivered, deliver it in the foreground.

-oE *file* Use *file* as the delivery-retry control file, instead of the default **/usr/lib/mail/retry**. **smail** resets the effective user and group identifiers to those of the real user and group, to avoid problems should an installation setuid **smail** to the superuser. If file is ‘-’, **smail** does not read a retry file. Use this option only when you are debugging **smail**.

-oep See option **-ep**, above.

-oeq See option **-eq**, above.

-oI See option **-I**, above.

-oi See option **-i**, above.

-oL *directory*
Use *directory* as the library directory — that is, the directory that holds configuration files and mailing-list directories. This overrides the default value compiled into smail through its option **smail_lib_dir** (under COHERENT **/usr/lib/smail**), as well as any name set in a configuration file.

-oMr *sender_proto*
Use *sender_proto* as the protocol by which sending host delivers the mail message. You can include this value in expansion strings via the variable **$sender_proto**.

-oMs *sender_host*
Set to sender_host the system that can send the mail message. You can include this value in expansion strings via the variable **$***sender_host*.

-om See option **-m**, above.

-oQ *file* Set the path name of the host-name qualification file to *file*, instead of the default **/usr/lib/mail/qualify**. **smail** resets the effective user and group identifiers to those of the real user and group, to avoid problems should an installation setuid **smail** to the superuser. If *file* is ‘-’, **smail** does not read a qualify file. Use this option only when you are debugging **smail**.

-oR *file* Use *file* as the router file, instead of the default **/usr/lib/mail/routers**. **smail** resets the effective user and group identifiers to the real user and group identifiers, to avoid problems should an installation setuid **smail** to the superuser. If file is ‘-’, **smail** does not read a router file. Use this option only when you are debugging **smail**.

-oT *file* Use *file* as the transport file, instead of the default **/usr/lib/mail/transports**. **smail** resets the effective user and group identifiers to those of the real user and group, to avoid problems should an installation setuid **smail** to the superuser. If file is '-', **smail** does not read a transport file. Use this option only when you are debugging **smail**.

-oU Tell **smail** to report memory usage when it exits.

-oX *mail-service*
Tell **smail** to listen for SMTP requests on the TCP/IP service or port *mail-service*. You can use this option with **-bd** mode to define alternate debugging versions of **smail**'s SMTP listening daemon; this can be useful when you test a new installation.

Please note that because COHERENT does not yet support networking, this option does nothing.

-Q
-odq Spool incoming messages, but do not deliver them until later queue. This mode of operation is somewhat more efficient in terms of CPU usage, but slows down the flow of mail.

-q[*interval***]**
Force **smail** to process its input spool directory. If you set *interval*, **smail** continually checks its input-spool directory, and sleeps for *interval* between checks. *interval* is a string that consists of a number followed by one of the following letters to indicate unit of time:

s seconds
m minutes
h hours
d days
w weeks
y years

For example, option **-q2h30m** tells **smail** to check its input spool directory every two hours and 30 minutes. This flag is useful with the **-bd** mode of operation, as it awakens the daemon process after each interval to process the queue. This is **smail**'s default mode of operation when you invoke it under the name **runq**.

-r *sender*
See option **-f**, above.

-t Extract addresses from the **To:**, **Cc:**, and **Bcc:** fields of the message header. This is useful for mailers that do not compute the recipient addresses themselves. In this mode, the addresses given on the command line will not receive mail, even as a result of expanding aliases or forwarding addresses. **smail** ignores this option unless it is in the mode set by command-line option **-bm** (which is the default mode).

-V Print the version smail onto the standard output.

Normal Use

A user agent can submit new mail message by invoking **smail** and passing it a message via the standard input. For example, mailers such as **mail** and **elm** submit mail by invoking **smail** with a command such as

```
smail -em -i address ...
```

Because **smail** also works correctly if invoked as **sendmail**, it is common to install **smail** as **/usr/lib/sendmail**, so that existing binaries on BSD systems, or other systems that currently run **sendmail**, need not be modified to run **smail** instead. This also lets you run applications that have been configured to send mail via **sendmail** without modifying their sources or recompiling.

Some user agents, such as GNU Emacs, may wish to have **smail** decipher the recipient list from the header. These programs can invoke **smail** with a command, such as:

```
smail -em -t -i
```

To receive mail over UUCP, **uuxqt** invokes the command **rmail**, which is a link to **smail**. **rmail** can also be another program that invokes **smail** directly as:

```
smail -em -i -fsender-address recipient address ...
```

An alternative method of receiving mail over UUCP is through the command **rsmtp**, which receives batched SMTP requests. This can be used between two sites running **smail** to gain many of the benefits of the SMTP protocol, such as the ability to use recipient addresses that **uux** cannot correctly pass to a remote **rmail** program. For example, an address that contains quotations marks or spaces cannot be expected to pass correctly over an **uux-rmail** link, but will pass correctly over a **uux-rsmtp** link.

Addressing Under smail

The following describes how **smail** interprets an E-mail address.

smail understands domain-style addresses (e.g., **henry@mwc.com**) UUCP-style path names, (e.g., **mwc!lepanto!henry**), and local addresses (e.g., **henry**). It assumes that an address of the form *user@domain* is a domain address, that an address of the form *host!address* is a UUCP path, and anything else is a local address.

When it parses a mixed address (that is, an address that contains both a '!' and a '@'), **smail** gives precedence to '@' over '!'. Thus, it parses the address **a!b@c** as **(a!b)@c**, rather than **a!(b@c)**, which means that mail addressed to **a!b@c** is forwarded to system **c** instead of to system **a**.

Resolving Addresses

An E-mail address has two forms: internal and external. The internal form of an address is what appears on the **To:** line in the message's header. This is the recipient's address as typed by the person who mailed the message. This is regardless of whether the sender typed the recipient's full address, or typed an alias for the recipient. (For details on how to use aliases to address mail messages, see the Lexicon entry for **aliases**.) The external form of an address (also called the message's *envelope*), is the address that smail passes to the mail-delivery agent (either **uux** or **lmail**).

Resolving is the act of transforming an internal address into an envelope. It has two stages: host resolution and alias resolution. *Host resolution* (also called *routing*) is how **smail** figures out the identity of the computer to which it must send the message. If **smail** determines that the message must be delivered on your local machine, it then applies *alias resolution* (also called *alias expansion*) to the address. If the address proves to be an alias, **smail** expands the alias and again performs host resolution to find the machine to which it should deliver the message. If, however, the address names a user on your local machine, then **smail** hands the message to the local mailer **lmail** for delivery.

Although **smail** understands domain-style addresses (i.e., addresses that contain a '@' and are read from right to left), it can deliver mail only to UUCP paths (i.e., addresses that contain '!' characters and are read from left to right) and local addresses. Thus, it must resolve a domain address into a UUCP path or local address.

To resolve a domain-style address, **smail** must find the route to the most specific part of the domain, as specified in the routing file **/usr/lib/mail/paths**. Two degrees of resolution can occur:

Full Resolution
: **smail** finds the full route to the machine. In this case, **smail** either tacks the user specification onto the end of the machine's UUCP path, or resolves it into a local address, whichever is appropriate.

Partial Resolution
: **smail** finds a route for only the right portion of the domain specification; e.g., for

```
henry@lepanto.mwc.com
```

it finds **mwc.com** but cannot identify **lepanto**. Here, **smail** tacks the complete address (in the form **domain!user**) onto the end of the UUCP path. For example, if **smail** finds that the route to **mwc.com** is via systems **foo**, **bar**, and **baz**, it constructs the path:

```
foo!bar!baz!lepanto.mwc.com!henry
```

This assumes that routing program on system **baz** (perhaps **smail**, perhaps some other program) will recognize the token **lepanto.mwc.com** as being a domain rather than a host.

It is an error to route a partially resolved address to the local host (a null UUCP path), because the local host is responsible for resolving the address more fully.

The command-line option **-r** tells **smail** to attempt to route the first (leftmost) component of a UUCP path, regardless of whether it knows how to send mail directly to a site named further to the right in the path. This is called *always routing*. For example, if a mail message is address to

```
foo!bar!baz!mwc!lepanto!fred
```

option **-r** tells **smail** always to route the mail to **foo**, even if also knows how to route mail to **mwc**.

The command-line option **-R** tells **smail** to route mail to the rightmost host named on a UUCP path. This is called *reroute routing*. Use it if you have a very up-to-date routing table, and wish to bypass some obsolete routing information in the current path.

If file **/usr/lib/mail/paths** does not contain a path to the remote system, **smail** forwards mail to the the host named in the entry **smart_path** in file **/usr/lib/mail/config**. This lets your system depend on another, better informed, system to deliver your mail. Note that before you name another system as your system's **smart_path**, you should get the permission of the person who administers that system. Please note that if you start to forward mail to a system without permission, that system's administrator may forward your mail to the bit bucket.

After **smail** resolves an address, it reparses the address to see if it is now a UUCP path or local address. If the new address turns out to be another domain address, **smail** complains. This error occurs when an address partially resolves to the local host.

By default, **smail** does not alter an explicit UUCP path of any mail message. If the stated path is unusable (i.e., the next host is unknown), then **smail** applies always-routing and attempts to deliver the message to first (leftmost) system named in the UUCP path. If this fails, **smail** then uses reroute-routing and again attempts to deliver the message. If this too fails, **smail** finally attempts to to find a path to a smart-host and passes the mail to it. And if that finally fails, **smail** mails an error message to user who mailed the message, and abandons any further attempt to deliver the message.

Headers

Document RFC822, which governs Internet mail, demands that a mail message contain certain entries in its header. These entries include one that begins with the string **To:**, one that begins with the string **From:**, and one that begins with the string **Date**. If a message's header does not contain one or more of these entries, **smail** inserts it.

Build the From: Line of a Message

The header of a mail message includes a line that begins **From:**. This line names the user who originated the message. This line is extremely important, as it will read by users and programs on the recipient's system to identify the sender, and possibly reply to the message.

smail collapses the **From_** and **>From_** lines within a mail message to generate a simple "from" argument, which it then uses to create its own **From:** line. This processing sometimes is called *from-ming* a message. The following gives the rules for from-ming:

- First, it concatenates all hosts named on remote from lines, separating them from each other by '!'s.
- It appends onto that concatenated address, the address from the last **From_** line.
- If that address is in domain format (i.e., the form *user@domain*), **smail** rewrites it in bang-path format (i.e., the form *domain***!***user*). If a host or domain names the local system, **smail** ignores it.
- Finally, **smail** removes redundant information from the **From_** line.

smail generates its own **From_** line. For mail that is to be forwarded via UUCP, **smail** generates a line of the form remote-from host; *host* is the UUCP host name (not the domain name), so that **From_** can indicate a valid UUCP path, thus leaving the sender's domain address in **From:**.

Undeliverable Mail

smail returns to sender all mail that is undeliverable. A message is declared to be undeliverable if the user is unknown, or if the user resides on an unknown host.

Logging

smail uses two log files:

/usr/spool/smail/log/logfile
: The log of every mail message that your system receives. Please note that if your system is busy, this file will quickly become very large. You should embed the command **/usr/lib/mail/savelog** in **root**'s **cron** file to ensure that this file is truncated and saved regularly. For details on **savelog** or **cron**, see their articles in the Lexicon.

/usr/spool/smail/log/paniclog
: The log of every mail that created a panic situation. If your system is configured properly, this file will never become large.

Registered Domains and Subdomains

You may wish to register your domain with the NIC. This will give you an internationally recognized e-mail address. For more information, send E-mail to **postmaster@internic.net**.

Once you have registered your domain, you can also set up subdomains for other systems, so they can receive information from the Internet via your system. The rest of this section discusses how to describe subdomains to your system, and related topics.

Let's say that you have registered your domain, and you have named it **mydomain**. To route mail systems subordinate to mydomain, do the following:

1. Insert the following entry into **/usr/lib/mail/paths**:

```
.mydomain.com  %s 50
```

This tells **smail** that the local host (i.e., your machine) must resolve that any address that ends in the suffix **.mydomain.com**, or it is an error.

2. For each site in **mydomain**, create two entries in **/usr/lib/mail/paths**. For example, for the site **foo.mydomain.com**, create the entries:

```
foo foo!%s  200
foo.mydomain.com  foo!%s  200
```

If the site bar.mydomain.com is fed by the route **froboz!florp!bar**, insert these lines into **paths**:

```
bar froboz!florp!bar!%s  200
bar.mydomain.com  froboz!florp!bar!%s  200
```

Note that you do not have to register subdomains with the NIC. Once you register the top-level domain with the NIC, you control the entire name space — you can subdomain to your heart's content.

The only restrictions on subdomaining may be with your Internet Nameserver. Most nameservers for UUCP domains publish a "wildcard" mail exchanger (MX) record, that essentially says, "send everything for ***.mydomain.com** to this Internet gateway." However, some nameserver managers require you to register every site in your domain, for which they provide a separate MX record. The advantage of this scheme is that anybody on the Internet who sends mail to your domain immediately receives an error message if the message is addressed to a non-existent site. For details, check with the person who manages your nameserver.

To route for an entire subdomain (e.g., **.subd.mydomain.com**), you must choose a gateway for that domain (e.g., **gateway.subd.mydomain.com**), and then use a line like this:

```
.subd.mydomain.com gateway!%s  200
```

smail automatically chooses the longest subdomain match it can find, so this rule applies before the **.mydomain.com %s** rule.

Note that the gateway need not be in the subdomain itself. You could have a line elsewhere in the **paths** file on **mydomain** that says:

```
gateway.mydomain.com  gateway!%s  200
```

Your main gateway may also have information about machines in subdomains, although this is not necessary. For instance, if your main machine is directly connected to a machine in a subdomain, you may want to put this information into **paths**, so the mail will not go through the gateway for that domain.

For example, the machine **smith.subd.mydomain.com** might be directly connected to your master gateway, **mydomain.com**:

```
smith  smith!%s  200
smith.subd.mydomain.com smith!%s  200
```

Without this rule, mail for **smith** would be queued for forwarding through **gateway.subd.mydomain.com**.

Compatibility With sendmail

smail was designed to be a plug-in replacement for the BSD program **sendmail**, in that external programs can call **smail** in the same way that they call **sendmail** and expect similar results. However, **smail** is completely different internally and has entirely different configuration files. As a result, the option **-o** to **smail** only sets a few configuration parameters that were believed to be commonly used by other programs. Also, for convenience, some new (upper-case only) parameters are defined only in **smail**. **smail** ignores attempts to use this flag to set other options. For a complete list of the **-o** options that **smail** recognizes, see the section on command-line options, above.

For compatibility with other software systems (in particular, the Taylor UUCP package), COHERENT links **smail** to command **/usr/lib/sendmail**. Thus, a program that expects to use **sendmail** can use **smail** without being recompiled or reconfigured.

Configuration Files

For most sites, the configuration compiled into **smail** is sufficient, and thus no configuration files are needed. However, you can use any or all of the following optional configuration files to restructure how **smail** behaves on your system:

/usr/lib/mail/config
General configuration. This file can override compiled-in configuration, including the names of any of the following configuration files.

/usr/lib/mail/directors
Configuration file for **smail** directors, i.e., configured methods for resolving local addresses.

/usr/lib/mail/routers
Configuration file for **smail** routers, i.e., configured methods for resolving or routing to remote hosts.

/usr/lib/mail/transports
Configuration for **smail** transports, i.e., configured methods of mail delivery.

The contents of file **config** dictate how **smail** configures its internal workings — where it looks to find other configuration files, where it should send error messages, and so on. The contents of **routers**, **directors**, and **transports** together tell **smail** how to deliver mail both on your local system and on remote systems. The following describes how these files work together.

When **smail** is given a list of addresses to which a message is to be delivered, it processes the list iteratively until it produces a list of resolved addresses. When an address is resolved, **smail** will know which transport it must use to deliver the message to the person or persons to whom it is addressed, and all data that this transport requires. To accomplish this, smail goes through the following steps:

A. **smail** first parses each address to find a host name, called the *target*, and the remaining part of the address, called the *remainder*.

As a simple example, in the address **tron@uts.amdahl.com**, the host part **uts.amdahl.com** is the target and **tron** is the remainder. Likewise, in the address **sun!amdahl!tron**, the target is **sun** and the remainder is **amdahl!tron**.

B. **smail** then shows each address to directors, in the order given in file **/usr/lib/mail/directors**, until one of the directors says that it knows what to do with that address. That director can either return a new list of addresses, or put the address onto a list of resolved addresses. If new addresses are produced, **smail** places them onto the input list, to be processed from step **A**.

C. When an address has passed through step **B**, **smail** shows it to various routers, in the order given in file **/usr/lib/mail/routers**, until a router can match the target name for the address. If no router can match the complete target, then **smail** selects the router that matches the longest portion of the target. The router names the transport to be used to deliver the message to that address, plus some other information that the transport requires (e.g., the next host and next address values). The information as to which transport to use can come either from the definition of the router, from a method file, or may be specified by the router specifically.

When all addresses have been resolved, **smail** sorts them and passes them to transports. The transport then delivers the message to the addresses it is given.

Host names and local user-names are matched independent of case; for example, “Postmaster”, “POSTMASTER”, and “postmaster” are in effect all the same. In addition, **smail** keeps an internal hash table of all regular recipient addresses, that is, all addresses that do not specify files or shell commands. This table is used to discard duplicate regular recipient addresses. This hash table is independent of case, as well, so that the address **Postmaster@SRI-NIC.ARPA** is considered a duplicate of **postmaster@sri-nic.arpa**.

Other Files and Directories

smail also uses the following configuration files:

/usr/lib/mail/qualify
Configuration file for host-name qualification.
/usr/lib/mail/retry
Optional delivery retry configuration file, i.e., minimum time between retries and maximum time to retry before giving up.

smail reads the following files to learn how to redirect mail locally and to give paths to remote sites:

/usr/lib/mail/aliases
Aliases for mail addresses.
/usr/lib/mail/paths
Paths to remote hosts.
/usr/lib/mail/lists
Mailing-list files.
/usr/spool/mail
The directory that holds each user's mailbox file.
$HOME/.forward
Forwarding address or addresses for a given user.

smail uses the following directories to hold incoming mail messages and its work files:

/usr/spool/smail
Directory that holds spool directories and work files.
/usr/spool/smail/input
Spool directory for incoming messages.
/usr/spool/smail/error
Directory that holds mail messages that failed for a reason that the site administrator should investigate.
/usr/spool/smail/msglog
Directory that holds transaction logs for individual messages.
/usr/spool/smail/lock
This directory holds **smail**'s input lock files.

The following files log **smail**'s activity. Please note that these files will grow without end. Your system's system administrator should check and truncate these files from time to time. She can also use the script **/usr/lib/mail/savelog** to manage these files; for details, see the Lexicon entry for this command:

/usr/spool/smail/log/logfile
A log of **smail**'s transactions.
/usr/spool/smail/log/paniclog
A log of configuration or system errors encountered by **smail**.

See Also

commands, mail [command], mail [overview], mailq, rmail, rsmtp, runq, savelog, smtpd

Diagnostics

If all goes well, **smail** returns zero to the shell when it exits. If an error occurs, it returns a value other than zero. The meaning of each code is set in **smail**'s source file **exitcodes.h**, as follows:

EX_USAGE. . . . . Error in command-line usage
EX_DATAERR. . . . . . Data-format error
EX_NOINPUT . . . . . . Cannot open input file
EX_NOUSER. . . . . . . Addressee unknown
EX_NOHOST . . . . . . Host unknown
EX_UNAVAILABLE. . . Service unavailable
EX_SOFTWARE. . . . . Internal software error
EX_OSERR . . . . System error
EX_OSFILE . . . . Critical OS file missing
EX_CANTCREAT . . . . Cannot create (user) output file
EX_IOERR. . . . . Error in file I/O
EX_TEMPFAIL . . . . . Temporary failure; user can retry
EX_PROTOCOL. . . . . Remote error in protocol
EX_NOPERM . . . . . . Permission denied

If you invoke **smail** with its option **-bd**, then the message

```
bind() failed: address already in use
```

means that another process is already listening to the SMTP socket.

Notes

Many mail bugs are not **smail** bugs. **smail** cannot help it if remote sites trash your mail messages.

Setting the input spool directory processing interval to a period of more than 2,147,483,647 seconds will result in an incorrectly calculated processing interval — and is a rather silly thing to do at any event.

For details on the distribution rights and restrictions associated with this software, see file **COPYING**, which is included with the source code to the **smail** system; or type the command:

```
smail -bc.
```

smtpd — Command

SMTP daemon
/bin/smtpd

The daemon **smtpd** reads SMTP commands from standard input, and writes SMTP replies onto the standard output. The following SMTP commands are implemented:

HELO	MAIL	FROM	RCPT
TO	DATA	RSET	NOOP
VRFY	EXPN	QUIT	DEBUG

See Also

commands, mail [overview], rsmtp, smail

Notes

smtpd is a link to command **smail**.

For details on the distribution rights and restrictions associated with this software, see file **COPYING**, which is included with the source code to the **smail** system; or type the command: **smail -bc**.

smult() — Multiple-Precision Mathematics (libmp)

Multiply multiple-precision integers
#include <mprec.h>
void smult(*a, n, c***)**
mint **a*, **c*; **int** *n*;

smult() multiplies the multiple-precision integer (or **mint**) pointed to by *a* by the integer *n*, which is <= 127. It writes the product into the **mint** pointed to by *c*.

See Also

libmp

SOCKADDRLEN() — Sockets Function (libsocket)

Return length of an address
#include <sys/socket.h>
#include <netinet/in.h>
#include <sys/un.h>
int SOCKADDRLEN(*address***)**
struct sockaddr **address*;

Function **SOCKADDRLEN()** returns the size, in bytes, of *address*->**sa_family**. This helps a program distinguish between a UNIX and an Internet address.

See Also

libsocket

Notes

COHERENT implements **SOCKADDRLEN()** as a function rather than as a macro.

socket() — Sockets Function (libsocket)

Create a socket

#include <sys/types.h>
#include <sys/socket.h>
int socket(*domain***,** *type***,** *protocol***)**
int *domain***,** *type***,** *protocol***;**

socket() creates a "socket" — that is, an endpoint for communication. It returns a descriptor that uniquely identifies the socket.

domain specifies the domain within which communication will take place. This selects the protocol family to be used. These families are defined in **<sys/socket.h>** Currently, **socket()** recognizes the following domains:

AF_UNIX UNIX internal protocols.
AF_INET ARPA Internet protocols.

The socket has the indicated *type*, which specifies the semantics of communication. **socket()** recognizes the following types:

SOCK_STREAM
This type provides a byte stream that is sequenced, reliable, two-way, and connection-based.

SOCK_DGRAM This type supports "datagrams" — that is, connectionless, unreliable messages of a fixed maximum length.

protocol identifies the protocol to be used with the newly created socket. In most instances, a given type of socket supports only one protocol. However, a socket type may support many different protocols, in which case you must specify the one to use. The protocol number to use is particular to the "communication domain" in which communication is to take place.

Sockets of type **SOCK_STREAM** are full-duplex byte streams, similar to pipes. A stream socket must be in a connected to another socket (through a call to function **connect()**) before any data can be sent to it or received on it. Once connected, data can be transferred using the system calls **read()** and **write()**. When a session has been completed, invoke the system call **close()** to close the socket.

If all goes well, **socket()** returns the descriptor of the newly created socket; this is always a positive integer. If something goes wrong, it returns -1 and sets **errno** to an appropriate value. The following lists the possible errors, by the value to which **socket()** sets **errno**:

EPROTONOSUPPORT
type or *protocol* is not supported within this domain.

EMFILE
The per-process descriptor table is full.

ENFILE
The system file table is full.

EACCESS
You do not have permission to create a socket of a given *type* or *protocol*.

ENOBUFS
Not enough buffer space is available. The socket cannot be created until sufficient resources are freed.

See Also

accept(), connect(), libsocket, listen(), read(), write()

socket.h — Header File

Define constants and structures with sockets
#include <sys/socket.h>

Header file **<socket.h>** defines constants, structures, and prototypes used with **sockets**.

See Also

header files, libsocket

socketpair() — Sockets Function (libsocket)

Create a pair of sockets
int socketpair (*family***,** *type***,** *protocol***,** *fds***)**
int *family***,** *type***,** *protocol***,** *fds***[2];**

Function **socketpair()** creates a pair of sockets. *family*, *type*, and *protocol* give the family, type, and protocol of the sockets to be created. At present, *family* must be set to **AF_UNIX**. *fds* gives the address of an array of two integers, into which **socketpair()** writes the file descriptors of the sockets it creates.

If all goes well, **socketpair()** returns zero. If an error occurs, it returns -1 and sets **errno** to an appropriate value.

See Also

libsocket

Notes

socketpair() does not connect the pair of sockets that it creates, so a call to **getpeername()** on one of them will not return the name of the other.

sort — Command

Sort lines of text
sort [-bcdfimnru] [-t *c***] [-o** *outfile***] [-T** *dir***] [+***beg***[-***end***]][***file …***]**

sort reads lines from each *file*, or from the standard input if no file is specified. It sorts what it reads, and writes the sorted material to the standard output.

sort sorts lines by comparing a *key* from each line. By default, the key is the entire input line (or *record*) and ordering is in ASCII order. The key, however, can be one or more *fields* within the input record; by using the appropriate options, you can select which fields are used as the key, and dictate the character that is used to separate the fields.

The following options affect how the key is constructed or how the output is ordered.

-b Ignore leading white space (blanks or tabs) in key comparisons.

-d Dictionary ordering: use only letters, blanks, and digits when comparing keys. This is essentially the ordering used to sort telephone directories.

-f Fold upper-case letters to lower case for comparison purposes.

-i Ignore all characters outside of the printable ASCII range (octal 040-0176).

-n The key is a numeric string that consists of optional leading blanks and optional minus sign followed by any number of digits with an optional decimal point. Ordering is by the numeric, as opposed to alphabetic, value of the string.

-r Reverse the ordering, i.e., **sort** from largest to smallest.

As noted above, the key compared from each line need not be the entire input line. The option **+***beg* indicates the beginning position of the key field in the input line, and the optional -*end* indicates that the key field ends just before the *end* position. If no -*end* is given, the key field ends at the end of the line. Each of these positional indicators has the form **+***m.nf* or -*m.nf*, where *m* is the number of fields to skip in the input line and *n* is the number of characters to skip after skipping fields. Optional flags *f* are chosen from the above key flags (**bdfinr**) and are local to the specified field.

The following additional options control how **sort** works.

-c Check the input to see if it is sorted. Print the first out-of-order line found.

-m Merge the input files. **sort** assumes each *file* to be sorted already. With large files, **sort** runs much faster with this option.

-o *outfile*
Put the output into *outfile* rather than on the standard output. This allows **sort** to work correctly if the output file is one of the input files.

-t*c* Use the character *c* to separate fields rather than the default blanks and tabs. For example, **-t/** uses the slash instead of white space to separate fields; this is useful when sorting file names and directory names.

-T *dir*
Create temporary files in directory *dir* rather than the standard place.

-u Suppress multiple copies of lines with key fields that compare equally.

The following example sorts the password file **/etc/passwd,** first by group number (field 4) and then by user name (field 1):

```
sort -t: +3n -4 +0 -1 /etc/passwd
```

Limits

The COHERENT implementation of **sort** sets the following limits on input and output:

Characters per input record	399
Characters per output record	399
Characters per field	399

Files

/usr/tmp/sort* — First attempt at temporary files
/tmp/sort* — Second attempt at temporary files

See Also

ASCII, commands, ctype.h, qsort(), shellsort(), tsort, uniq

Diagnostics

sort returns a nonzero exit status if internal problems occurred, or if the file was not correctly sorted in the case of the **-c** option.

spac — Command

Sort a file system
spac *raw_device*

The command **spac** uses the default **dpac** sorting algorithm to re-organize file system *raw_device*.

See Also

commands, dpac, fmap, fsck, qpac, upac

Notes

spac is a link to the command **dpac**. **spac** was written by Randy Wright (rw@rwsys.wimsey.bc.ca).

spell — Command

Find spelling errors
spell [-a][-b][*file ...***]**

spell builds a set of unique words from a document contained in each input *file*, or the standard input if none. It writes a list of words believed to be misspelled onto the standard output.

spell should normally be invoked with the document in the form of the input to the text formatter **nroff** rather than the output. **spell** deletes control information to the formatter by invoking **deroff**.

The default dictionary is for American spelling of English. The **-a** option specifies this dictionary explicitly. Under the **-b** option, British spelling is checked. This accepts *favour*, *fibre*, and *travelled* rather than the American spellings *favor*, *fiber*, and *traveled* for the same words. Words ending in *ize* are also accepted when ending in *ise* (e.g., *digitize*, *digitise*).

The dictionary has a reasonably complete coverage of proper names as well as technical terms in certain fields. However, it covers some fields (e.g., computer science) better than others (e.g., medicine).

Looking up a Word

The COHERENT command **look** reads **spell**'s dictionaries to find words that resemble a fraction of a word that you type. For example, the command

```
look consider
```

returns the following to the standard output:

```
consider#
considerable
considerably
considerate
considerately
consideration#
considered
considering
```

The '#' indicates a possible plural form by adding 's' to the end of the word. This lets you check the spelling of a word without having to enter the word into a file and run **spell** on it.

Files

/usr/dict/clista — Compressed American dictionary
/usr/dict/clistb — Compressed British dictionary
/usr/dict/spellhist — History file for dictionary maintainer
/usr/lib/spell

See Also

commands, deroff, look, nroff, sort, typo

Notes

Dictionaries are not provided for languages other than English.

No dictionary can be complete. You must add new words to the dictionary to ensure that it fully meets your needs.

Obscure words (such as opcodes, variable names, etc.) are flagged as spelling errors.

Because the data files required for **spell** are quite large, they might not be installed onto systems with limited disk space. As a result, the command might not work as expected on all systems.

split — Command

Split a text file into smaller files
split [-*lines***][-c***count***][***infile* **[***outfile***]]**

split divides a file into a number of smaller files. This is especially useful for dividing text files into chunks that can be managed by MicroEMACS or similar editors, or for dividing binary files into chunks that can be easily transmitted via UUCP.

split uses *infile* as its input file if given; otherwise, it uses the standard input. If *infile* is '-', **split** uses the standard input.

split puts its output into files with names prefixed by *outfile* and suffixed consecutively with **aa**, **ab**, **ac**, and so on. If no *outfile* is specified, file names are prefixed with **x**.

Normally, **split** puts 1,000 lines in each output file. This default may be changed for text files by the option *-lines*, where *lines* gives the desired number of lines per file. When using **split** on binary files, the argument *count* to the **-c** option lets you specify the number of characters to place into each output file.

See Also

commands

spow() — Multiple-Precision Mathematics (libmp)

Raise multiple-precision integer to power
#include <mprec.h>
void spow(*a, n, b***)**
mint **a*, **b*; **int** *n*;

spow() raises the multiple-precision integer (or **mint**) pointed to by *a* to the power of *n*, and writes the result into the **mint** pointed to by *b*. In no case may the exponent be negative.

See Also

libmp

sprintf() — STDIO Function (libc)

Format output
#include <stdio.h>
int sprintf(*string***,** *format* [**,** *arg*] ...**)**
char **string***,** **format***;**

sprintf() formats and prints a string. It resembles the function **printf()**, except that it writes its output into the memory location pointed to by *string*, instead of to the standard output.

sprintf() reads the string pointed to by *format* to specify an output format for each *arg*; it then writes every *arg* into *string*, which it ends with a null character. For a detailed discussion of **sprintf()**'s formatting codes, see **printf()**.

If it wrote the formatted string correctly, **sprintf()** returns the number of characters written. Otherwise, it returns a negative number.

Example

For an example of this function, see the entry for **sscanf()**.

See Also

printf(), fprintf(), libc, vsprintf()
ANSI Standard, §7.9.6.5
POSIX Standard, §8.1

Notes

The output *string* passed to **sprintf()** must be large enough to hold all output characters.

Because C does not perform type checking, it is essential that each argument match its format specification.

sqrt() — Mathematics Function (libm)

Compute square root
#include <math.h>
double sqrt(*z***) double** *z***;**

sqrt() returns the square root of *z*.

Example

The following program prints all prime numbers between one and a positive integer that the user enters. It was written by Michael B. Young (myoung@mcs.csuhayward.edu).

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

int main()
{
      int i, userinput;

      /* get user input */
      fprintf(stderr, "Enter an integer value greater than 2:  ");
      scanf("%d", &userinput);
```

```
	if (userinput < 3) {
		fprintf(stderr, "Error:  enter a positive integer > 2\n");
		exit(EXIT_FAILURE);
	}

	/* test for all numbers between one and "userinput". */
	/* for efficiency's sake, even numbers are not tested. */
	/* two is the only even prime number */

	printf("%d\n", 2);
	for (i = 3; i < userinput; i += 2)
		if (prime(i))
			printf("%d\n", i);

	exit(EXIT_SUCCESS);
}

/*
 *  function prime() - tests the passed integer testvalue for "prime-ness"
 *  by testing whether each integer between 1 and the square root of
 *  testvalue divides evenly into testvalue.  Returns 1 if prime, 0 if not.
 */
int prime(testvalue)
int testvalue;
{
	int end, j, result;

	end = (int) sqrt ( (double) testvalue );
	for (j = 2, result = 1; result == 1 && j <= end; j++) {
		if ((testvalue % j) == 0)
			result = 0;
	}
	return result;
}
```

See Also

cos(), cosh(), libm, sin()
ANSI Standard, §7.5.5.2
POSIX Standard, §8.1

Diagnostics

When a domain error occurs (i.e., when *z* is negative), **sqrt()** sets **errno** to **EDOM** and returns zero.

srand() — Random-Number Function (libc)

Seed random number generator
#include <stdlib.h>
void srand(*seed*) int *seed*;

srand() uses *seed* to initialize the sequence of pseudo-random numbers returned by **rand()**. Different values of *seed* initialize different sequences.

Example

For an example of this function, see the entry for **rand()**.

See Also

libc, rand(), stdlib.h
The Art of Computer Programming, vol. 2
ANSI Standard, §7.10.2.2
POSIX Standard, §8.1

Notes

For a superior but non-standard random-number generator, see the function **randl()**, described in the Lexicon article **libmisc**.

srand() cannot be used with any of the “rand48” functions. For an overview of these functions, see the entry for **srand48()**.

srand48() — Random-Number Function (libc)

Seed the 48-bit pseudo-random number routines
void srand48(*seedval***)**
long seedval;

Computation of 48-bit pseudo-random numbers uses two 48-bit integers and one 16-bit integer. One of the 48-bit values holds the "seed" value from which the 48-bit pseudo-random value is computed. This seed can be set explicitly, or is the previously computed pseudo-random number. The other 48-bit integer holds the multiplier from which the pseudo-random number is computed; and the 16-bit integer gives holds the addend.

Function **srand48()** builds the 48-bit "seed" value from a long integer. The 32 bits of the long integer comprise the high 32 bits of the seed; the low 16 bits are filled with the value 0x33E.

Functions **lcong48()** and **seed48()** can also be used to seed the routines that generate 48-bit pseudo-random numbers. **srand48()** returns nothing.

See Also

drand48(), erand48(), jrand48(), libc, lcong48(), lrand48(), mrand48(), nrand48(), seed48()

srandom() — Sockets Function (libsocket)

Seed the random-number generator
int srandom(*seed***)**
int *seed***;**

The function **srandom()** "seeds" the random-number generator with value *seed*. It is a synonym for **srand()**.

srandom() does not return a meaningful value.

See Also

libsocket, srand()

srcpath — Command

Find source files
srcpath [-aw] [-p *path***]** *filename pattern* **...**

The command **srcpath** expands the environmental variable **SRCPATH**, applies it to each argument, and prints the full path of each unique result.

An argument can either be a file name or a pattern. For example, the command

```
srcpath "*.[ch]"
```

finds all **.c** and **.h** files on **SRCPATH**. By default. **srcpath** keeps only the first file that it finds with a given name. **srcpath** automatically appends '.' to the beginning of **SRCPATH** so files in the current directory have precedence.

srcpath recognizes the following command-line options:

-p *path*
Use *path* as its path instead of **SRCPATH**. For example,

```
srcpath -p ".:/usr/src/cmd" "*.c"
```

tells **srcpath** to search '.' and **/usr/src/cmd** instead of **SRCPATH**. Note that with this option, **srcpath** does not automatically place '.' at the beginning of the list.

-a Disable shadowing. Normally, if **srcpath** finds a file is found in more than one directory on the path, it prints only the first. The **-a** option forces **srcpath** to print all instances of the file name.

-w By default, **srcpath** silently bypasses directories and matching files for which it has no read permission. The **-w** option causes it to print a warning message when this happens.

See Also

commands, find, make, PATH

sscanf() — STDIO Function (libc)

Format a string
#include <stdio.h>
int sscanf(*string***,** *format* [**,** *arg*] ...**)**
char **string***; char** **format***;**

sscanf() reads the argument *string*, and uses *format* to specify a format for each *arg*, each of which must be a pointer. For more information on **sscanf()**'s conversion codes, see **scanf()**.

Example

This example uses **sprintf()** to create a string, and then reads it with **sscanf()**. It also illustrates a common problem with this routine.

```
#include <stdio.h>

main()
{
	char string[80];
	char s1[10], s2[10];

	sprintf(string, "1234567890123456789012345678901234567890");
	sscanf(string, "%9c", s1);
	sscanf(string, "%10c", s2);

	printf("\n%s is the string\n", string);
	printf("%s: first 9 characters in string\n", s1);
	printf("%s: first 19 characters in string\n", s2);
}
```

See Also

fscanf(), libc, scanf()
ANSI Standard, §7.9.6.6
POSIX Standard, §8.1

Diagnostics

sscanf() returns the number of arguments filled. It returns zero if no arguments can be filled or if an error occurs.

Notes

Because C does not perform type checking, an argument must match its format specification. **sscanf()** is best used only to process data that you are certain are in the correct data format, such as data that were written with **sprintf()**.

sscanf() is difficult to use correctly, and incorrect usage can create serious bugs in programs. It is recommended that you use **strtok()** instead.

stack — Definition

The **stack** is the segment of memory that holds function arguments, local variables, function return addresses, and stack frame linkage information.

If your program uses recursive algorithms, or declares large amounts of automatic data, or simply contains many levels of functions calls, the stack may "overflow", and overwrite the program data. Note that this is unlikely with COHERENT, because the 80386 has implemented dynamic stack allocation.

See Also

Programming COHERENT

standard error — Definition

The **standard error** is the peripheral device or file where programs write error messages by default. It is defined in the header file **stdio.h** under the abbreviation **stderr**, and by default is the computer's monitor.

The shell lets you redirect into a file all text written to the standard error device. To do so, use the shell operator **2>**. For example

```
	make 2>errorfile
```

redirects all error messages generated by **make** into file **errorfile**.

See Also

Programming COHERENT, stderr, stdio.h

standard input — Definition

The **standard input** is the device or file from which data are accepted by default. It is defined in the header file **stdio.h** under the abbreviation **stdin**, and will be the computer's keyboard unless redirected by the operating system, a shell, or **freopen**.

The shell lets you redirect the standard input device. To do so, use the shell operator **<**. For example

```
mail fwb <textfile
```

the standard input device from your terminal to file **textfile**; in effect, this commands mails the contents of **textfile** to user **fwb**.

See Also

Programming COHERENT, stdin, stdio.h

standard output — Definition

The **standard output** is the device or file where programs write output by default. It is defined in the header file **stdio.h** under the abbreviation **stdout**, and in most instances is defined to be the computer's monitor.

The shell lets you redirect into a file all text written to the standard output device. To do so, use the shell operator **>**. For example

```
sort myfile >sortfile
```

redirects the text output by **sort** into file **sortfile**.

See Also

Programming COHERENT, stdio.h, stdout

stat() — System Call

Find file attributes
#include <sys/stat.h>
int stat(*file, statptr***)**
char **file***; struct stat** **statptr***;**

stat() returns a structure that contains the attributes of a file, including protection information, file type, and file size.

file points to the path name of file. *statptr* points to a structure of the type **stat**, as defined in the header file **stat.h**. For information on **stat**, see the Lexicon entry for **stat.h**.

Example

The following example uses **stat()** to print a file's status.

```
#include <sys/stat.h>
main()
{
	struct stat sbuf;
	int status;

	if (status = stat("/usr/include", &sbuf)) {
		printf("Can't find\n");
		exit(EXIT_FAILURE);
	}

	printf("uid = %d gid = %d\n", sbuf.st_uid, sbuf.st_gid);
}
```

See Also

chmod(), chown(), libc, ls, open(), stat.h
POSIX Standard, §5.6.2

Diagnostics

stat() returns -1 if an error occurs, e.g., the file cannot be found. Otherwise, it returns zero.

Notes

stat() differs from the related function **fstat()** mainly in that **fstat()** accesses the file through its descriptor, which was returned by a successful call to **open()**, whereas **stat()** takes the file's path name and opens it before checking its status.

The call

```
stat("", &s)
```

is identical to

```
stat(".", &s)
```

Both calls succeed. The POSIX Standard forbids the former call — in fact, the POSIX Standard forbids the NULL string as a path name under any circumstances; therefore you should never use the former call.

stat.h — Header File

Definitions and declarations used to obtain file status
#include <sys/stat.h>

stat.h is a header file that declares the structure **stat** plus constants used by the routines that manipulate files, directories, and named pipes. It holds the prototypes for the routines **chmod()**, **fstat()**, **mkdir()**, **stat()**, and **umask()**.

The following summarizes the structure **stat**:

```
struct stat {
        dev_t                   st_dev;                 /* Device */
        ino_t                   st_ino;                 /* Inode number */
        mode_t                  st_mode;                /* Mode */
        nlink_t                 st_nlink;               /* Link count */
        uid_t                   st_uid;                 /* User id */
        gid_t                   st_gid;                 /* Group id */
        dev_t                   st_rdev;                /* Real device; NB, this is non-POSIX */
        off_t                   st_size;                /* Size */
        time_t                  st_atime;               /* Access time */
        time_t                  st_mtime;               /* Modify time */
        time_t                  st_ctime;               /* Change time */
};
```

st_dev and **st_ino** together form a unique description of the file. The former is the device on which the file and its i-node reside, whereas the latter is the index number of the file. **st_mode** gives the permission bits, as outlined below. **st_nlink** gives the number of links to the file. **st_uid** and **st_gid,** respectively given the user id and group id of the owner. **st_rdev**, valid only for special files, holds the major and minor numbers for the file. **st_size** gives the size of the file, in bytes. For a pipe, the size is the number of bytes waiting to be read from the pipe.

Three entries for each file give the last occurrences of various events in the file's history. **st_atime** gives time the file was last read or written to. **st_mtime** gives the time of the last modification, write for files, create or delete entry for directories. **st_ctime** gives the last change to the attributes, not including times and size.

The following manifest constants define file types:

S_IFMT	Type
S_IFDIR	Directory
S_IFCHR	Character-special file
S_IFPIP	Pipe
S_IFIFO	Pipe
S_IFBLK	Block-special file
S_IFREG	Regular file

The following manifest constants define file modes:

S_IREAD	Read permission, owner
S_IWRITE	Write permission, owner
S_IEXEC	Execute/search permission, owner
S_IRWXU	RWX permission, owner
S_IRUSR	Read permission, owner
S_IWUSR	Write permission, owner
S_IXUSR	Execute/search permission, owner
S_IRWXG	RWX permission, group
S_IRGRP	Read permission, group
S_IWGRP	Write permission, group
S_IXGRP	Execute/search permission, group
S_IRWXO	RWX permission, other
S_IROTH	Read permission, other
S_IWOTH	Write permission, other
S_IXOTH	Execute/search permission, other

See Also

chmod(), fstat(), header file, stat()
POSIX Standard, §5.6.1

statfs() — System Call (libc)

Get information about a file system
#include <sys/types.h>
#include <sys/statfs.h>
int statfs (*path***,** *buffer***,** *length***,** *fstype***)**
char **path***;**
struct statfs **buffer***;**
int *length***,** *fstype***;**

The COHERENT system call **statfs()** returns information about a file system, either mounted or unmounted.

buffer points to a structure of type **statfs**, which contains the following members:

```
short       f_fstyp;        /* type of the file system */
short       f_bsize;        /* block size */
short       f_frsize;       /* fragment size */
long        f_blocks;       /* number of blocks in the file system */
long        f_bfree;        /* number of free blocks */
long        f_files;        /* number of file nodes */
long        f_ffree;        /* number of free file nodes */
char        f_fname[6];     /* name of the volume */
char        f_fpack[6];     /* name of the pack */
```

length is the length of the area into which **statfs()** can write its output. This should always be set to **sizeof(struct statfs)**.

path and *fstype* identify the file system. If the file system is unmounted, then *path* should name the device by which the file system is accessed, and *fstype* should contain the type of the file system. If the file system is mounted, then *path* should give the full path name of a file on the file system in question, and *fstype* must be set to zero.

statfs() returns zero if all went well. If something went wrong, it returns -1 and sets **errno** to an appropriate value.

See Also

fstatfs(), libc, mkfs, statfs.h, ustat()

static — C Keyword

Declare storage class

static is a C storage class. It has two entirely different meanings, depending upon whether it appears inside or outside a function.

Outside a function, **static** means that the function or variable it preceeds may not be seen outside the module.

Inside a function, static may only precede a variable. It means that that variable is permanently allocated, rather than allocated on the stack when the function is entered and discarded when the function exits. If a **static** variable is initialized, that occurs before the program starts rather than every time the function is entered. If a function returns a pointer to a variable, often that variable is declared **static** within the function. If a pointer to a **non-static** local variable is returned, that variable is freed when the function returns and the pointer points to an unprotected location.

Example

The following example demonstrates the uses of the **static** keyword. It returns the next integer in a sequence as a string.

```
/* static to keep function hidden outside of this module */
static char *nextInt()
{
      /* static to protect value between calls */
      static int next = 0;
      /* static to allow the return of a pointer to s */
      static char s[5];

      sprintf(s, "%d", next++);
      return(s);
}
```

See Also

auto, C keywords, extern, register variable, storage class
ANSI Standard, §6.5.1

stdarg.h — Header File

Header for variable numbers of arguments
#include <stdarg.h>

stdarg.h is the header file that ANSI C uses to declare and define the routines that traverse a variable-length argument list. It declares the type **va_list** and defines the macros **va_arg()**, **va_start()**, and **va_end()**.

Example

The following example concatenates multiple strings into a common allocated string and returns the string's address. method of handling variable arguments:

```
#include <stdarg.h>
#include <stdlib.h>
#include <stddef.h>
#include <stdio.h>

char *
multcat(numargs)
int numargs;
{
      va_list argptr;
      char *result;
      int i, siz;

      /* get size required */
      va_start(argptr, numargs);
      for(siz = i = 0; i < numargs; i++)
            siz += strlen(va_arg(argptr, char *));

      if ((result = calloc(siz + 1, 1)) == NULL) {
            fprintf(stderr, "Out of space\n");
            exit(EXIT_FAILURE);
      }
      va_end(argptr);

      va_start(argptr, numargs);
      for(i = 0; i < numargs; i++)
            strcat(result, va_arg(argptr, char *));
      va_end(argptr);
      return(result);
}
```

```
int
main()
{
	printf(multcat(5, "One ", "two ", "three ",
		"testing", ".\n"));
}
```

See Also

header files, varargs.h
ANSI Standard, §7.8

Notes

The routines defined in **<stdarg.h>** were first implemented under UNIX System V, where they are declared in the header file **<varargs.h>**. The ANSI C committee recognized the usefulness of **<varargs.h>**, but decided that it had semantic problems. In particular, **<varargs.h>** introduced the notion of declaring "..." for the variable-arguments argument list in the function prototype. This, unfortunately, left them with declarations of the form

```
	void error(...)
	{
		whatever
	}
```

and no obvious hook for accessing the parameter list within the body of the function. So, the ANSI committee changed the header declaration: it insisted on one or more formal parameters, followed by the list of variables.

The committee had the wisdom to change the name of its header file, hence **<stdarg.h>** came into being. Unfortunately, the committee kept the same macro names, but in one macro (**va_start()**) changed the number of arguments it takes.

COHERENT includes both **<varargs.h>** and **<stdarg.h>**, to support both ANSI and System-V code.

stddef.h — Header File

Header for standard definitions
#include <stddef.h>

stddef.h defines types and macros that are used through the library.

See Also

header files, offsetof()
ANSI Standard, §7.1.6

stderr — Definition

stderr is the name of the **FILE** pointer assigned to the standard error device. It is set in the header file **stdio.h**.

See Also

Programming COHERENT, stdin, stdio.h, stdout, standard error
ANSI Standard, §4.9.1, §4.9.3

stdin — Definition

stdin is the name of the **FILE** pointer that is assigned to the standard input device. It is set in the header file **stdio.h**.

See Also

Programming COHERENT, standard input, stderr, stdio.h, stdout
ANSI Standard, §7.9.1

STDIO — Definition

STDIO is an abbreviation for *standard input and output*. It refers to a set of standard library functions that accompany all C compilers and that govern input and output with peripheral devices. For details on the STDIO routines, see the Lexicon entries for **libc** and **stdio.h**.

See Also

libc, Programming COHERENT, stdio.h
ANSI Standard, §4.9

stdio.h — Header File

Declarations and definitions for I/O

stdio.h is a header file that defines manifest constants used in standard I/O, prototypes the STDIO functions, and defines numerous I/O macros, as follows:

Types

FILE . . . . Descriptor of file used by STDIO routines
stderr. . . . . Standard error device (by default, the screen)
stdin . . . . Standard input device (by default, the keyboard)
stdout . . . . Standard output device (by default, the screen)

Manifest Constants

BUFSIZ. . . . . Default buffer size
EOF. . . . . End of file
FILENAME_MAX . . . . Maximum length of a file name
FOPEN_MAX . . . . . . Maximum number of of open files
L_ctermid . . . . Length of **ctermid()**
L_tmpnam. . . . . Length of a temporary file name
P_tmpdir. . . . . Default directory for temporary files
TMP_MAX . . . . Maximum number of temporary file names

Functions and Macros

clearerr(). . . . . Present status stream
fclose() . . . . Close a file stream
fdopen() . . . . Open a file stream for I/O
feof() . . . . Discover a file stream's status
ferror() . . . . Discover a file stream's status
fflush() . . . . Flush an output buffer
fgetc(). . . . . Get a character
fgetpos() . . . . Read the file-position indicator
fgets(). . . . . Get a string
fgetw() . . . . Get a word
fileno() . . . . Get a file descriptor from a **FILE** structure
fopen() . . . . Open a file stream
fprintf(). . . . . Format and print to a file stream
fputc() . . . . Output a character
fputs() . . . . Output a string
fputw() . . . . Output a word
fread() . . . . Read a file stream
freopen() . . . . Open a file stream
fscanf() . . . . Format and read from a file stream
fseek() . . . . Seek in a file stream
fsetpos() . . . . Set the file-position indicator
ftell() . . . . Return file pointer position
fwrite() . . . . Write to a file stream
getc() . . . . Get a character
getchar() . . . . Get a character
gets() . . . . Get a string
getw(). . . . . Get a word
pclose(). . . . . Close a pipe
popen() . . . . Open a pipe
printf() . . . . Print a formatted string
putc() . . . . Output a character
putchar(). . . . . Output a character
puts() . . . . Output a string
putw(). . . . . Output a word
rewind() . . . . Reset a file pointer
scanf() . . . . Format and input from standard input

setbuf() . . . . Set alternative file-stream buffer
setvbuf() . . . . Set alternative file-stream buffer
sprintf() . . . . Format and print to a string
sscanf(). . . . . Format and read from a string
tmpfile() . . . . Create a temporary file
ungetc() . . . . Return character to file stream
vfprintf() . . . . Format and print to a file stream
vprintf() . . . . Print a formatted string
vsprintf(). . . . . Format and print to a string

See Also

header file, libc, STDIO
ANSI Standard, §7.9

Notes

COHERENT release 4.2 has rewritten its version of **stdio.h** so that it conforms to the ANSI Standard. For this reason, program that use STDIO and are compiled under COHERENT release 4.2 (or subsequent releases) will not run correctly under versions of COHERENT prior to release 4.2.

stdlib.h — Header File

Declare/define general functions
#include <stdlib.h>

stdlib.h is a header file that is defined in the ANSI Standard. It declares a set of general utilities and defines attending macros and data types, as follows.

Types

div_t . . . . Type of object returned by **div**
ldiv_t . . . . Type of object returned by **ldiv**

Manifest Constants

EXIT_FAILURE . . . . . Value to indicate that program failed to execute properly
EXIT_SUCCESS. . . . . Value to indicate that program executed properly
MB_CUR_MAX . . . . . Largest size of multibyte character in current locale
MB_LEN_MAX. . . . . . Largest overall size of multibyte character in any locale
RAND_MAX . . . . Largest size of pseudo-random number

Functions

abort() . . . . End program immediately
abs(). . . . . Compute the absolute value of an integer
atof() . . . . Convert string to floating-point number
atoi() . . . . Convert string to integer
atol() . . . . Convert string to long integer
bsearch(). . . . . Search an array
calloc() . . . . Allocate dynamic memory
div(). . . . . Perform integer division
exit() . . . . Terminate a program gracefully
free() . . . . De-allocate dynamic memory to free memory pool
getenv() . . . . Read environmental variable
labs() . . . . Compute the absolute value of a long integer
ldiv() . . . . Perform long integer division
malloc() . . . . Allocate dynamic memory
qsort() . . . . Sort an array
rand(). . . . . Generate pseudo-random numbers
realloc() . . . . Reallocate dynamic memory
srand() . . . . Seed the random-number generator
strtod(). . . . . Convert string to floating-point number
strtol() . . . . Convert string to long integer
strtoul() . . . . Convert string to unsigned long integer
system() . . . . Suspend a program and execute another

See Also

header files
ANSI Standard, §7.10

stdout — Definition

stdout is the name of the **FILE** pointer that is assigned to the standard output device. It is set in the header file **stdio.h**.

See Also

Programming COHERENT, standard output, stderr, stdin, stdio.h
ANSI Standard, §7.9.1

sticky bit — Definition

The *sticky bit* is one of the mode bits associated with a file. If the sticky bit is set for an executable file and swapping is enabled, COHERENT behaves in a special way when it executes that file.

When the COHERENT system executes the file the first time, all proceeds normally. When the program exits, however, the pure segments are left on the swap device; when the program is re-invoked, COHERENT reads "pure" code (text) areas from the swap device and all other (impure) segments from the file system. This speeds execution of large programs that are executed frequently.

This strategy works well on systems that have large swap devices. Because overuse of the sticky bit would quickly swamp the swap device, only the superuser can set the sticky bit.

See Also

chmod, Using COHERENT

stime() — System Call (libc)

Set the time
#include
int stime(*timep***)**
time_t ****timep***;**

stime() sets the system time. *timep* points to a variable of type **time_t**, which contains the number of seconds since midnight GMT of January 1, 1970.

If all goes well, **stime()** zero. If a problem occurs, it returns -1.

stime() is restricted to the superuser.

Files

<sys/types.h>

See Also

ctime(), date, ftime(), libc, stat(), utime()

storage class — Definition

Storage class refers to the part of a declaration that indicates how data are to be stored. The C language recognizes the following storage clases:

> **auto**
> **extern**
> **register**
> **static**

typedef is technically defined as a storage class as well, but it does not actually indicate how data are stored. The default class is **auto**.

See Also

auto, extern, Programming COHERENT, register, static, typedef

store() — DBM Function (libgdbm)

Write a record into a DBM data base
#include <dbm.h>
int store(*key***,** *datum***)**
datum *key***,** *datum***;**

Function **store()** writes a record into the currently open DBM data base. The data base must first have been opened by a call to **dbminit()**.

key points to the key by which the datum is identified. *datum* points to the datum itself. If the data base already contains a record with *key*, **store()** overwrites it.

The sizes of *key* and *datum* together must not exceed **BSIZE** bytes — that is, the size of one file-system block. (**BSIZE** is defined in header file **<sys/buf.h>**.)

If all goes well, **store()** returns zero. If an error occurs, it returns a negative value.

See Also

Notes

For a statement of copyright and permissions on this routine, see the Lexicon entry for **libgdbm**.

strcasecmp() — Sockets Function (libsocket)

Case-insensitive string comparison
int strcasecmp (*left***,** *right***)**
char **left***,** **right***;**

Function **strcasecmp()** compares strings *left* and *right*. It returns zero if the strings are identical; -1 if *left* is lexigraphically less than (that is, occurs earlier in the alphabet) than *right*; or one if if *left* is lexicographically greater than *right*. Unlike the function **strcmp()**, **strcasecmp()** ignores case when it compares the strings.

See Also

libsocket, strcmp(), string.h

strcasencmp() — Sockets Function (libsocket)

Case-insensitive string comparison
int strcasencmp (*left***,** *right***,** *n***)**
char **left***,** **right***;**
int *n***;**

Function **strcasencmp()** compares the first *n* bytes of strings *left* and *right*. It returns zero if the first *n* bytes of the strings are identical; -1 if *left* is lexigraphically less than (that is, occurs earlier in the alphabet) than *right*; or one if if *left* is lexicographically greater than *right*. Unlike the function **strncmp()**, **strcasencmp()** ignores case when it compares the strings.

See Also

libsocket, strncmp(), string.h

strcat() — String Function (libc)

Concatenate two strings
#include <string.h>
char *strcat(*string1***,** *string2***)**
char **string1***,** **string2***;**

strcat() appends all characters in *string2* onto the end of *string1*. It returns the modified *string1*.

Example

For an example of this function, see the entry for **string.h**.

See Also

libc, string.h, strncat()
ANSI Standard, §7.11.3.2
POSIX Standard, §8.1

Notes

string1 must point to enough space to hold itself and *string2*; otherwise, another portion of the program may be overwritten.

strchr() — String Function (libc)

Find a character in a string
#include <string.h>
char *strchr(*string*, *character*)
char **string*; int *character*;

strchr() searches for the first occurrence of *character* within *string*. The null character at the end of *string* is included within the search. It is equivalent to the COHERENT function **index()**.

strchr() returns a pointer to the first occurrence of *character* within *string*. If *character* is not found, it returns NULL.

Having **strchr()** search for a null character will always produce a pointer to the end of a string. For example,

```
char *string;
assert(strchr(string, '\0') == string + strlen(string));
```

never fails.

See Also

libc, string.h
ANSI Standard, §7.11.5.2
POSIX Standard, §8.1

strcmp() — String Function (libc)

Compare two strings
#include <string.h>
int strcmp(*string1*, *string2*)
char **string1*, **string2*;

strcmp() compares *string1* with *string2* lexicographically. It returns zero if the strings are identical, returns a number less than zero if *string1* occurs earlier alphabetically than *string2*, and returns a number greater than zero if it occurs later. This routine is compatible with the ordering routine needed by **qsort()**.

Example

For examples of this function, see the entries for **string.h** and **malloc()**.

See Also

libc, qsort(), shellsort(), string.h, strncmp()
ANSI Standard, §7.11.4.2
POSIX Standard, §8.1

strcoll() — String Function (libc)

Compare two strings, using locale-specific information
#include <string.h>
int strcoll(*string1*, *string2*)
char **string1*; char **string2*;

strcoll() lexicographically compares the string pointed to by *string1* with one pointed to by *string2*. Comparison ends when a null character is read.

strcoll() compares the two strings character by character until it finds a pair of characters that are not identical. It returns a number less than zero if the character in *string1* is less (i.e., occurs earlier in the character table) than its counterpart in *string2*. It returns a number greater than zero if the character in *string1* is greater (i.e., occurs later in the character table) than its counterpart in *string2*. If no characters are found to differ, then the strings are identical and **strcoll()** returns zero.

See Also

libc, localization, string.h
ANSI Standard, §7.11.4.3

Notes

The string-comparison routines **strcoll()**, **strcmp()**, and **strncmp()** differ from the memory-comparison routine **memcmp()** in that they compare strings rather than regions of memory. They stop when they encounter a null character, but **memcmp()** does not.

strcoll() differs from **strcmp()** and similar functions in that it reads the user's locale, as set by a call to function **setlocale()**, to determine the lexicographic value of each character. For details, see the Lexicon entry for **localization**.

strcpy() — String Function (libc)

Copy one string into another
#include <string.h>
char *strcpy(*string1***,** *string2***)**
char ****string1*, ****string2***;**

strcpy() copies the contents of *string2*, up to the NUL, into the memory to which *string1* points. It returns *string1*.

Example

See **string**.

See Also

libc, memcpy(), string.h, strncpy()
ANSI Standard, §7.11.2.3
POSIX Standard, §8.1

Notes

string1 must point to enough space to hold *string2*, or another portion of the program or operating system may be overwritten.

strcspn() — String Function (libc)

Return length a string excludes characters in another
#include <string.h>
unsigned int strcspn(*string1***,** *string2***)**
char ****string1*, ****string2***;**

strcspn() compares *string1* with *string2*. It then returns the length, in characters, for which *string1* consists of characters *not* found in *string2*.

See Also

libc, string.h
ANSI Standard, §7.11.5.3
POSIX Standard, §8.1

strdup() — String Function (libc)

Duplicate a string
#include <string.h>
char *strdup(*string***)**
char ****string*;**

The string function **strdup()** duplicates the text to which *string* points. It calls **malloc()** to allocate memory for the duplicate, copies the string, and returns a pointer to the memory that holds the copy. If something goes wrong, it returns NULL.

See Also

libc, string.h

Notes

strdup() is not part of the ANSI Standard. Using it in your programs may limit their portability.

stream — Definition

The term **stream** is a metaphor for any entity that can be named and from which bits can flow, such as a device or a file. The name "stream" reflects the fact that the C programming environment does not depend upon record descriptors and other devices that predetermine what form data can assume; instead, data from whatever source are conceived as being a flow of bytes whose significance is set entirely by the program that reads them.

For example, whether 16 bits forms an **int**, two **char**s, and should be used as an absolute value or a bit map, is entirely up to the program that receives it. It is also irrelevant to the program that processes these 16 bits whether they come from the keyboard, from a file on disk, or from a peripheral device.

The **FILE** structure holds all of the information needed to manipulate a stream. The STDIO functions can be used to open, close, or reopen a stream; read data from it; or write data to it.

See Also

bit, byte, data formats, file, FILE, Programming COHERENT, stdio.h

stream.h — Header File

Definitions for message facility
#include <stream.h>

stream.h definitions constants and structures used by the routines that implement the COHERENT version of STREAMS.

See Also

header files, STREAMS

STREAMS — Definition

COHERENT implementation of STREAMS

Beginning with release 4.2, COHERENT supports STREAMS. This is a system that helps programmers create system-independent device-drivers. STREAMS replaces most of the kernel-accessible routines that are unique to COHERENT.

For details on the COHERENT implementation of STREAMS, and for summaries of the STREAMS routines, see the manual that comes with release 2.2 of the COHERENT Device-Driver Kit.

To add the STREAMS driver to your kernel (should it not already have it), log in as the superuser **root** and then enter the following commands:

```
cd /etc/conf
streams/mkdev
bin/idmkcoh -o /kernel_name
```

where *kernel_name* names the new kernel to build. Then reboot to invoke the newly built *kernel_name*.

See Also

device drivers, getmsg(), Programming COHERENT, putmsg(), stropts.h

strerror() — String Function (libc)

Translate an error number into a string
#include <string.h>
char *strerror(*error*)
int *error*;

strerror() helps to generate an error message. It takes the argument *error*, which presumably is an error code generated by an error condition in a program, and may return a pointer to the corresponding error message.

The error numbers recognized and the texts of the corresponding error messages are set by COHERENT.

See Also

libc, perror(), string.h
ANSI Standard, §7.11.6.2

Notes

strerror() returns a pointer to a static array that may be overwritten by a subsequent call to **strerror()**.

strerror() differs from the related function **perror()** in the following ways: **strerror()** receives the error number through its argument *error*, whereas **perror()** reads the global constant **errno**. Also, **strerror()** returns a pointer to the error message, whereas **perror()** writes the message directly into the standard error stream.

The error numbers recognized and the texts of the messages associated with each error number are set by COHERENT. However, **strerror()** and **perror()** return the same error message when handed the same error number.

strftime() — Time Function (libc)

Format locale-specific time
#include <time.h>
size_t strftime(*string***,** *maximum***,** *format***,** *brokentime***)**
char **string***; size_t** *maximum***; const char** **format***;**
    const struct tm **brokentime***;**

The function **strftime()** provides a locale-specific way to print the current time and date. It also gives you an easy way to shuffle the elements of date and time into a string that suits your preferences.

strftime() references the portion of the locale that is affected by the calls

```
setlocale(LC_TIME, locale);
```

or

```
setlocale(LC_ALL, locale);
```

For more information on setting locales, see the entry for **localization**.

string points to the region of memory into which **strftime()** writes the date and time string it generates. *maximum* is the maximum number of characters that can be written into *string*. *string* should point to an area of allocated memory at least *maximum*+1 bytes long; if it does not, reserved portions of memory may be overwritten.

brokentime points to a structure of type **tm**, which contains the broken-down time. This structure must first be initialized by either of the functions **localtime()** or **gmtime()**.

Finally, *format* points to a string that contains one or more conversion specifications, which guide **strftime()** in building its output string. Each conversion specification is introduced by the percent sign '%'. When the output string is built, each conversion specification is replaced by the appropriate time element. Characters within *format* that are not part of a conversion specification are copied into *string*; to write a literal percent sign, use "%%".

strftime() recognizes the following conversion specifiers:

a The locale's abbreviated name for the day of the week.
A The locale's full name for the day of the week.
b The locale's abbreviated name for the month.
B The locale's full name for the month.
c The locale's default representation for the date and time.
d The day of the month as an integer (01 through 31).
H The hour as an integer (00 through 23).
I The hour as an integer (01 through 12).
j The day of the year as an integer (001 through 366).
m The month as an integer (01 through 12).
M The minute as an integer (00 through 59).
p The locale's way of indicating morning or afternoon (e.g, in the United States, "AM" or "PM").
S The second as an integer (00 through 59).
U The week of the year as an integer (00 through 53); regard Sunday as the first day of the week.
w The day of the week as an integer (0 through 6); regard Sunday as the first day of the week.
W The day of the week as an integer (0 through 6); regard Monday as the first day of the week.
x The locale's default representation of the date.
X The locale's default representation of the time.
y The year within the century (00 through 99).
Y The full year, including century.

z The name of the locale's time zone. If no time zone can be determined, print a null string.

Use of any conversion specifier other than the ones listed above will result in undefined behavior.

If the number of characters written into *string* is less than or equal to *maximum*, then **strftime()** returns the number of characters written. If, however, the number of characters to be written exceeds *maximum*, then **strftime()** returns zero and the contents of the area pointed to by *string* are indeterminate.

See Also

asctime(), ctime(), gmtime(), libc, localtime(), time [overview]
ANSI Standard, §7.12.3.5
POSIX Standard, §8.1

string.h — Header File

Declarations for string library
#include <string.h>

string.h is the header that holds the prototypes of all ANSI routines that handle strings and buffers. It declares the following routines:

fnmatch() . . . . Match a string with a normal expression
index() . . . . Search string for a character; use **strchr()** instead
memccpy(). . . . . Copy a region of memory up to a set character
memchr() . . . . Search a region of memory for a character
memcmp(). . . . . Compare two regions of memory
memcpy() . . . . Copy one region of memory into another
memmove() . . . . Copy one region of memory into another with which it overlaps
memset(). . . . . Fill a region of memory with a character
pnmatch() . . . . Match string pattern
strcat() . . . . Concatenate two strings
strcmp() . . . . Compare two strings
strncat() . . . . Append one string onto another
strncmp() . . . . Compare two lengths for a set number of bytes
strcpy(). . . . . Copy a string
strncpy() . . . . Copy a portion of a string
strcoll(). . . . . Compare two strings, using locale information
strcspn() . . . . Return length one string excludes characters in another
strdup(). . . . . Duplicate a string
strerror() . . . . Translate an error number into a string
strlen() . . . . Measure a string
strpbrk() . . . . Find first occurrence in string of character from another string
strchr() . . . . Find leftmost occurrence of character in a string
strrchr() . . . . Find rightmost occurrence of character in a string
strspn(). . . . . Return length one string includes character in another
strstr() . . . . Find one string within another string
strtok() . . . . Break a string into tokens
strxfrm() . . . . Transform a string, using locale information

Example

This example reads from **stdin** up to **NNAMES** names, each of which is no more than **MAXLEN** characters long. It then removes duplicate names, sorts the names, and writes the sorted list to the standard output. It demonstrates the functions **shellsort()**, **strcat()**, **strcmp()**, **strcpy()**, and **strlen()**.

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

#define NNAMES 512
#define MAXLEN 60

char *array[NNAMES];
char first[MAXLEN], mid[MAXLEN], last[MAXLEN];
char *space = " ";

int compare();
```

```
main()
{
      register int index, count, inflag;
      register char *name;

      count = 0;
      while (scanf("%s %s %s\n", first, mid, last) == 3) {
            strcat(first, space);
            strcat(mid, space);
            name = strcat(first, (strcat(mid, last)));
            inflag = 0;

            for (index=0; index < count; index++)
                  if (strcmp(array[index], name) == 0)
                        inflag = 1;

            if (inflag == 0) {
                  if ((array[count] =
                        malloc(strlen(name) + 1)) == NULL) {
                        fprintf(stderr, "Insufficient memory\n");
                        exit(EXIT_FAILURE);
                  }
                  strcpy(array[count], name);
                  count++;
            }
      }

      shellsort(array, count, sizeof(char *), compare);
      for (index=0; index < count; index++)
            printf("%s\n", array[index]);
      exit(EXIT_SUCCESS);
}

compare(s1, s2)
register char **s1, **s2;
{
      return(strcmp(*s1, *s2));
}
```

See Also

header files, libc, strcasecmp(), strcasencmp()
ANSI Standard, §7.1.1

Notes

Some implementations of UNIX call this header file **strings.h**. If you are porting code to COHERENT, you may have to modify the **#include** directives that invoke this header file.

The ANSI standard allows adjacent string literals, e.g.:

```
"hello" "world"
```

Adjacent string literals are automatically concatenated. Thus, the compiler will automatically concatenate the above example into:

```
"helloworld"
```

Because this departs from the Kernighan and Ritchie description of C, it will generate a warning message if you use the compiler's **-VSBOOK** option.

strings — Command

Print all character strings from a file
strings [-dopx] [-*length***] [***file ...* **]**

strings looks for ASCII strings in a binary file. A "string" is defined as any sequence of four or more printable characters. **strings** is useful for identifying unknown object files, or for looking at the messages printed by commands. You can also use it as a filter if *file* is not specified.

strings recognizes the following command-line options:

-d Precede each string by its offset in the file in decimal.

-o Precede each string by its offset in the file in octal.

-p Strip the parity bits of all characters in the string prior to comparison.

-x Precede each string by its offset in the file in hexadecimal.

Finally, the option *-length* forces **strings** to use *length* as the minimum length for a printable string.

See Also

commands, isprint, od

strip — Command

Strip tables from executable file
strip *file [...]*

strip removes the symbol table, relocation information, and debug tables from a file. It makes the executable file noticeably smaller.

See Also

cc, commands, ld, nm, size

strlen() — String Function (libc)

Measure a string
#include <string.h>
int strlen(*string***)**
char **string***;**

strlen() measures *string*, and returns its length in bytes, *not* including the null terminator. This is useful in determining how much storage to allocate for a string.

Example

For an example of how to use this function, see the entry for **string**.

See Also

libc, string.h
ANSI Standard, §7.11.6.3
POSIX Standard, §8.1

strncat() — String Function (libc)

Append one string onto another
#include <string.h>
char *strncat(*string1***,** *string2***,** *n***)**
char **string1***,** **string2***; unsigned** *n***;**

strncat() copies up to *n* characters from *string2* onto the end of *string1*. It stops when *n* characters have been copied or it encounters a null character in *string2*, whichever occurs first, and returns the modified *string1*.

Example

For an example of this function, see the entry for **strncpy**.

See Also

libc, strcat(), string.h
ANSI Standard, §7.11.3.2
POSIX Standard, §8.1

Notes

string1 should point to enough space to hold itself and *n* characters of *string2*. If it does not, a portion of the program or operating system may be overwritten.

strncmp() — String Function (libc)

Compare two strings
#include <string.h>
int strncmp(*string1***,** *string2***,** *n***)**
char **string1***,** **string2***; unsigned** *n***;**

strncmp() compares lexicographically the first *n* bytes of *string1* with *string2*. Comparison ends when *n* bytes have been compared, or a null character encountered, whichever occurs first. **strncmp()** returns zero if the strings are identical, returns a number less than zero if *string1* occurs earlier alphabetically than *string2*, and returns a number greater than zero if it occurs later. This routine is compatible with the ordering routine needed by **qsort()**.

Example

For an example of this function, see the entry for **strncpy()**.

See Also

libc, strcmp(), string.h
ANSI Standard, §7.11.4.4
POSIX Standard, §8.1

strncpy() — String Function (libc)

Copy one string into another
#include <string.h>
char *strncpy(*string1***,** *string2***,** *n***)**
char **string1***,** **string2***; unsigned** *n***;**

strncpy() copies up to *n* bytes of *string2* into *string1*, and returns *string1*. Copying ends when **strncpy()** has copied *n* bytes or has encountered a null character, whichever comes first. If *string2* is less than *n* characters in length, **strncpy()** pads *string1* to length *n* with one or more null bytes.

Example

This example, called **swap.c**, reads a file of names, and changes them from the format

```
first_name  [middle_initial] last_name
```

to the format

```
last_name, first_name [middle_initial]
```

It demonstrates **strncpy()**, **strncat()**, **strncmp()**, and **index()**.

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#define NNAMES 512
#define MAXLEN 60

char *array[NNAMES];
char gname[MAXLEN], lname[MAXLEN];

main(argc, argv)
int argc; char *argv[];
{
	FILE *fp;
	register int count, num;
	register char  *name, string[60], *cptr, *eptr;
	unsigned glength, length;

	if (--argc != 1) {
		fprintf (stderr, "Usage: swap filename\n");
		exit(EXIT_FAILURE);
	}

	if ((fp = fopen(argv[1], "r")) == NULL)
		printf("Cannot open %s\n", argv[1]);
	count = 0;
```

```
	while (fgets(string, 60, fp) != NULL) {
		if ((cptr = index(string, '.')) != NULL) {
			cptr++;
			cptr++;
		} else if ((cptr = index(string,' ')) != NULL)
			cptr++;

		strcpy(lname, cptr);
		eptr = index(lname, '\n');
		*eptr = ',';

		strcat(lname," ");
		glength = (unsigned)(strlen(string) - strlen(cptr));
		strncpy(gname, string, glength);

		name = strncat(lname, gname, glength);
		length = (unsigned)strlen(name);
		array[count] = malloc(length + 1);

		strcpy(array[count],name);
		count++;
	}

	for (num = 0; num < count; num++)
		printf("%s\n", array[num]);
	exit(EXIT_SUCCESS);
}
```

See Also

libc, strcpy(), string.h
ANSI Standard, §7.11.2.4
POSIX Standard, §8.1

Notes

string1 must point to enough space to *n* bytes; otherwise, a portion of the program or operating system may be overwritten.

stropts.h — Header File

User-level STREAMS routines
#include <stropts.h>

The header file **stropts.h** gives user-level information about STREAMS system calls and calls to **ioctl()**.

See Also

header files, STREAMS

strpbrk() — String Function (libc)

Find first occurrence of a character from another string
#include <string.h>
char *strpbrk(*string1***,** *string2***)**
char ****string1*, ****string2***;**

strpbrk() returns a pointer to the first character in *string1* that matches any character in *string2*. It returns NULL if no character in *string1* matches a character in *string2*.

The set of characters that *string2* points to is sometimes called the "break string". For example,

```
char *string = "To be, or not to be: that is the question.";
char *brkset = ",;";
strpbrk(string, brkset);
```

returns the value of the pointer **string** plus five. This points to the comma, which is the first character in the area pointed to by **string** that matches any character in the string pointed to by **brkset**.

See Also

libc, string.h
ANSI Standard, §7.11.5.4

POSIX Standard, §8.1

Notes

strpbrk() resembles the function **strtok()** in functionality, but unlike **strtok()**, it preserves the contents of the strings being compared. It also resembles the function **strchr()**, but lets you search for any one of a group of characters, rather than for one character alone.

strrchr() — String Function (libc)

Search for rightmost occurrence of a character in a string
#include <string.h>
char *strrchr(*string***,** *character***)**
char ****string*; int** *character***;**

strrchr() looks for the last, or rightmost, occurrence of *character* within *string*. *character* is declared to be an **int**, but is handled within the function as a **char**. Another way to describe this function is to say that it performs a reverse search for a character in a string. It is equivalent to the COHERENT function **rindex()**.

strrchr() returns a pointer to the rightmost occurrence of *character*, or NULL if *character* could not be found within *string*.

See Also

libc, rindex(), string.h
ANSI Standard, §7.11.5.5
POSIX Standard, §8.1

strspn() — String Function (libc)

Return length a string includes characters in another
#include <string.h>
unsigned int strspn(*string1***,** *string2***)**
char ****string1*; char ****string2***;**

strspn() returns the length for which *string1* initially consists only of characters that are found in *string2*. For example,

```
char *s1 = "hello, world";
char *s2 = "kernighan & ritchie";
strcspn(s1, s2);
```

returns two, which is the length for which the first string initially consists of characters found in the second.

See Also

libc, string.h
ANSI Standard, §7.11.5.6
POSIX Standard, §8.1

strstr() — String Function (libc)

Find one string within another
#include <string.h>
char *strstr(*string1***,** *string2***)**
char ****string1*, ****string2***;**

The string function **strstr()** looks for *string2* within *string1*. The terminating NUL is not considered part of *string2*.

strstr() returns a pointer to where *string2* begins within *string1*, or NULL if *string2* does not occur within *string1*.

For example,

```
char *string1 = "Hello, world";
char *string2 = "world";
strstr(string1, string2);
```

returns **string1** plus seven, which points to the beginning of **world** within **Hello, world**. On the other hand,

```
char *string1 = "Hello, world";
char *string2 = "worlds";
strstr(string1, string2);
```

returns NULL because **worlds** does not occur within **Hello, world**.

See Also

libc, string.h
ANSI Standard, §7.11.5.7
POSIX Standard, §8.1

Notes

Neither *string1* nor *string2* can be more than 2,147,483,647 characters long.

strtod() — General Function (libc)

Convert string to floating-point number
#include <stdlib.h>
double strtod(*string, tailptr***)**
char **string***; char** ***tailptr***;**

strtod() converts the number given in *string* to a double-precision floating-point number and returns its value. It is a more general version of the function **atof()**. **strtod()** also stores a pointer to the first character following the number through *tailptr*, provided *tailptr* is not NULL.

strtod() parses the input *string* into three portions: beginning, subject sequence, and tail.

The *beginning* consists of zero or more white-space characters that begin the string.

The *subject sequence* is the portion of the input *string* that **strtod()** converts into a floating-point number. It consists of an optional sign character, a nonempty sequence of decimal digits optionally including a decimal-point character, and an optional exponent. If present, the exponent consists of either ‘e’ or ‘E’ followed by an optional sign and a nonempty sequence of decimal digits. **strtod()** reads characters until it encounters either a second decimal-point character or exponent marker, or any other non-numeral.

The *tail* continues from the end of the subject sequence to the null character that ends the string.

strtod() ignores the beginning portion of the string. It converts the subject sequence to a double-precision number. Finally, it sets the pointer pointed to by *tailptr* to the address of the first character of the string’s tail.

strtod() returns the **double** generated from the subject sequence. If no subject sequence could be recognized, it returns zero and stores the initial value of *string* through *tailptr*. If the number represented by the subject sequence is too large or too small to fit into a **double**, then **strtod()** sets the global constant **errno** to **ERANGE** and returns **HUGE_VAL** or zero, respectively. If, however, the number given in the subject sequence has more digits to the right of the decimal point than can be encoded within an IEEE **double** (which has a fraction of 53 bits), **strtod** trims the excess digits before it converts the string.

Example

The following gives an example for **strtod()**.

```
#include <stdlib.h>

main()
{
      static char st[] = " 123.4 567.8";
      char *head, *tail;

      for (head = st;; head = tail) {
            double amt = strtod(head, &tail);

             /* No token found is end of string */
            if (head == tail)
                  break;
            printf("%f\n", amt);
      }
      exit(EXIT_SUCCESS);
}
```

See Also

atof(), double, errno, libc, limits.h, stdlib.h, strtol(), strtoul()
ANSI Standard, §7.10.1.4

Notes

strtod() ignores initial white space in the string pointed to by *string*; white space is defined as being all characters so recognized by the function **isspace()**.

strtok() — String Function (libc)

Break a string into tokens
#include <string.h>
char *strtok(*string1***,** *string2***)**
char ****string1*, ****string2***;**

strtok() divides a string into a set of tokens. *string1* points to the string to be divided, and *string2* points to the character or characters that delimit the tokens.

strtok() divides a string into tokens by being called repeatedly.

On the first call to **strtok()**, *string1* should point to the string being divided. **strtok()** searches for a character that is *not* included within *string2*. If it finds one, then **strtok()** regards it as the beginning of the first token within the string. If one cannot be found, then **strtok()** returns NULL to signal that the string could not be divided into tokens. When it finds the beginning of the first token, **strtok()** then looks for a character that *is* included within *string2*. When it finds one, **strtok()** replaces it with NUL to mark the end of the first token, stores a pointer to the remainder of *string1* within a static buffer, and returns the address of the beginning of the first token.

On subsequent calls to **strtok()**, pass it NULL instead of *string1*. **strtok()** then looks for subsequent tokens using the address that it saved from the first time you called it.

Note that with each call to **strtok()**, *string2* may point to a different delimiter or set of delimiters.

Example

The following example breaks **command_string** into individual tokens and puts pointers to the tokens into the array **tokenlist[]**. It then returns the number of tokens created. No more than **maxtoken** tokens will be created. **command_string** is modified to place '\0' over token separators. The token list points into **command_string**. Tokens are separated by spaces, tabs, commas, semicolons, and newlines.

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

tokenize(command_string, tokenlist, maxtoken)
char *command_string, *tokenlist[]; size_t maxtoken;
{
	static char tokensep[]="\t\n ,;";
	int tokencount;
	char *thistoken;

	if(command_string == NULL || !maxtoken)
		return 0;

	thistoken = strtok(command_string, tokensep);

	for(tokencount = 0; tokencount < maxtoken &&
			thistoken != NULL;) {
		tokenlist[tokencount++] = thistoken;
		thistoken = strtok(NULL, tokensep);
	}

	tokenlist[tokencount] = NULL;
	return tokencount;
}

#define MAXTOKEN 100
char *tokens[MAXTOKEN];
char buf[80];

main()
{
	for(;;) {
		int i, j;
```

```
            printf("Enter string ");
            fflush(stdout);
            if(gets(buf) == NULL)
                  exit(EXIT_SUCCESS);

            i = tokenize(buf, tokens, MAXTOKEN);
            for (j = 0; j < i; j++)
                  printf("%s\n", tokens[j]);
      }
}
```

See Also

libc, string.h
ANSI Standard, §7.11.5.8
POSIX Standard, §8.1

strtol() — General Function (libc)

Convert string to long integer
#include <stdlib.h>
long strtol(*string, tailptr, base***)**
char **string***; char** ***tailptr***; int** *base***;**

strtol() converts the number given in *string* to a **long** and returns its value; it is a more general version of the function **atol()**. **strtol()** also stores a pointer to the first character following the number through *tailptr*, provided *tailptr* does not equal NULL.

base gives the base of the number being read, either zero or a value from two to 36. If the given *base* is zero, **strtol()** determines an implicit base for the number: hexadecimal if the number starts with **0x** or **0X**, octal if the number starts with **0**, or decimal otherwise. Alternatively, you can specify a *base* between 2 and 36.

strtol() parses *string* into three portions: beginning, subject sequence, and tail.

The *beginning* consists of zero or more white-space characters that begin the string.

The *subject sequence* is the portion of the string that **strtol()** converts into a **long**. It consists of an optional sign character, an optional prefix **0x** or **0X** if the *base* is 16, and a nonempty sequence of *digits* in the specified base. For example, if the *base* is 16, then **strtol()** recognizes numeric characters '0' to '9' and alphabetic characters 'A' through 'F' and 'a' to 'f' as digits. It continues to scan until it encounters a nondigit.

The *tail* continues from the end of the subject sequence to the null character that ends the string.

strtol() ignores the beginning portion of the string. It converts the subject sequence to a **long**. Finally, if *tailptr* is not NULL, it sets the pointer pointed to by *tailptr* to the address of the first character of the string's tail.

strtol() returns a **long** representing the value of the subject sequence. If the input *string* does not specify a valid number, it returns zero and stores the initial value of *string* through *tailptr*. If the number it builds is too large or too small to fit into a **long**, it sets the global variable **errno** to the value of the macro **ERANGE** and returns **LONG_MAX** or **LONG_MIN**, respectively.

See Also

libc
ANSI Standard, §7.10.1.5

Notes

strtol() ignores initial white space in the input *string*. White space is defined as being all characters so recognized by the function **isspace()**.

strtoul() — General Function (libc)

Convert string to unsigned long integer
#include <stdlib.h>
unsigned long strtoul(*string, tailptr, base***)**
char **string***; char** ***tailptr***; int** *base***;**

strtoul() converts the number given in *string* to a **unsigned long** and returns its value. It is the **unsigned long** counterpart of **strtol()** and a more general version of the function **atol()**. **strtoul()** also stores a pointer to the first character following the number through *tailptr*, provided *tailptr* does not equal NULL.

base gives the base of the number being read, either zero or a value from two to 36. If the given *base* is zero, **strtoul()** determines an implicit base for the number: hexadecimal if the number starts with **0x** or **0X**, octal if the number starts with **0**, or decimal otherwise. Alternatively, the user can specify an explicit *base* between two and 36.

strtoul() parses the *string* into three portions: beginning, subject sequence, and tail.

The *beginning* consists of zero or more white-space characters that begin the string.

The *subject sequence* is the portion of the string that **strtoul()** converts into an **unsigned long**. It consists of an optional sign character, an optional prefix **0x** or **0X** if the *base* is 16, and a nonempty sequence of *digits* in the specified base. For example, if the *base* is 16, then **strtoul()** recognizes numeric characters '0' to '9' and alphabetic characters 'A' through 'F' and 'a' to 'f' as digits. It continues to scan until it encounters a nondigit.

The *tail* continues from the end of the subject sequence to the null character that ends the string.

strtoul() ignores the beginning portion of the string. It converts the subject sequence to an **unsigned long**. Finally, if *tailptr* does not equal NULL , it sets the pointer pointed to by *tailptr* to the address of the first character of the string's tail.

strtoul() returns an **unsigned long** representing the value of the subject sequence. If the input *string* does not specify a valid number, it returns zero and stores the initial value of *string* through *tailptr*. If the number it builds is too large to fit into an **unsigned long**, it sets the global variable **errno** to the value of the macro **ERANGE** and returns **ULONG_MAX**.

Example

This example uses **strtoul()** as a hash function for table lookup. It demonstrates both hashing and linked lists. Hash-table lookup is the most efficient when used to look up entries in large tables; this is an example only.

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

/*
 * For fastest results, use a prime about 15% bigger
 * than the table. If short of space, use a smaller prime.
 */
#define HASHP 11
struct symbol {
      struct symbol *next;
      char *name;
      char *descr;
} *hasht[HASHP], codes[] = {

      NULL,        "a286",            "frogs togs",
      NULL,        "xy7800",          "doughnut holes",
      NULL,        "z678abc",         "used bits",
      NULL,        "xj781",           "black-hole varnish",
      NULL,        "h778a",           "table hash",
      NULL,        "q167",            "log(-5.2)",
      NULL,        "18888",           "quid pro quo",
      NULL,        NULL,              NULL    /* end marker */
};

void
buildTable()
{
      long h;
      register struct symbol *sym, **symp;

      for(symp = hasht; symp != (hasht + HASHP); symp++)
            *symp = NULL;
```

```
	for(sym = codes; sym->descr != NULL; sym++) {
		/*
		 * hash by converting to base 36. There are
		 * many ways to hash, but use all the data.
		 */

		h = strtoul(sym->name, NULL, 36) % HASHP;
		sym->next = hasht[h];
		hasht[h] = sym;
	}
}

struct symbol *
lookup(s)
char *s;
{
	long h;
	register struct symbol *sym;

	h = strtoul(s, NULL, 36) % HASHP;
	for(sym = hasht[h]; sym != NULL; sym = sym->next)
		if(!strcmp(sym->name, s))
			return(sym);
	return(NULL);
}

main()
{
	char buf[80];
	struct symbol *sym;

	buildTable();
	for(;;) {
		printf("Enter name ");
		fflush(stdout);

		if(gets(buf) == NULL)
			exit(EXIT_SUCCESS);

		if((sym = lookup(buf)) == NULL)
			printf("%s not found\n", buf);

		else
			printf("%s is %s\n", buf, sym->descr);
	}
}
```

See Also

errno, libc, limits.h, stdlib.h, strtol()
ANSI Standard, §7.10.1.6

Notes

strtoul() ignores initial white space in the input *string*. White space is defined as being all characters so recognized by the function **isspace()**.

struct — C Keyword

Data type

struct is a C keyword that introduces a structure. The following is an example of how **struct** can be used in the description of a name and address file:

```
struct address {
      char firstname[10];
      char lastname[15];
      char street[25];
      char city[10];
      char state[2];
      char zip[5];
      int  salescode;
};
```

The C Programming Language, second edition prohibits the assignment of structures, the passing of structures to functions, and the returning of structures by functions. COHERENT, however, lifts these restrictions. It allows one structure to be assigned to another, provided the two structures are of the same type. It also allows structures to be passed by and returned by functions. These features are supported by most compilers, but users should be aware that their use can cause problems in porting code to some compilers.

See Also

array, C keywords, field, initialization, structure
ANSI Standard, §3.1.2.5, §3.5.2.1

structure — Definition

A *structure* is a set of variables that has been given a name and can be manipulated as a single entity. The variables may be of different data types. Structures are a convenient way to deal with data elements that belong together, such as names and addresses, employee descriptions, or sales and inventory information.

See Also

Programming COHERENT, struct

structure assignment — Definition

The C Programming Language, second edition forbids structure assignment, the passing of structures to functions, and returning structures from functions (as opposed to the passing or returning of *pointers* to structures). The COHERENT C compiler lifts these restrictions.

Some C compilers transform structure arguments and structure returns into structure pointers. Note that the use of structure assignment, structure arguments, or structure returns may create problems when porting the code to another C compiler.

See Also

portability, Programming COHERENT, struct, structure

Notes

Because this feature deviates from the description of the C language found in the first edition of *The C Programming Language* compiling with the **-VSBOOK** option will flag all points where it occurs in your program.

strxfrm() — String Function (libc)

Transform a string using locale information
#include <string.h>
unsigned int strxfrm(*string1***,** *string2***,** *n***)**
char **string1***,** **string2***; unsigned int** *n***;**

strxfrm() transforms *string2* using information concerning the program's locale, as set by the function **setlocale()**. It then writes up to *n* bytes of the transformed result into the area to which *string1* points. It returns the length of the transformed string, not including the terminating null character. The transformation incorporates locale-specific material into *string1*.

If *n* is set to zero, **strxfrm()** returns the length of the transformed string.

If two strings return a given result when compared by **strcoll()** before transformation, they will return the same result when compared by **strcmp()** after transformation.

Example

The following simple program demonstrates **strxfrm()**:

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

main()
{
	char string1[20], string2[20];

	strcpy (string1, "This is string 1");
	strcpy (string2, "This is string 2");

	printf ("String 1 before transformation: %s\n", string1);
	printf ("String 2 before transformation: %s\n", string2);

	strxfrm (string1, string2, 18);

	printf ("String 1 after transformation: %s\n", string1);
	printf ("String 2 after transformation: %s\n", string2);
}
```

See Also

libc, string.h
ANSI Standard, §7.11.4.5

Notes

If **strxfrm()** returns a value equal to or greater than *n*, the contents of the area to which *string1* points are indeterminate.

stty — Command

Set/print terminal modes
stty
stty -a
stty -g
stty *x*:*x*: **...** :*x*
stty *arglist ...*

The command **stty** lets you change or display the settings of the standard input device. The device is usually a terminal, although tapes, disks and other special files may be applicable.

Default Settings

The following describes how COHERENT sets up a terminal device by default. This normal processing is often called "cooked" mode. Note that on some machines, the default characters differ from those given below.

The *erase* and *kill* characters (normally **<crtl-H>** and **<ctrl-U>**) erase, respectively, one typed character and an entire line of typing.

The *stop-output* and *start-output* characters (normally **<ctrl-S>** and **<ctrl-Q>**) respectively stop and restart output.

The *interrupt* character (normally **<ctrl-C>**) sends the signal **SIGINT**, which usually terminates program execution.

The *quit* character (normally **<ctrl-\>**) sends the signal **SIGQUIT**, which usually terminates program execution with a core dump.

The *end of file* character (normally **<ctrl-D>**) generates an end-of-file signal from the terminal.

You can change the setting of each special character by invoking **stty** with the appropriate option.

Options

When called without any arguments, **stty** gives a brief listing of settings for the standard-input device.

stty can read the settings of devices other than the standard-input device by redirecting that device to it. For example, the command

```
stty < /dev/com1l
```

prints a brief summary of the settings for serial device **com1l**.

stty's command-line arguments can take a number, as indicated below by *n*; or they can take a character, as indicated below by *c*. Argument *c* can be one of the following:

- A single character.
- A caret '^' followed by a single character (to indicate a control character, e.g., **^X** for **<ctrl-X>**).
- An **^?**, which denotes the **<del>** character.
- An '0x' followed by two hexadecimal digits.
- An **^-**, which indicates that that option is not used.

stty recognizes the following command-line arguments:

-a Give a complete listing of settings for the standard-input device.
-g Give a complete list of settings for the standard-input device, but in hexadecimal. This is a dump of the **termio** structure in effect at the moment. For more information on the **termio** structure, see the Lexicon entry for **termio**.
x:x:...:x Establish new settings for the standard-input device. The settings are hexadecimal values that are separated by colons. This form can be combined with **-g** option to copy **stty** settings from one device to another. For example, to set device **com2l** so that it mimics device **com1l**, use the following command:

```
stty `stty -g < /dev/com1l` < /dev/com2l
```

0 Hang up the telephone.
50 Set line speed to 50 bps.
75 Set line speed to 75 bps.
110 Set line speed to 110 bps.
134 Set line speed to 110 bps.
150 Set line speed to 150 bps.
200 Set line speed to 200 bps.
300 Set line speed to 300 bps.
600 Set line speed to 600 bps.
1200 Set line speed to 1200 bps.
1800 Set line speed to 1800 bps.
2400 Set line speed to 2400 bps.
4800 Set line speed to 4800 bps.
9600 Set line speed to 9600 bps.
19200 Set line speed to 19200 bps.
38400 Set line speed to 38400 bps.
brkint Send interrupt on break.
-brkint Do not send interrupt on break.
bs0 No delay on backspace.
bs1 Delay briefly on backspace.
clocal Turn on modem control.
-clocal Turn off modem control.
cooked Set the device into cooked mode. This is a composite of options **parenb**, **-parodd**, **cs7**, **brkint**, **ignpar**, **istrip**, **icrnl**, **ixon**, **opost**, **onlcr**, **isig**, and **icanon**.
cr0 No delay on carriage returns.
cr1 Carriage-return delay depends upon column position.
cr2 Delay approximately 0.10 seconds on carriage return.
cr3 Delay appoximately 0.15 seconds on carriage return.
cread Enable the receiver.
-cread Disable the receiver.
cs5 Character size is five bits.
cs6 Character size is six bits.
cs7 Character size is seven bits.
cs8 Character size is eight bits.
cstopb Use two stop bits per character.
-cstopb Use one stop bit per character.
echo Echo every character.
-echo Do not echo characters.
echoe Echo the erase character as backspace-space-backspace.
-echoe Do not echo the erase character as backspace-space-backspace.
echok Echo newline after the kill character.
-echok Do not echo newline after the kill character.

echonl Echo newline.
-echonl Do not echo newline.
ek Set the kill and erase characters to printable characters. A composite of **erase '#'** and **kill '@'**.
eof *c* Set the end-of-file character to *c*.
eol *c* Set the end-of-line character to *c*.
erase *c* Set the erase character to *c*.
evenp Set the port to even parity. This is a composite of the options **parenb**, **-parodd**, and **cs7**.
-evenp Turn off even parity — in effect, turn off parity altogether. This is a composite of the options **-parenb** and **cs8**.
ff0 No delay on formfeeds.
ff1 Delay approximately two seconds on formfeeds.
hup Hang up the telephone on logging out.
-hup Do not hang up the telephone on logging out.
hupcl Same as **hup**.
-hupcl Same as **-hup**.
icanon Enable canonical input.
-icanon Disable canonical input.
icrnl Map carriage-return to newline on input.
-icrnl Do not map carriage-return to newline on input.
ignbrk Ignore break on input.
-ignbrk Do not ignore break on input.
igncr Ignore carriage return on input.
-igncr Do not ignore carriage return on input.
ignpar Ignore parity errors on input.
-ignpar Do not ignore parity errors on input.
inlcr Map newline to carriage return on input.
-inlcr Do not map newline to carriage return on input.
inpck Enable parity checking on input.
-inpck Do not enable parity checking on input.
intr *c* Set the interrupt character to *c*.
isig Check input against interrupt and quit characters.
-isig Do not check input against interrupt and quit characters.
iuclc Map input's upper-case characters to lower case.
-iuclc Do not map input's upper-case characters to lower case.
istrip Strip input to seven bits.
-istrip Do not strip input to seven bits.
ixany Allow any on input character to restart output.
-ixany Do not allow any input character to restart output.
ixoff Request that system send start or stop characters when the input queue is, respectively, nearly full or nearly empty.
-ixoff Do not request that system send start or stop characters to manage input queue.
ixon Use start/stop characters to control output queue.
-ixon Do not use start/stop characters to control output queue
kill *c* Set the kill character to *c*.
lcase Map upper-case characters to lower case. A composite of options **xcase**, **iuclc**, and **olcuc**.
-lcase Turn off mapping of upper-case character to lower case. A composite of options **-xcase**, **-iuclc**, and **-olcuc**.
LCASE A synonym for **lcase**.
-LCASE A synonym for **-lcase**.
min *n* Set the constant **VMIN** to decimal value *n*. For more about **VMIN**, see the Lexicon entry for **termio**.
nl A composite of options **-icrnl** and **-onlcr**.
-nl A composite of options **icrnl**, **-inlcr**, **-igncr**, **onlcr**, **-ocrnl**, and **-onlret**.
nl0 No delay on newline.
nl1 Delay approximately 0.10 seconds on newline.
noflsh Flush buffer on interrupt or quit.
-noflsh Do not flush buffer on interrupt or quit.
ocrnl In output, map carriage return to newline.
-ocrnl In output, do not map carriage return to newline.
oddp Set device to odd parity. This option is a composite of the options **parenb**, **parodd**, and **cs7**.
-oddp Turn off odd parity — in effect, turn off parity altogether. This is a composite of the options **-parenb** and **cs8**.

ofdel Use delete characters as fill characters.
-ofdel Do not use delete characters as fill characters.
ofill Use fill characters for delays.
-ofill Do not use fill characters for delays.
olcuc Map lower-case characters to upper case on output.
-olcuc Do not map lower-case characters to upper case on output.
onlcr Map newline to carriage return/newline on output.
-onlcr Do not map newline to carriage return/newline on output.
onlret A newline character executes a carriage return.
-onlret A newline character does not execute a carriage return.
onocr Do not output carriage returns at column 0.
-onocr Output carriage returns at column 0.
opost Post-process output.
-opost Do not post-process output.
parenb Enable parity generation and detection.
-parenb Disable parity generation and detection.
parity Synonym for option **evenp**.
-parity Synonym for option **-evenp**.
parmrk Mark parity errors.
-parmrk Do not mark parity errors.
parodd Odd parity.
-parodd Turn off odd parity; i.e., use even parity.
quit *c* Set the quit character to *c*.
raw Set the device into raw mode. This is a composite of the options **-parenb**, **-parodd**, **-hupcl**, **cs8**, **-opost**, **-olcuc**, **-ocrnl**, **-onocr**, **-onlret**, **-ofill**, **-ofdel**, **nl0**, **cr0**, **tab0**, **bs0**, **vt0**, and **ff0**. This turns off most character processing, including all input processing (see **c_iflag** fields in **<termio.h>**), canonical input buffering (**-icanon**), and output processing (**-opost**). It does not turn off echo.
-raw Turn off raw mode — in effect, restore the device to cooked mode. Same as **cooked**.
sane Restore the device to “sanity” — for example, after an editor or communications program has died unexpectedly. This is a composite of options **icrnl**, **opost**, **onlcr**, **isig**, **icanon**, **-xcase**, **echo**, **echoe**, **echok**, and **erase ^h**.
tab0 No delay for horizontal-tab character.
tab1 Delay for horizontal-tab character depends on column position.
tab2 Delay approximately 0.10 seconds on horizontal tab.
tab3 Expand horizontal-tab characters into spaces.
tabs A synonym for **tab0**.
-tabs A synonym for **tab3**.
time *n* Set the constant **VTIME** to decimal value *n*. For more about **VTIME**, see the Lexicon entry for **termio**.
vt0 No delay on vertical-tab characters.
vt1 Delay approximately two seconds on vertical-tab characters.
xcase Canonical presentation of upper-case and lower-case characters.
-xcase Do not process upper-case and lower-case characters.

See Also

ASCII, commands, getty, init, ioctl(), signal()

Notes

Executing **stty** with input redirected from another device does not have an effect unless the device being read is open. The last close of any terminal device resets all **termio** values to the system defaults. Thus, to change the settings of a device, you must first open the device.

For example,

```
enable com1l
```

or

```
sleep 32000 > /dev/com1l &
```

might precede:

```
stty evenp < /dev/com1l
```

Note, too, that **stty** does not check its arguments for consistency.

stty provides complete access to the System-V-style **termio** structure. Note, however, that the settings of **termio** are processed by the kernel's in-line discipline and device-driver modules. Under COHERENT, none of these modules pays attention to delay settings. Therefore, setting delays with **stty** does not, at present, affect the behavior of the terminal device.

stty() — System Call (libc)

Set terminal modes
#include <sgtty.h>
int stty(*fd, sgp***)**
int *fd***;**
struct sgttyb **sgp***;**

The COHERENT system call **stty()** sets a terminal's attributes. See the Lexicon article for **stty** for information on terminal attributes and their legal values.

Example

This example demonstrates both **stty()** and **gtty()**. It sets terminal input to read one character at a time (that is, it reads the terminal in "raw" form). When you type 'q', it restores the terminal to its previous settings, and exits. For an additional example, see the **pipe** Lexicon article.

```
#include <sgtty.h>

main()
{
	struct sgttyb os, ns;
	char buff;

	printf("Waiting for q\n");
	gtty(1, &os);		/* save old state */
	ns = os;		/* get base of new state */
	ns.sg_flags |= RAW;	/* prevent <ctl-c> from working */
	ns.sg_flags &= ~(ECHO|CRMOD);/* no echo for now... */
	stty(1, &ns);		/* set mode */

	do {
		buff = getchar();	/* wait for the keyboard */
	} while(buff != 'q');

	stty(1, &os);		/* reset mode */
}
```

Files

<sgtty.h> — Header file

See Also

exec, gtty(), ioctl(), libc, open(), read(), sgtty.h, stty, write()

Notes

Please note that if you use **stty()** to change the baud rate on a port, you must first invoke **sleep()**. If you do not, the port reverts back to its default settings.

stune — System Administration

Set values of tunable kernel variables
/etc/conf/stune

File **stune** names each tunable variable within the kernel, and gives the value to which it is actually set. Command **idmkcoh** reads this file when it builds a new kernel, and uses its contents to patch the kernel appropriately.

Each entry within this file has two fields. The first field names the variable; the name must match that given in **stune**. The second field gives the value of the variable; this value must fall between the minimum and maximum values given in **stune**.

If a line begins with a pound sign '#', it is a comment and **idmkcoh** ignores it. If a tunable variable is not named in this file, **idmkcoh** uses the default value given in **stune**.

See Also

Administering COHERENT, device drivers, mdevice, mtune, sdevice

su — Command

Substitute user id, become superuser
su [*user* **[***command***]]**

Default *user* is **root**; default *command* is **sh**. **su** changes the real user id and the effective user id to that of the *user*. If *user* has a login password, **su** requests it. Then it executes the specified *command*.

If *command* is absent, **su** invokes an interactive sub-shell.

If *user* is absent, **su** assumes user name **root** (the superuser).

Files

/etc/passwd — Login names and passwords

See Also

commands, login, newgrp, sh, superuser

sum — Command

Print checksum of a file
sum [*file ...***]**

sum prints an unsigned integer checksum and a size in blocks (rounding up) for each *file* specified. If more than one *file* is specified, **sum** also prints the file name. If no *file* is specified, **sum** reads the standard input.

sum may be used to verify the integrity of data transferred across phone lines or stored on an unreliable medium.

See Also

cmp, commands

superuser — Definition

The *superuser* is the user who has system-wide permissions. He can execute any program, read any file, and write into any directory. Thus, superuser status is reserved to the system administrator, also called **root**, who needs this status to control the operation of the system.

No person should be able to become the superuser without knowing a password. Because the superuser in effect "owns" the system, the superuser password should be guarded most carefully.

See Also

root, su, Using COHERENT

swab() — General Function (libc)

Swap a pair of bytes
void swab(*src***,** *dest***,** *nb***) char ****src***, ****dest***; unsigned** *nb***;**

The ordering of bytes within a word differs from machine to machine. This may cause problems when moving binary data between machines. **swab()** interchanges each pair of bytes in the array *src* that is *n* bytes long, and places the result into the array *dest*. The length *nb* should be an even number, or the last byte will not be touched. *src* and *dest* may be the same place.

Example

This example prompts for an integer; it then prints the integer both as you entered it, and as it appears with its bytes swapped.

```
#include <stdio.h>

main()
{
	int word;
```

```
        printf("Enter an integer: \n");
        scanf("%d", &word);
        printf("The word is 0x%x\n", word);
        swab(&word, &word, 2);
        printf("The word with bytes swapped is 0x%x\n", word);
}
```

See Also

byte ordering, dd, canon.h, libc

switch — C Keyword

Test a variable against a table

switch is a C keyword that lets you perform a number of tests on a variable in a convenient manner. For example,

```
while(foo < 10)
        switch(foo) {
        case 1:
                dosomething();
                break;
        case 2:
                somethingelse();
        case 3:
                anotherthing();
                break;
        default:
                break;
        }
}
```

is equivalent to

```
while(foo < 10) {
        if(foo == 1) {
                dosomething();
                continue;
        } else if (foo == 2) {
                somethingelse();
                anotherthing();
                continue;
        } else if(foo == 3) {
         /* Note: compiler eliminates duplicate code */
                anotherthing();
                continue;
        } else
                break;
}
```

switch is always used with the **case** statement, and nearly always with the **default** statement.

See Also

break, C keywords, case, default, while
ANSI Standard, §6.6.4.2

sync — Command

Flush system buffers
sync

Most COHERENT commands manipulate files stored on a disk. To improve system performance, the COHERENT system often changes a copy of part of the disk in a buffer in memory, rather than repeatedly performing the time-consuming disk access required.

sync writes information from the memory buffers to the disk, updating the disk images of all mounted file systems which have been changed. In addition, it writes the date and time on the root file system.

sync should be executed before system shutdown to ensure the integrity of the file system.

See Also

commands

sync() — System Call (libc)

Flush system buffers
sync()

sync() is the COHERENT system call that copies the contents of all memory buffers to disk.

See Also

libc

sys — System Administration

Data base for UUCP connections
/usr/lib/uucp/sys

The file **/usr/lib/uucp/sys** describes how to communicate with a remote system. The UUCP daemon **uucico** uses the information in this file to telephone a remote system, log into a remote system, and control what it allows a remote system to do on your system.

Command **cu** also reads file **sys** for information on how it can call a remote system. However, the following descriptions concentrate on how **sys** is used by **uucico**.

Structure of the sys File

sys has the following structure:

> command *argument*
> ...
> **alternate**
> command *argument*
> ...
> **alternate**
> command *argument*
> ...
> **system** *remotesystem*
> command *argument*
> ...
> **alternate**
> command *argument*
> ...
> **system** *remotesystem*
> command *argument*
> ...
> **alternate**
> command *argument*
> ...

Blank lines in the file are ignored. The body of the file consists of a series of commands. Each command defines one or more values; each value, in turn, determines one aspect of how your system interacts with a remote system. A backslash at the end of a line lets an entry extend over more than one one line.

The commands from the top of **sys** to the first **system** command set global values — that is, the values used by default when dealing with every remote system. Note that **uucico** recognizes a number of global values that are not explicitly written in **sys**.

The command **system** names a remote system. The commands from one **system** command to the next (or the end of the file, whichever comes first) define the values that **uucico** uses when it communicates with that system. These system-specific values can override any of the global values.

The command **alternate** introduces a block of alternate values. The commands from one **alternate** command to the next **alternate** command (or to the next **system** command or to the end of the file, whichever comes first) set a block of alternate values. **uucico** uses a block of alternate values when the default values (and all preceding blocks of alternate values) fail for any reason. By defining blocks of alternate values, you can define multiple ways to interact with a remote system.

Order of Command Execution

As you can see the above display, both the global section and each system-specific section can contain blocks of alternate commands. The order in which **uucico** reads blocks of commands is important: each block can contain its own version of a given command, and **uucico** uses the value set by the command that it has read *last*.

The following describes the order in which **uucico** reads commands when it attempts to call site *remotesite*:

1. **uucico** reads its default global values (which are described below). It then reads the global-values section of **sys**, up to the first **alternate** command.

2. **uucico** reads the section *remotesite*, from its **system** command to the first **alternate** command.

3. **uucico** calls *remotesite*.

4. If the call to *remotesite* succeeds, then all is well. If it fails, however, then **uucico** reads the first block of alternate commands in the global section, then the first block of alternate commands in the section for *remotesite*.

Note that a block of alternate values can simply reproduce values set previously. This in effect forces **uucico** to try the same values once again.

5. **uucico** again calls *remotesite*.

6. If the call succeeds, then all is well. If it fails, **uucico** reads the second block of alternate values (should there be one) in the global-defaults section, then the second block of alternate values (again, should there be one) in the section for *remotesystem*. **uucico** makes its third attempt to call *remotesite*.

This process continues either until **uucico** succeeds in getting through to *remotesite*, or until it runs out of blocks of alternate values in both the global section and in the site-specific section.

As you can see, it can be difficult at times to tell just what values **uucico** is using at any given time. The command **uuchk** can help you untangle this skein of values. See its Lexicon entry for details.

Structural Commands

The following commands help control the manner in which **uucico** reads commands from **sys**:

system *remotesystem*
: Name the remote system. All commands up to the next **system** command refer to the system *remotesystem*.

alternate [*name***]**
: Introduce an alternate set of commands. The optional *name* lets you name this block of alternate commands; if **uucico** uses this block of alternate commands, it records *name* in the log file for *remotesystem*.

default-alternates true | false
: If its argument is **false**, do not use any blocks of alternate values from the global section. The default is **true**.

Chat Commands

The command **chat** defines a chat script. A *chat script* summarizes the conversation that your system has with the remote system as it attempts to log into that system.

chat has the following structure:

> **chat** *expect respond expect respond ... expect respond*

As you can see, a chat script consists of pairs of strings. Each pair contains an *expect* string, which gives what you expect the remote system to say to your system; and a *respond* string, which gives what your system sends in reply. When **uucico** runs out of *expect/respond* pairs, it assumes that it has succeeded in logging into *remotesystem*. If you want to send something to the remote system without waiting an *expect* string, then the *expect* string in a *expect/respond* pair should be simply a pair of quotation marks with nothing between.

Each string in the chat script is demarcated by white space. Therefore, you must use the escape sequence '**\s**' to indicate white space within a string. You can embed other escape sequences within the *respond* string; these are given below.

An *expect* string can contain several sub-strings separated by hyphens. The sub-strings themselves comprise pairs of *expect*/*respond* strings. If your system does not receive the first expect sub-string, it can send the first *respond* string (to prod the remote system), then await the second *expect* string; and so on, until your system either runs out of sub-strings or it receives an expect sub-string that it recognizes. You can, of course, repeat the same *expect*/*respond* pair more than once. Because sub-strings are separated by hyphens, you cannot use a literal hyphen in a string; you should indicate a literal hyphen by the escape sequence '\055' (ASCII for the hyphen character).

You can embed the following escape sequences in a *respond* string:

****	Literal backslash character
*****DDD*	Character with octal value *DDD*
\b	Backspace
\c	Suppress carriage return at end of send string
\d	Delay sending for one or two seconds
\E	Enable echo checking
\e	Disable echo checking
\K	Same as **BREAK**
\L	Your system's login name
\N	NUL
\n	Newline or line feed
\P	The password on the system being contacted
\p	Pause sending for a fraction of a second
\r	Carriage return
\s	Space
\t	Tab
\x*DDD*	Character with hexadecimal value *DDD*
\Z	Send name of the system being called
EOT	End-of-transmission character (**<ctrl-D>**)
BREAK	Break character

As in C, up to three octal digits may follow a backslash The escape sequence **\x** can be followed by an indefinite number of hexadecimal digits. To follow a hexadecimal escape sequence with a hexadecimal digit, interpose a send string of """.

uucico sends a carriage return at the end of each send string, unless the escape sequence **\c** appears in the string.

"Echo checking" means that after **uucico** writes each character, it waits for the remote system to echo it. You must turn on echo checking separately for each send string for which you want it.

The following gives an example chat script; the numbers simply mark the elements of the chat script for the discussion that follows, and are not part of the chat script:

```
         1    2     3                           4     5     6
     chat "" \r\c ogin:-BREAK-ogin:-BREAK-ogin: nuucp word: public
```

This script does the following:

1. Expect nothing from the modem (as indicated by the empty string "").

2. Send newline and carriage-return characters, as indicated by the escape sequence **\r\c**.

3. Expect the string **ogin:** (or a string that ends with **ogin:**). If this is not received within the defined pause period, send a break character (as indicated by the escape sequence **BREAK**), and wait again for **ogin:**. If the procedure times out again, send another break character and wait again. If the third attempt times out, quit.

4. Having received **ogin:** from **remotesystem**, send the string **nuucp**.

5. Wait for the string **word:**, that is, the tail of the prompt **Password:**.

6. When the password prompt is received, reply with the password **public**.

Some users may experience trouble when logging into a machine that is running SCO UNIX: it appears not to recognize carriage returns. The simplest work around is to embed the "delay" escape sequence **\d** in the send strings. For example, if you were using the above chat script to communicate with a SCO UNIX system, and the system was not responding to your transmission, you could modify it as follows:

```
chat "" \r\c ogin:-BREAK-ogin:-BREAK-ogin: \d\dnuucp\d\d\d\d word: \d\dpublic\d\d\d
```

This slows how your system responds to the SCO system; giving it enough time to "digest" your transmission appears to work around the problem. You can try adjusting the number of \d characters to get best performance.

The following commands help control the chat your system has with a *remotesystem*:

chat-fail *string*
Abort the chat if *string* is received. *string* cannot contain white-space characters; use escape sequences instead.

The description for *remotesystem* can contain multiple **chat-fail** commands. The default is to have none.

chat-program *program arguments*
Pass *arguments* (if any) to *program*, which is the path name of a program that you want **uucico** to execute before it executes your **chat** command. *program* can contain its own version of a chat script, but this is not required. If both a system's description contains both the commands **chat-program** and **chat**, **uucico** always executes the former first.

arguments can contain any of the following escape sequences:

\Y Port device name
\S Port speed
**** Literal backslash

uucico connects the standard input and standard output of *program* to the port in use, and connects the standard error of *program* to the UUCP log file. If *program* does not exit with a status of zero, **uucico** assumes that it has failed.

uucico runs *program* as user **uucp**, and the environment is that of the process that invoked **uucico**. Take care that by using *program*, you do not compromise your system's security.

chat-seven-bit true|false
If the argument is **true**, **uucico** strips all incoming characters to seven bits before it compares them with the expect string; otherwise, it uses all eight bits. The default is **true** because some UNIX systems generate parity bits during the login prompt that must be ignored while running a chat script.

chat-timeout *seconds*
Wait *seconds* for the remote system to respond to a send string. If send string times out, **uucico** sends the next send sub-string (if there is one), or fails. The default is timeout time is 60 seconds.

Aliases and Identifiers

The following commands let you manipulate how your system identifies itself to a *remotesystem*:

alias *systemalias*
Define *systemalias* to be an alias for *remotesystem*. The commands **uucp** and **uux** can use *systemalias*, as can *remotesystem* itself. This command is helpful should *remotesystem* change its name: it spares you the trouble of having to comb through your system to replace every occurrence of the old name. The default is to have no aliases.

myname *mysysname*
Tell your system to identify itself as *mysysname* instead of its true name (as kept in file **/etc/uucpname**) when it calls *remotesystem*.

If the description of *remotesystem* includes the command **called-login** without the argument **ANY**, your system will identify itself as *mysysname* when it is called by *remotesystem*.

call-login *loginname*
Tell **uucico** how to expand the escape sequence **\L**, which stands for the login name. With this command, you can use a default chat script with several different systems, expanding the login escape sequence (and password, as will be shown next) with the appropriate strings.

call-password *password*
Tell **uucico** how to expand the escape sequence **\P**, which stands for a password. As with the command **call-login**, described above, this command lets you use the same chat script with a number of different systems, by expanding the login and password escape sequences as needed.

Accepting a Call

The following commands affect how your system handles a call from another system:

called-login *login_identifer* **[***remotesystem ...* **]**
Recognize the remote system with the name *login_identifier* when it attempts to log into your system. If you set *login_identifier* to **ANY**, **uucico** will accept any login identifier. The optional *remotesystem* arguments name each remote system that is allowed to log in under that login identifier.

Some systems use this command to select a number of different alternate sections within **sys**; in effect, this allows **uucico** to jump to a given portion of **sys** based upon the identity of the system that is attempting to log in. In this case, the *remotesystem* arguments will not be used.

callback true|false
If **true**, this command tells **uucico** to hang up when the given remote system calls, and call it back. This is a security measure, to protect your system from being penetrated by remote systems. The default is **false**.

called-chat " " \r\d\r in:--in: nuucp word: public word: *serialnumber*
called-chat-fail *string*
called-chat-program *program arguments*
called-chat-seven-bit true|false
called-chat-timeout *seconds*
These commands control how a remote system logs into your system. They are analogous to the commands **chat**, **chat-fail**, **chat-program**, **chat-seven-bit**, and **chat-timeout**, and are structured just like them.

Note that **called-chat** the rest of these commands are invoked after protocol negotiation has been completed between **uucico** on your system and its counterpart on the remote system, but before data exchange has begun. How this chat sequence dovetails with the conversation that COHERENT has with the remote system when it logs into your system depends upon a number of factors, in particular whether COHERENT or **uucico** controls the port in question. It is customary to let COHERENT control logging in through serial ports, as these ports can be used by interactive users as well as by UUCP sessions, while **uucico** usually is allowed to control its well-known TCP port (540). However, **called-chat** can be used to perform special tasks on normal serial lines, such as put the modem into a special state that is required by a given remote site's hardware.

Time Strings and Time Commands

Many of the commands that you can use in *sys* commands use a special kind of string, the *time string*, to specify a range of time. The following describes the structure of a time string.

Each simple time string begins with a token that sets the day of the week. You can use any one of the following values:

Su	Sunday only
Mo	Monday only
Tu	Tuesday only
We	Wednesday only
Th	Thursday only
Fr	Friday only
Sa	Saturday only
Wk	Every week (Monday through Friday)
Any	Every day of the week

You can name more than one day of the week in a time string: just use commas to separate entries.

After the day of the week comes a range of hours and minutes The beginning and ending times are separated by a hyphen. Military time is used, i.e., hour 0 (midnight) through hour 23 (11 PM). **uucico** uses the local time on your system. The range of time can may cross midnight; for example **2300-0700** indicates 11 PM to 7 AM the following day.

If no time is given, any time applies. The word **Never** in place of the time string indicates that this remote system is never to be contacted. You should use this setting for systems that contact you but which you never contact.

You can specify more than one day/time combination in a time string; use commas to separate entries.

The following gives examples of time strings:

Wk2305-0855,Sa,Su2305-1655
Weekdays from 11:05 PM to 8:55 AM the following day; any time on Saturday; and Sunday from 11:05 PM to 4:55 PM the following day.

Wk0905-2255,Su1705-2255
Weekdays from 9:05 AM to 10:55 PM, and Sunday from 5:05 PM to 10:55 PM. The remote system cannot be called on Saturday.

The following commands control when *remotesystem* is contacted:

time *timestring* **[***retry***]**
Specify when your system can call *remotesystem*. *timestring* gives a time string; the section **Time Strings**, above, describes how to construct one. *retry*, if used, defines how long to wait before your system attempt to call *remotesystem* again. The default time for each *remotesystem* is **Never**.

The optional argument *retry* sets many minutes your system will wait before it attempts to recontact *remotesystem*, should a call made during *timestring* fail. If *retry* is not defined, **uucico** uses an exponentially increasing retry time: after each failure the next retry period is longer.

The description of *remotesystem* can contain multiple **time** commands. **uucico** will call *remotesystem* if the current time matches the time defined by any of them.

timegrade *grade timestring* **[retry]**
This command tells **uucico** to call *remotesystem* only if a file with a grade greater than or equal to *grade* is awaiting transfer to that system.

grade gives the grade of file to await. It is a single letter or digit, from '0' to '9', 'A' to 'Z', and 'a' to 'z', in this order from highest grade to lowest.

timestring gives the period of time to which this command applies. *retry* gives the length of time, in minutes, that your system must wait before it recontacts *remotesystem* should a call made during *timestring* fail.

The command **time** is equivalent to the command **timegrade** with a grade of 'z', which permits all jobs to be run. The command **uucico -S** overrides *grade*; the command **uucico -s** does not.

The *grade* applies only to calls made to *remotesystem*, not to calls that it makes to you.

The description of *remotesystem* can have multiple **timegrade** commands.

The command **call-timegrade**, described below, complements this command.

call-timegrade *grade timestring*
This command tells **uucico** to ask *remotesystem* to execute only the jobs with a grade of *grade* or higher, should it call *remotesystem* during the period of time defined in *timestring*. This commands complements the command **timegrade**: while **timegrade** limits what your system does with the remote system, **call-timegrade** attempts to limit what *remotesystem* does to your system.

grade gives the grade of the job to send. It is be a single letter or digit, from '0' to '9', 'A' to 'Z', and 'a' to 'z', in this order from highest grade to lowest. *timestring* gives the period of time to which this command applies. It is a time string, as defined above.

The description of a *remotesystem* can contain multiple **call-timegrade** commands.

Please note that not every implementation of UUCP will cooperate in setting grades to its jobs. If this command does not appear, or if no time string matches, the remote system can send whatever grade of work it chooses.

Retries and Waiting

The following commands define how often **uucico** will try to do something, and how long it will wait for a particular event to happen.

max-retries *retries*
Recontact a *remotesystem* no more than than *retries* times during any time time period. The default number of retries is 26.

success-wait *seconds*
Wait *seconds* before recontacting a *remotesystem* after a successful call. This limits the number of times a *remotesystem* will be contacted during a given time period. The default is zero.

Ports and Telephones

The following commands govern how **uucico** selects a port and telephones *remotesystem*.

address *ip_address* **|** *domain_name*
Name a remote system to contact a TCP/IP network. This command can name the remote system to contact either by its domain name (e.g., **lepanto.com**) or its IP address (e.g., 199.3.32.100). Note that if the port named by **port** command is not a TCP port, **uucico** ignores this command.

baud *speed*
speed *speed*
Set the speed (or "baud rate") at which to call *remotesystem*. This tells **uucico** to try every port defined in file **/usr/lib/uucp/port** until it finds an unlocked port that runs at *speed*.

If the description of *remotesystem* contains both the **baud** and **port**, **uucico** uses both when it selects a port.

If you wish to try multiple speeds when contacting a *remotesystem*, you must embed each **baud** command in its own set of alternate commands.

uucico does not use a default speed. The command

```
baud 0
```

tells **uucico** to use the "natural" speed of a port (whatever that is), and override and overrides any **baud** or **speed** commands that appear in the global defaults.

To place a call to a *remotesystem*, its description (or the global defaults) must name a port through which to dial out, either with **baud** or with the command **port** (described below).

port *portname*
Name or describe the port through which to contact *remotesystem*.

If used with only one argument, **uucico** assumes that that string names a port defined in the file **/usr/lib/uucp/port**. *portname* may point to more than one physical device; **uucico** tries each in turn until it finds one that is unlocked.

If used with more than one string, **uucico** assumes that the strings define a port, in the same way as done in the file **port**.

To place a call to a *remotesystem*, its description (or the global defaults) must name a port through which to dial out, either with **port** or with the command **baud** (described above).

phone *number*
Give the telephone number of *remotesystem*. An '=' character in the telephone number tells **uucico** to wait for a secondary dial tone. A '-' character tells **uucico** to pause for one second while dialing

The description of a *remotesystem* can have more than one **phone** command, one for each number at which you can call that *remotesystem*. If you want your system to telephone *remotesystem*, then its description must contain at least one **phone** command.

Protocols and Protocol Variables

The command

protocol *codes*

names the communication protocols to use with *remotesystem*. *code* must one or more lower-case letters, each of which names a protocol. If more than one protocol is named, **uucico** considers them in the order in which you give them.

uucico recognizes the following protocol codes:

t
e These protocols perform no checking at all. They are intended to be used over a communication path that has end-to-end reliability, e.g., TCP. **uucico** will consider them only when it is talking to a TCP port that is both reliable and eight-bit.

i This is a bidirectional protocol; that is, your system and *remotesystem* can both send and receive simultaneously. It requires an eight-bit connection. This protocol is preferred for a serial connection, as it offers the fastest transmission of data.

g This is the first, and the commonest UUCP protocol. Every implementation of UUCP supports this protocol; some support no other. It requires an eight-bit connection. For a detailed description of how this protocol works, see the article by Steven Baker, cited below.

G This is the System V Release 4 version of the **g** protocol.

a This mimics the Z-Modem protocol. It requires an eight-bit connection; but unlike the **g** and **i** protocols, it works even if certain control characters cannot be transmitted. (Code for this protocol was contributed by Doug Evans.)

j This is a variant of the **i** protocol, which can avoid certain control characters. The set of characters it avoids can be set by a parameter. It is useful over a eight-bit connection that will not transmit certain control characters.

f This protocol supports X.25 connections. It checksums each file as a whole, so any error causes the entire file to be retransmitted. It requires a reliable connection, but uses only seven-bit transmissions. It is a streaming protocol; therefore, you can use it with a serial port, but the port must be completely reliable and flow controlled.

If you do not use the **protocol** command to specify a protocol, **uucico** considers the protocols in the order given above, and chooses one based on the characteristics of the port and the dialer specified in the files **/usr/lib/uucp/port** and **/usr/lib/uucp/dial**. The port and dial must meet the requirements of a protocol before **uucico** will consider it during negotiation with *remotesystem*.

If neither the **seven-bit** nor the **reliable** command is used, **uucico** will use the **i** protocol (subject, of course, to what is supported by the remote system; you cannot assume that all systems support the **i** protocol). No current protocol can be used with a port for which you have specified **seven-bit true** and **reliable false**. You must use the **protocol** command for the system, or **uucico** will select no protocol at all. (The only reasonable choice would be **protocol f**.) You can use the command

> **protocol-parameter** *protocol parameter* **[***argument ...* **]**

to modify a protocol's default parameters.

The **i** protocol recognizes the following parameters:

window *size*
: Request that *remotesystem* use a *size* window, between one and 31, inclusive. The default is 16.

packet-size *size*
: Request that *remotesystem* use a packet of *size* bytes, between one and 4,095, inclusive. The default is 1,024.

remote-window *size*
: Ignore the window size requested by *remotesystem*, and instead us a window of *size*. The default is zero, which means that the request of *remotesystem* is honored.

remote-packet-size *size*
: Ignore the packet size requested by *remotesystem*, and instead use a packet of *size* bytes. The default is zero, which means that the request of *remotesystem* is honored.

sync-timeout *seconds*
: Wait *seconds* for a SYNC packet from *remotesystem*. The default is ten.

sync-retries *number*
: Resend a SYNC packet *number* times before giving up. The default is six.

timeout *seconds*
: Wait *seconds* for an incoming packet before sending a NAK (negative acknowledgement) The default is ten.

retries *number*
: Resend a packet or negative acknowedgement *number* times before giving and closing the connection. The default is six.

errors *number*
Quit after *number* errors have occurred. The default is 100.

error-decay *number*
Decrease the count of errors by one after receiving *number* packets. This keeps occasional errors from accumulating during a long conversation, and so aborting what is actually a successful transmission. The default is ten.

ack-frequency *number*
Send an acknowledgement after receiving *number* packets. By default, this is set to half the requested size of the window.

The protocols **g** and **G** recognize the following parameters:

window *size*
Request that *remotesystem* use a *size* window, between one and seven, inclusive. The default is seven.

packet-size *size*
Request that *remotesystem* use a packet size of *size* bytes. **size** must be a power of two, between 32 and 4,096, inclusive. The default is 64, which is the only packet size supported by many older UUCP packages.

startup-retries *number*
Retry the entire initialization sequence *number* times before quitting. The default is eight.

init-retries *number*
Retry one phase of the initialization sequence *number* times before quitting. The default is four.

init-timeout *seconds*
Wait for *seconds* before timing out one phase of the initialization sequence. The default is ten.

retries *number*
Resend a packet or a request for a packet *number* times before quitting. The default is six.

timeout *seconds*
Wait for *seconds* for a packet or an acknowledgement before timing out. The default is ten.

garbage *number*
Drop the connection after receiving *number* unrecognized characters. *number* must be larger than the packet size. The default is 10,000.

errors *number*
Quit after *number* errors have occurred. Errors include malformed packets, out-of-order packets, bad checksums, and packets rejected by the remote system. The default is 100.

error-decay *number*
Decrease the count of errors by one after receiving *number* packets. This keeps occasional errors from accumulating during a long conversation, and so aborting what is actually a successful transmission. The default is ten.

remote-window *size*
Ignore the window size requested by *remotesystem*, and instead us a window of *size*. The default is zero, which means that the request of *remotesystem* is honored.

remote-packet-size *size*
Ignore the packet size requested by *remotesystem*, and instead use a packet of *size* bytes. The default is zero, which means that the request of *remotesystem* is honored.

short-packets true|false
If **true**, optimize transmission by sending shorter packets when there is less data to send. This confuses some UUCP packages; when connecting to such a package, this parameter must be set to **false**. The default is **true** for the **g** protocol and **false** for the **G** protocol.

The **a** protocol mimics the Z-modem protocol. It supports the following parameters: All take numeric arguments, except for **escape-control**, which takes a Boolean argument:

timeout *seconds*
Wait *seconds* for a packet before timing out. The default is ten.

retries *number*
Resend a packet *number* times before quitting. The default is ten.

startup-retries *number*
Retry sending the initialization sequence *number* times before quitting. The default is four.

garbage *number*
Drop the connection after receiving *number* unrecognized "garbage" characters. *number* must be larger than the packet size. The default is 2,400.

send-window *number*
Send *number* characters before waiting for an acknowledgement. The default is 1,024.

escape-control true|false
If **true**, **uucico** can use the protocol over a connection that does not transmit certain control characters, such as **XON** or **XOFF**. The connection must still transmit eight-bit characters other than control characters. The default is **false**.

The **j** protocol can be used over an eight-bit connection that will not transmit certain control characters. It accepts the same parameters as the **i** protocol, plus the following:

avoid *string*
Avoid every character defined in *string*. *string* can contain escape sequences, as defined above for the chat script. Each character must be a non-printable ASCII character (i.e., ASCII values less than 32 or greater than 126). Each must be defined using using the escape sequence *****DDD*, where *DDD* gives three octal digits.

The default value is **\021\023** (i.e., **XON** and **XOFF**). If the package is configured to use **HAVE_BSD_TTY**, then you may have to avoid **\377** as well.

The **f** protocol is intended for use with error-correcting modems only. It checksums each file as a whole, so any error causes the entire file to be retransmitted. It recognizes the following parameters:

timeout *seconds*
Wait *seconds* before timing out. The default is 120.

retries *number*
Retry sending a file *number* times before quitting. The default is two.

The protocols **t** and **e** recognize the following parameter:

timeout *seconds*
Wait *seconds* before timing out. The default is 120.

Note that the command **protocol-parameter** can be used in files **/usr/lib/uucp/dial** and **/usr/lib/uucp/port** as well as in **sys**. In case of a conflict between the entries in these files, the entries in **dial** takes precendence; then those in **port**. The entries in **sys** have lowest precedence.

File Transfers

The following commands help to control the transfer of files.

send-request yes|no
Set whether *remotesystem* can request files from your system. The default is **yes**, that is, *remotesystem* may request files.

receive-request yes|no
Set whether *remotesystem* can send files to your system. The default is **yes**, that is, *remotesystem* may send files.

request yes|no
This combines the commands **send-request** and **receive-request** into one.

call-transfer yes|no
Set whether your system may transfer files to *remotesystem* when it calls *remotesystem*. The default is **yes**.

called-transfer yes|no
Set whether your system may transfer files to *remotesystem* when *remotesystem* calls your system. The default is **yes**.

transfer yes|no
This combines commands **call-transfer** and **called-transfer** into one.

call-local-size *number timestring*
Send or receive no file larger than *number* bytes when your system calls *remotesystem* during the time defined in *timestring*. You can use this command to help limit the length of a call made during times when toll charges are higher. The description of a system may contain multiple **call-local-size** commands, one for each period during which you wish to limit activity. The default is to have no limit.

Please note that the size-control commands, are guaranteed only to limit the size of files being sent by your system. The size of files being sent from *remotesystem* can be checked if the other system is running the Taylor UUCP package. Other UUCP packages do not understand a maximum-size request, nor do they inform this package of the size of the files they are sending.

call-remote-size *number timestring*
Limit to *number* bytes the size of a file that can be fetched by remote request (either by your system on *remotesystem* or by it on your system) when your system calls *remotesystem* during the time defined in *timestring*. The description of a *remotesystem* can contain multiple **call-local-size** commands, one for each period during which you wish to limit activity. The default is to have no limit.

called-local-size *number timestring*
Send or receive no file larger than *number* bytes when *remotesystem* calls your system during the time defined in *timestring*. The description of a system may contain multiple **call-local-size** commands, one for each period during which you wish to limit activity. The default is to have no limit.

called-remote-size *number timestring*
Limit to *number* bytes the size of a file that can be fetched by remote request (either by your system on *remotesystem* or by it on your system) when *remotesystem* calls your system during the time defined in *timestring*. The description of a *remotesystem* can contain multiple **call-local-size** commands, one for each period during which you wish to limit activity. The default is to have no limit.

local-send *directorylist ...*
Limit to *directorylist* the directories from which your system can send files to *remotesystem*. Each directory in *directorylist* must be separated by white space. You can use a tilde '~' for the public directory, i.e., **/usr/spool/uucppublic**. Listing a directory within *directorylist* lets your system send all files within that directory and its subdirectories.

Prefixing a directory with an exclamation point '!' specifically excludes it and its subdirectories from being sent. For example, the command

```
local-send /v/fwb !/v/fwb/Personal
```

means that your system can send all files in directory **/v/fwb** to *remotesytem* except for the files in directory **/v/fwb/Personal**.

uucico reads *directorylist* from left to right, and the last directory to apply takes effect. Therefore, you should list directories from top down. The default is the root directory, i.e., your system can send any file to *remotesystem*.

remote-send *directorylist*
Limit to *directorylist* the directories from which *remotesystem* can request files. The default is **/usr/spool/uucppublic**.

local-receive *directorylist*
Limit to *directorylist* the directories into which your system can write files requested from *remotesystem*. The default is **/usr/spool/uucppublic**.

remote-receive *directorylist*
Limit to *directorylist* the directories on *remotesystem* into which your system can write files. The default is **/usr/spool/uucppublic**. This command cannot override permissions that *localsystem* has granted to your system.

forward-to *systemlist*
Limit the systems to which your system will forward files to those named in *systemlist*. A *systemlist* of **ANY** lets *remotesystem* forward files through your system to any system it wants. The default is not to permit forwarding to other systems. Note that if you permit *remotesystem* to execute the command **uucp** on your system, it effectively has permission to forward to any system.

forward-from *systemlist*
Limit the systems from which *remotesystem* request files through your system to those named in *systemlist*. A *systemlist* of **ANY** lets *remotesystem* request files from any system. The default is not to permit *remotesystem* to request files from anywhere. Note that if you permit *remotesystem* to execute the command **uucp** on your system, it effectively has permission to fetch files through your system from any other system.

forward *systemlist*
This command combines the commands **forward-to** and **forward-from**.

Miscellaneous Commands

The following gives miscellaneous commands that can be used in **sys**:

sequence yes|no
If **true**, this commands tells **uucico** to use the conversation sequencing for *remotesystem*. This means that if somebody impersonates *remotesystem* and logs into your system, that fact will be discovered the next time *remotesystem* actually calls. The default is **false**.

command-path *path*
Limit to *path* the directories that a command file forwarded from *remotesystem* can search for commands to execute. The directories named in *path* must be separated by white space.

commands *commandlist*
Limit the commands that *remotesystem* can execute on your system to those named in *commandlist*. A *commandlist* **ALL** lets **remotesystem** execute all programs on your system. The default is **rnews rmail**.

free-space *number*
This command tells **uucico** always to leave free *number* bytes of space in a file system. This command ensures that **uucico** will not permit *remotesystem* to fill up your file system. If an incoming file is too large to leave *number* bytes free on the file system, **uucico** refuses the file or aborts its downloading.

Note that not every version of UUCP supports this.

pubdir *directory*
Name the public directory available to remote UUCP systems. The default is **/usr/spool/uucppublic**.

debug *activitylist*
Log each UUCP activity named in *activitylist* when talking with *remotesystem*. These logs can help you debug problems with **uucico** and **cu**. **uucico** recognizes the following activities:

abnormal	**chat**	**handshake**
uucp-proto	**proto**	**port**
config	**spooldir**	**execute**
incoming	**outgoing**	

none tells **uucico** to log nothing.

max-remote-debug *typelist*
Limit to *typelist* the types of debugging that *remotesystem* can request on your system. This command is designed to stop *remotesystem* from filling your disk with debugging information.

Defaults

The following gives the default settings for all systems. You should regard these as appearing at the head of **/usr/lib/uucp/sys**, even though they do not explicit appear in that file:

```
time Never
chat "" \r\c ogin:-BREAK-ogin:-BREAK-ogin: \L word: \P
chat-timeout 10
callback n
sequence n
request y
transfer y
local-send /
remote-send ~
local-receive ~
remove-receive ~
commands rnews rmail
max-remote-debug abnormal,chat,handshake
```

Example

The following gives the entry in **/usr/lib/uucp/sys** for the system **mwcbbs**, which is the Mark Williams bulletin board:

```
system mwcbbs
time Any
baud 2400
port MODEM
phone 17085590412
chat "" \r\d\r in:--in: nuucp word: public word: serialnumber
protocol g
protocol-parameter g window 3
protocol-parameter g packet-size 64
myname bbsuser
request yes
transfer yes
remote-send /usr/spool/uucppublic /tmp
remote-receive /usr/spool/uucppublic /tmp
commands rmail uucp
```

The following describes each command in detail:

system mwcbbs
Name the sytem being described, in this case **mwcbbs**.

time Any
Set the time during your system can contact **mwcbbs**, in this case any time.

baud 2400
Set the speed at which your system can contact **mwcbbs**; here 2400 baud.

port MWCBBS
Set the port through which your system can dial out to **mwcbbs**; here, port **MWCBBS**. This port is defined in the file **/usr/lib/uucp/port**; for details on this file and how to modify its data, see the Lexicon entry for **port**.

phone 17085590412
This gives the telephone number of **mwcbbs**.

chat " " \r\d\r in:--in: nuucp word: public word: *serialnumber*
Give the chat script with which your system logs into **mwcbbs**. See the section on chat scripts, above, for details on how to interpret this command.

protocol g
Use the **g** protocol.

protocol-parameter g window 3
Set the window used with protocol **g** to three.

protocol-parameter g packet-size 64
Set the size of the packet used with protocol **g** to 64 bytes.

myname bbsuser
Identify yourself to **mwcbbs** as user **bbsuser**.

request yes
Let **mwcbbs** send files to your system, and request files from your system. Setting this to **no** would forbid **mwcbbs** to do so.

transfer yes
Permit files to be transferred from your system to **mwcbbs**, and vice versa, regardless of whether your system calls **mwcbbs** or vice versa.

remote-send /usr/spool/uucppublic /tmp
Permit **mwcbbs** to request files only from directories **/usr/spool/uucppublic** and **/tmp**.

remote-receive /usr/spool/uucppublic /tmp
Limit the directories into which your system will write files requested from **mwcbbs** to **/usr/spool/uucppublic /tmp**.

commands rmail uucp
Limit the commands that **mwcbbs** can execute on your system to **rmail** and **uucp**.

See Also

Administering COHERENT, dial, port, UUCP
Baker, S.: From UUCP to eternity. *UNIX Review*, April 1993, pp. 15-26. *Summarizes the history of UUCP and describes the working of the* **g** *protocol.*

Notes

Only the superuser **root** can edit **/usr/lib/uucp/sys**.

The file **sys** supports many commands in addition to the ones described here. This article describes only those commands that might be used in typical UUCP connections. For more information, see the original Taylor UUCP documentation, which is in the archive **/usr/src/alien/uudoc104.tar.Z**.

sysconf() — System Call (libc)

Get configurable system variables
#include <unistd.h>
long sysconf(*name***)**
int *name***;**

sysconf() returns the value of the system limit or option identified by *name*.

In the following table, the left column gives a symbolic constant to which *name* can be set, and the right column gives the corresponding system variable (as defined in **<limits.h>** and **<unistd.h>**) that **sysconf()** reads and returns:

Name	*Variable*
_SC_ARG_MAX	**ARG_MAX**
_SC_CHILD_MAX	**CHILD_MAX**
_SC_CLK_TCK	**CLK_TCK**
_SC_NGROUPS_MAX	**NGROUPS_MAX**
_SC_OPEN_MAX	**OPEN_MAX**
_SC_PASS_MAX	**PASS_MAX**
_SC_JOB_CONTROL	**_POSIX_JOB_CONTROL**
_SC_SAVED_IDS	**_POSIX_SAVED_IDS**
_SC_VERSION	**_POSIX_VERSION**

The following describes the values returned in more detail:

ARG_MAX
Maximum number of bytes that can be occupied by a process's argument list and environment.

CHILD_MAX
Number of processes a user can run simultaneously.

CLK_TCK
Length of a clock tick, in microseconds.

NGROUPS_MAX
The maximum number of groups to which a user can belong, in addition to her primary group.

OPEN_MAX
The number of files a process can have open simultaneously.

PASS_MAX
The maximum length of a password. Please note that the constant **_SC_PASS_MAX** is defined only for programs compiled for UNIX System V release 4.

_POSIX_JOB_CONTROL
This is a Boolean flag that indicates whether the operating system supports the POSIX job-control functions.

_POSIX_SAVED_IDS
This is a Boolean flag that indicates whether the operating system permits each process to have a saved set-user ID and a saved set-group ID.

_POSIX_VERSION
This is a long integer that encodes the four-digit year and two-digit month of approval for the version of the POSIX standard supported by the operating system. For example, 199009L indicates the version approved in September of 1990.

The value of variable **CLK_TCK** can vary; you should not assume that it is a compile-time constant.

If *name* is an invalid value, **sysconf()** returns -1 and set **errno** to an appropriate value. If **sysconf()** fails due to a value of *name* that is not defined on the system, it returns -1 without setting **errno**.

Example

At the time of this writing (August 1994), the program

```
#include <unistd.h>
main()
{
	printf("_SC_ARG_MAX: %d\n", sysconf(_SC_ARG_MAX));
	printf("_SC_CHILD_MAX: %d\n", sysconf(_SC_CHILD_MAX));
	printf("_SC_CLK_TCK: %d\n", sysconf(_SC_CLK_TCK));
	printf("_SC_NGROUPS_MAX: %d\n", sysconf(_SC_NGROUPS_MAX));
	printf("_SC_OPEN_MAX: %d\n", sysconf(_SC_OPEN_MAX));
	printf("_SC_JOB_CONTROL: %d\n", sysconf(_SC_JOB_CONTROL));
	printf("_SC_SAVED_IDS: %d\n", sysconf(_SC_SAVED_IDS));
	printf("_SC_VERSION: %d\n", sysconf(_SC_VERSION));
}
```

returns the following values:

```
	_SC_ARG_MAX: 5120
	_SC_CHILD_MAX: 25
	_SC_CLK_TCK: 100
	_SC_NGROUPS_MAX: 32
	_SC_OPEN_MAX: 60
	_SC_JOB_CONTROL: 0
	_SC_SAVED_IDS: 1
	_SC_VERSION: 199009
```

See Also

libc, unistd.h
POSIX Standard, §4.8.1

Notes

Programs can use the appropriate **#ifndef** guards to control whether they use **sysconf()** or a symbol from **<limits.h>** for each kind of limit. For example:

```
	#include <unistd.h>
	#include <limits.h>
```

```
#ifdef      _SC_OPEN_MAX
      max = sysconf (_SC_OPEN_MAX);
#elif defined (OPEN_MAX)
      max = OPEN_MAX;
#else
/* either complain, or make some rational assumption, e.g. */
#error      Open file descriptor limits cannot be determined
#endif
```

sysi86() — System Call (libc)

Identify parts within Intel-based machines
#include <sys/sysi86.h>
int sysi86(*hardware***,** *type***)**
int *hardware***,** ***type***;**

The system call **sysi86()** identifies parts within Intel-based computers. *hardware* names the machine part that you wish to identify; you should always use one of the constants defined in header file **<sys/sysi86.h>**. *type* point to the **int** into which **sysi86()** writes an identifying code.

sysi86() returns -1 if it was unable to read your machine. It returns a value other than -1 if it succeeds in reading your machine.

Example

The following program identifies the type of floating-point processor in your machine.

```
#include <sys/sysi86.h>

#ifndef     FP_NO
/*
 * The following header may be needed to get the FP_... constants on some
 * other implementations of the iBCS2 specification; while the sysi86()
 * system call and the SI86FPHW constant are part of the iBCS2 specification,
 * the FP_... constants and the <sys/fp.h> header are not.
 */
#include <sys/fp.h>
#endif

const char *
floating_point_provider ()
{
      int fp_type;

      if (sysi86 (SI86FPHW, & fp_type) == -1)
            return "unable to retrieve FP type";

      switch (fp_type) {

      case FP_NO:
            return "no FP hardware or emulation available";

      case FP_SW:
            return "software emulation of FP hardware";

      case FP_287:
            return "80287 hardware FP";

      case FP_387:
            return "80387 or 80486DX hardware FP";

      default:
            return "unknown floating-point provider";
      }
}

main()
{
      printf("%s\n", floating_point_provider());
}
```

See Also

libc, sysi86.h

Notes

At present under COHERENT, this system call can interrogate a machine only for the type of its floating-point processor.

system() — General Function (libc)

Pass a command to the shell for execution

#include <stdlib.h>
int system(*commandline***) char ***commandline***;**

system() passes *commandline* to the shell **sh**, which loads it into memory and executes it. **system()** executes commands exactly as if they had been typed directly into the shell. **system()** may be used by commands such as **ed**, which can pass commands to the COHERENT shell in addition to processing normal interactive requests.

Example

This example uses **system** to list the names of all C source files in the parent directory.

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

main()
{
	system("cd .. ; ls *.c > mytemp; cat mytemp");
}
```

See Also

exec fork(), libc, popen(), stdlib.h, wait()
ANSI Standard, §7.10.4.5

Diagnostics

system() returns the exit status of the child process, in the format described in **wait()**: exit status in the high byte, signal information in the low byte. Zero normally means success, whereas nonzero normally means failure. This, however, depends on the *command*. If the shell is not executable, **system()** returns a special code of octal 0177.