## *raise()* — General Function (libc)

Let a process send a signal to itself
**#include <signal.h>**
**int raise(***signal***)**
**int** *signal***;**

**raise()** sends *signal* to the program that is currently being executed. If called from within a signal handler, the processing of this signal may be deferred until the signal handler exits.

### *Example*

This example sets a signal, raises it itself, then allows the signal to be raised interactivly. Finally, it clears the signal and exits.

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

void gotcha(void);

void
setgotcha(void)
{
      if(signal(SIGINT, gotcha) == SIG_ERR) {
            printf("Couldn't set signal\n");
            abort();
      }
}

void
gotcha(void)
{
      char buf[10];

      printf("Do you want to quit this program? <y/n> ");
      fflush(stdout);
      gets(buf);

      if(tolower(buf[0]) == 'y')
            abort();

      setgotcha();
}

main(void)
{
      char buf[80];

      setgotcha();
      printf("Set signal; let's pretend we get one.\n");
      raise(SIGINT);

      printf("Returned from signal\n");
      printf("Try typing <ctrl-c> to signal <enter> to exit");
      fflush(stdout);
      gets(buf);
```

```
      if(signal(SIGINT, SIG_DFL) == SIG_ERR) {
            printf("Couldn't lower signal\n");
            abort();
      }
      printf("Signal lowered\n");
      exit(EXIT_SUCCESS);
}
```

### See Also

**libc, signal(), signal.h**
ANSI Standard, §7.7.2.1

## *ram* — Device Driver

Driver for manipulating RAM

The COHERENT **ram** devices let you allocate and use the random-access memory (RAM) of the computer system directly. A typical use is for a RAM disk, which is a COHERENT file system kept in memory rather than on a floppy disk or hard disk.

The COHERENT RAM device driver has major number 8. You can access it either as a block-special device or as a character-special device. The high-order bit of the minor number gives the RAM device number (0 or 1); as you can see, you can have no more than two RAM devices in memory at any one time. The low-order seven bits give the device's size in 64-kilobyte chunks.

The first call to **open()** on a RAM device with nonzero size (1 to 127) allocates memory for the device; **open()** fails if sufficient memory is not available. Accessing a RAM device with a minor number that specifies size zero frees the allocated memory, provided all earlier **open()** calls have been closed.

Initially, COHERENT includes two block-special devices for RAM disks: the 512-kilobyte device **/dev/ram0** (8, 8) and the 192-kilobyte device **/dev/ram1** (8, 131). It also includes the devices **/dev/ram0close** (8, 0) and **/dev/ram1close** (8, 128). You should resize the RAM devices to suit the amount of memory available on your system.

### Examples

The following example formats and mounts a 512-kilobyte RAM disk on directory **/fast**.

```
mkdir /fast
/etc/mkfs /dev/ram0 1024
/etc/mount /dev/ram0 /fast
```

When the RAM disk is no longer needed, its allocated memory can be freed as follows:

```
/etc/umount /dev/ram0
cat /dev/null >/dev/rram0close
```

The next example replaces the default **/dev/ram0** with a one-megabyte device that contains a COHERENT file system. The minor number 16 specifies RAM device 0 and a size of one megabyte (i.e., 16 chunks of 64 kilobytes each). The new RAM device contains 2,048 blocks of 512 bytes each.

```
rm /dev/ram0
/etc/mknod /dev/ram0 b 8 16
/etc/mknod /dev/rram0 c 8 16
/etc/mkfs /dev/ram0 2048
chmod ugo=rw /dev/ram0
chmod ugo=rw /dev/rram0
```

The command **chmod** is necessary to make the new RAM drive accessible.

### Files

**/dev/ram***

### See Also

**compress, device drivers, fsck, mkfs, mount, ramdisk, umount, uncompress, zcat**

### Notes

Moving frequently used commands or files to a RAM disk can improve system performance substantially. However, the contents of a RAM device are lost if the system loses power, reboots, or crashes. Therefore, you should frequently back up files from the RAM disk to a more permanent medium.

If a RAM device uses most but not all available system memory, its **open()** call will succeed but subsequent commands may fail because insufficient memory remains for the system.

The COHERENT installation program **/etc/build** uses RAM device **/dev/ram1** as a RAM disk during installation. Commands **compress**, **uncompress**, **zcat**, and **fsck** sometimes use **/dev/ram1** as a temporary storage device. Users should avoid using **/dev/ram1** as a RAM disk because of these programs. In addition, users of **compress**, **uncompress**, and **zcat** may have to change the size of **/dev/ram1** from the default size of 192 to 512 kilobytes, to handle files compressed to 16 bits. The following script makes this change; note that it must be run by the superuser **root**:

```
cat /dev/null >/dev/rram1close
rm /dev/ram1 /dev/rram1
mknod /dev/ram1  b 8 136
mknod /dev/rram1 c 8 136
```

## *ramdisk* — System Administration

Script to create a RAM-disk
**/usr/bin/ramdisk**

**ramdisk** is a script that creates a 500-kilobyte RAM disk that is accessed via device **/dev/ram0**.

To use **ramdisk** to create a RAM disk for you at boot-time, do the following:

1. Log in as the superuser **root**.
2. Type:

   ```
   touch /dev/ram0close
   ```

   This closes the RAM disk and removes it from memory.
3. Remake the ram disk as a smaller size device. As an example, we'll make one that is 64 kilobytes. Type the command:

   ```
   /etc/mknod /dev/ram0 b 8 1
   ```

   To break down this command:

   **/etc/mknod**
   This is the command that creates a special file (e.g., a block-special file) through which a device like a printer or RAM is accessed.

   **/dev/ram0**
   The directory path and name of your RAM disk.

   **b** This argument tells **mknod** to build a block-special file. Every device like a printer, floppy drive, COM port, or RAM drives, are considered a "block special file."

   **8** This is the major device number for a RAM drive. All major-device numbers are listed in the Lexicon entry for "device drivers."

   **1** This is the minor device number of your new **ram0.** This shows that the **ram0** you are building will be 64 kilobytes in size. If the minor device number would have been '2', then the size of **ram0** would have been two times 64, or 128 kilobytes. Each increment in the minor-device number is equal to an additional 64 kilobytes for the RAM device. A minor device of 16 multiplied by 64 kilobytes would equal a one megabyte size RAM drive.
4. Next, make a file system in **ram0**:

   ```
   /etc/mkfs /dev/ram0 128
   ```

   The number "128" is exactly twice the memory size, in this case 64 kilobytes. Whatever size memory you choose to allocate to a RAM device, the block size you specify in the **mkfs** command will be double. A one-megabyte RAM device for example would have 2,048 blocks.

**5.** Your new RAM disk is now ready to be mounted. Typically, you would mount ram0 in a directory named **fast** or some other unique name, so to mount, type;

```
/etc/mkdir /fast
/etc/mount /dev/ram0 /fast
```

If **/fast** already exists, do not create it.

Once you have created your RAM disk, you should load commonly used utilities into it.

If you wish to create a RAM disk automatically whenever you boot COHERENT, un-comment and edit the appropriate lines in file **/etc/rc**.

### See Also

**Administering COHERENT, ram, rc**

### Notes

This script only works in machines that have sufficient memory.

## *rand()* — Random-Number Function (libc)

Generate pseudo-random numbers
**#include <stdlib.h>**
**int rand()**

**rand()** generates a set of pseudo-random numbers. It returns integers in the range 0 to 32,767, and purportedly has a period of 2^32. **rand()** will always return the same series of random numbers unless you first call the function **srand()** to change **rand()**'s *seed*, or beginning-point.

### Example

The following example uses **rand()** to implement the "Let's Make a Deal" game of probability described by Massimo Piattelli-Palmarini in the March/April 1991 issue of *Bostonia* magazine. In brief, an investigator places a dollar bill into one of three boxes. A subject enters the room and guesses which box holds the bill. The investigator then opens one of the two unselected boxes (one that is always empty), shows it to the subject, then offers the subject a choice: either stand pat with the box he has selected, or switch to the other non-selected box. The laws of probability state that the subject should always switch from the box he has selected; this example program tests that hypothesis.

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

main()
{
      int box[3], win, i, j;

      srand(time(NULL));

      /* Test 1:  the subject always stands pat.  For the sake of simplicity,
       * the subject always chooses box 0. */
      for (i = 0, win = 0; i < 1500; i++) {
            for (j = 0; j < 3; j++)
                  box[j] = 0;

            box[rand()%3]++;

            if (box[0])
                  win++;
      }
      printf("Test 1, always stand pat:  1500 iterations, %d winners\n", win);

      /* Test 2:  the subject always switches boxes. */
      for (i = 0, win = 0; i < 1500; i++) {
            for (j = 0; j < 3; j++)
                  box[j] = 0;

            box[rand()%3]++;
```

```
            /* if box 2 is empty, pick box 1 */
            if (!box[2])
                  win += box[1];
            else
                  win += box[2];
      }
      printf("Test 2, always switch:  1500 iterations, %d winners\n", win);

      /* Test 3:  the subject switches boxes randomly. */
      for (i = 0, win = 0; i < 1500; i++) {
            for (j = 0; j < 3; j++)
                  box[j] = 0;

            box[rand()%3]++;

            /* if box 2 is empty, pick box 1 */
            if (!box[2]) {
                  if (rand()%2)
                        win += box[1];
                  else
                        win += box[0];
            } else {
                  if (rand()%2)
                        win += box[2];
                  else
                        win += box[0];
            }
      }
      printf("Test 3, randomly switch:  1500 iterations, %d winners\n", win);
}
```

### See Also

**libc, RAND_MAX, srand(), stdlib.h**
*The Art of Computer Programming*, vol. 2
ANSI Standard, §7.10.2.1
POSIX Standard, §8.1

### Notes

This function cannot be used with any of the "rand48" functions. For an overview of these functions, see the entry for **srand48()**.

## *RAND_MAX* — Manifest Constant

Largest size of a pseudo-random number
**#include <stdlib.h>**

**RAND_MAX** is a manifest constant that is defined in the header **stdlib.h**. It indicates the largest pseudo-random number that can be returned by the function **rand()**.

### Example

For an example of using this manifest constant in a program, see **rand()**.

### See Also

**manifest constant, rand(), stdlib.h**
ANSI Standard, §7.10

## *random()* — Sockets Function (libsocket)

Return a random number
**int random();**

The function **random()** returns a random number. It is a synonym for **rand()**.

### See Also

**libsocket, rand()**

## *random access* — Definition

In the context of computing, **random access** means that an entity, such as memory, can be accessed at any point, not just at the beginning. This means that all points within memory can be accessed equally quickly. This contrasts with *sequential access*, in which entities must be accessed in a particular order, so that some entities take longer to access than do others.

A tape drive is an example of a sequential access device, i.e., the order in which data are read is dictated by the order in which they stream past the tape head. Random-access memory (RAM) is an example of random access. Hard disks and floppy disks combine elements of random access and sequential access.

RAM, which usually consists of semiconductor integrated circuits, is also strictly random access. In this regard, the term "RAM" is slightly misleading; a more accurate name would be "read/write memory", to contrast RAM with read-only memory (ROM), which is also random access memory.

### *See Also*

**read-only memory, Programming COHERENT**

## *ranlib* — Command

Create index for object library
**ranlib** *library ...*

The **ranlib** is a "directory" that appears at the beginning of each library. It contains the name of each global symbol (i.e., function name) that appears within the library, and a pointer to the module in which that symbol is defined. Thus, the ranlib eliminates the need for the linker to search the entire library sequentially to find a given global symbol, which speeds up linking noticeably.

If the date on the library file is later than that in the ranlib header, the linker will ignore the ranlib and perform a sequential search through the library; the linker will also send the warning message

```
Outdated ranlib
```

to the standard error device. This is done to prevent the accidental use of an outdated ranlib, which could be disastrous.

The command **ranlib** creates a ranlib header for an archive. If the header already exists, **ranlib** updates it.

### *Files*

**__.SYMDEF** — Index module

### *See Also*

**ar, ar.h, commands, ld**

### *Diagnostics*

**ranlib** issues appropriate messages for I/O errors or bad format files. It does not rewrite a library until the last possible moment, so the library is usually unchanged in case of error. **ranlib** processes each library independently. The exit status is the number of libraries in which errors were encountered.

**ranlib** is a link to the archiver **ar**.

## *rc* — System Administration

Perform standard maintenance chores
**/etc/rc**

The shell script **/etc/rc** is executed by the **init** process when the COHERENT system enters multi-user mode. The commands in **rc** do such things as set the local time zone and initialize file **/usr/adm/wtmp**, which holds records of user logins.

### *See Also*

**Administering COHERENT, brc, init**

## *read-only memory* — Definition

As its name suggests, **read-only memory,** or ROM, is memory that can be read but not overwritten. It most often is used to store material that is used frequently or in key situations, such as a language interpreter or a boot routine.

### See Also

**Programming COHERENT, random access**

## *read* — Command

Assign values to shell variables
**read** *name ...*

**read** reads a line from the standard input. It assigns each token of the input to the corresponding shell variable *name*. If the input contains fewer tokens than the number of names specified, **read** assigns the null string to each extra variable. If the input contains more tokens than the number of names specified, **read** assigns the last *name* in the list the remainder of the input.

**read** normally returns an exit status of zero. If it encounters end of file or is interrupted while reading the standard input, it returns one.

The shell executes **read** directly.

### Example

The command

```
read foo bar baz
hello how are you
```

parses the line "hello how are you" and assigns the tokens to, respectively, the shell variables **foo**, **bar**, and **baz**. If you further type

```
echo $foo
echo $bar
echo $baz
```

you will see:

```
hello
how
are you
```

### See Also

**commands, ksh, sh**

## *read()* — System Call (libc)

Read from a file
**#include <unistd.h>**
**int read(***fd***,** *buffer***,** *n***)**
**int** *fd***; char** **buffer***; **int** n***;**

**read()** reads up to *n* bytes of data from the file descriptor *fd* and writes them into *buffer*. The amount of data actually read may be less than that requested if **read()** detects **EOF**. The data are read beginning at the current seek position in the file, which was set by the most recently executed **read()** or **lseek()** routine. **read()** advances the seek pointer by the number of characters read.

If all goes well, **read()** returns the number of bytes read; thus, zero bytes signals the end of the file. It returns -1 if an error occurs, e.g., *fd* does not describe an open file, or if *buffer* contains an illegal address.

### Example

For an example of how to use this function, see the entry for **open()**.

### See Also

**libc, unistd.h**
POSIX Standard, §6.4.1

### Notes

**read()** is a low-level call that passes data directly to COHERENT. It should not be mixed with the STDIO routines **fread()**, **fwrite()**, or **fopen()**.

## *readdir()* — General Function (libc)

Read a directory stream
**#include <sys/types.h>**
**#include <dirent.h>**
**struct dirent *readdir(***dirp***)**
**DIR ****dirp***;**

The COHERENT function **readdir()** is one of a set of COHERENT routines that manipulate directories in a device-independent manner. It reads the directory stream pointed to by *dirp* and returns information about the next active entry within the stream. It does not report on inactive entries.

**readdir()** returns a pointer to a structure of type **dirent**, which contains information about the next active entry within the stream. The internal structure may be overwritten by another operation on the same directory stream. The amount of memory needed to hold a copy of the internal structure is given by the value of a macro, **DIRENTSIZ(strlen(direntp->d_name))**, not by **sizeof(struct dirent)** as one might expect.

**readdir()** returns NULL if it has reached the end of the directory, has detected an invalid location within the directory, or if an error occurs while it is reading the directory. If an error occurs, **readdir()** exits and sets **errno** to an appropriate value.

### Example

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

### See Also

**closedir(), dirent.h, getdents(), libc, opendir(), rewinddir(), seekdir(), telldir()**
POSIX Standard, §5.1.2

### Notes

The **dirent** routines buffer directories; and because directory entries can appear and disappear as other users manipulate the directory, your application should continually rescan a directory to keep an accurate picture of its active entries.

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

## *readline()* — Editing Function (libedit)

Read and edit a line of input
**char *readline(***prompt***)**
**char ****prompt***;**

The function **readline()** displays on the standard output the text to which *prompt* points, then accepts what the user types. It lets the user type simple, EMACS-style commands to edit what she has typed; when the user types (¢), **readline()** returns the line of text with the trailing newline removed.

**readline()** returns a pointer to the newly entered line. This return value can be passed to the function **add_history()**, which adds it to an internal "history" buffer. The user can use a command within **readline()** recall a saved line, re-edit it, and re-submit it.

**readline()** returns NULL when the user types EOF, or if it cannot allocate space for the line of input. Otherwise, it returns the address of the edited string that the user input.

### Editing Commands

**readline()** provides a simple, EMACS-like editing interface. You can type control characters or escape sequences to edit a line before it is sent to the calling program, much like the EMACS editing feature of the Korn shell.

**readline()** recognizes the following editing commands:

**<ctrl-A>** Move the cursor to the beginning of the line.
**<ctrl-B>** Move the cursor one character to the left (backwards).

**<ctrl-D>** Delete the character under which the cursor is positioned (the "current character").
**<ctrl-E>** Move the cursor to end of line.
**<ctrl-F>** Move the cursor one character to the right (forwards).
**<ctrl-G>** Ring the bell.
**<ctrl-H>** Delete the character to the left of the cursor. Note that **<ctrl-H>** is the character that normally is output by the **<backspace>** key.
**<ctrl-I>** Complete file name. Note that **<ctrl-I>** is the character that normally is output by the **<tab>** key.
**<ctrl-J>** Submit the line for processing. Note that **<ctrl-J>** is the character that normally is output by the (¢) key.
**<ctrl-K>** Kill all text from the cursor to end of line.
**<ctrl-L>** Redisplay the line.
**<ctrl-M>** Submit the line for processing. Note that on some systems, **<ctrl-M>** is output by the (¢) key.
**<ctrl-N>** Get the next line from the history buffer.
**<ctrl-P>** Get the previous line from the history buffer.
**<ctrl-R>** Search backwards through the history buffer for a given string.
**<ctrl-T>** Transpose the character over the cursor with the character to its left.
**<ctrl-V>** Insert next character into the line, even if it is a control character. Note that under MicroEMACS, this command is bound to **<ctrl-Q>**.
**<ctrl-W>** Kill (wipe) all text from the cursor to the mark.
**<ctrl-X><ctrl-X>**
Move the cursor from current position to the mark; reset the mark at the previous position of the cursor.
**<ctrl-Y>** Yank back the most recently killed text.
**<ctrl-]>***c* Move the cursor forward to next character *c*.
**<ctrl-?>** Delete the character under which the cursor is positioned. This command is identical with **<ctrl-D>**. Note that **<ctrl-?>** is the character that normally is output by the **<del>** key.
**<ctrl-[>** Begin an escape sequence. Note that **<ctrl-[>** is the character that normally is output by the **<esc>** key.
**<esc><ctrl-H>**
Delete the previous word (the word to the left of the cursor). A word is delineated by white space.
**<esc><del>** Delete the current word — that is, from the cursor to the end of the word as delineated by white space or the end of the line.
**<esc><space>**
Set the mark.
**<esc>.** Get the last (or *n*'th) word from previous line.
**<esc>?** Show possible completions. This feature is detailed below.
**<esc><** Move the cursor to the beginning of the history buffer.
**<esc>>** Move the cursor to the end of the history buffer.
**<esc>B** Move the cursor backwards (to the left) by one word.
**<esc>D** Delete the word under which the cursor is positioned.
**<esc>F** Move the cursor forward (to the right) by one word.
**<esc>L** Make the current word lower case.
**<esc>M** Toggle displaying eight-bit characters normally (meta-mode), or displaying them prefixed with the string **M-**. In the meta-mode, you can generate characters with the top bit set by pressing the **<alt>** key with an alphanumeric key; this is interpreted the same as **<esc><***key***>**.
**<esc>U** Make the current word upper case.
**<esc>Y** Yank back the most recently killed text.
**<esc>V** Show the version of the library **libedit.a**.
**<esc>W** Make yankable all text from the cursor to the mark.
**<esc>***n* Set the argument to integer *n*.
**<esc>***C* Read input from environment variable _*C*_, where *C* is an upper-case letter.

Most editing commands can be given an argument *n*, where *n* is an integer. To enter a numeric argument, type **<esc>**, the number, and then the command to execute. For example,

```
<esc> 4 <ctrl-F>
```

moves the cursor four characters forward.

Note that you can type an editing command on the line of input, not just at the beginning. Likewise, you can type (¢) to submit a line for processing, regardless of where on the line the cursor is positioned.

**readline()** has a modest macro facility. If you type **<esc>** followed by an upper-case letter, then **readline()** reads the contents of environment variable _*C*_ as if you had typed them at the keyboard.

## LEXICON

**readline()** also can complete a file name. For example, suppose that the root directory contains the following files:

```
coherent
coherent.old
```

If you type

```
rm /c
```

into **readline()** and then press the **<tab>** key, **readline()** completes as much of the name as it can — in this case, by adding **oherent**. Because the name is not unique, **readline()** then beeps. If you press **<esc>?**, **readline()** displays the two choices. If you then enter a tie-break character (in this case, '.'), followed by the **<tab>** character, **readline()** completes the file name for you.

### Using Line Editing

To include **readline()** in your program, simply call it as you do any other function. You must link the library **libedit.a** into your program.

### Example

The following brief example lets you enter a line and edit it, and then displays it.

```
#include <stdlib.h>

extern char *readline();
extern void add_history();

int main(ac, av)
int ac; char *av[];
{
      char *p;

      while ((p = readline ("Enter a line:")) != NULL) {
            (void) printf ("%s\n", p);
            add_history (p);
            free (p);
      }
      return 0;
}
```

### See Also

**add_history(), libedit**

### Notes

**readline()** calls **malloc()** to allocate space for the text that the user enters. Therefore, an application must call **free()** to free this space when it has finished with it.

**readline()** cannot handle lines longer than 80 characters.

The original manual page was written by David W. Sanderson <dws@ssec.wisc.edu>.

## *readonly* — Command

Mark a shell variable as read only
**readonly**

Mark each *variable* as a read-only shell variable. The shell will not permit subsequent assignments to a readonly variable. With no arguments, **readonly** prints the name and value of each read-only variable.

### See Also

**commands, ksh, sh**

## *readonly* — C Keyword

Storage class

**readonly** is a C keyword that modifies data declarations. As its name implies, the **readonly** modifier declares that data are to be read only; this helps protect key data against casual modification by the user or another programmer.

### See Also

**C keywords**

### Notes

The ANSI Standard eliminates this keyword.

## realloc() — General Function (libc)

Reallocate dynamic memory
**#include <stdlib.h>**
**char *realloc(***ptr***,** *size***)**
**char ****ptr***; unsigned** *size***;**

**realloc()** helps you manage a program's arena. It returns a block of *size* bytes that holds the contents of the old block, up to the smaller of the old and new sizes. **realloc()** tries to return the same block, truncated or extended; if *size* is smaller than the size of the old block, **realloc()** will return the same *ptr*.

If *ptr* is set to NULL, **realloc()** behaves like **malloc()**.

### Example

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

### See Also

**alloca(), arena, calloc(), free(), libc, malloc(), memok(), setbuf(), stdlib.h**
ANSI Standard, §7.10.3.4
POSIX Standard, §8.1

### Diagnostics

**realloc()** returns NULL if insufficient memory is available. It prints a message and calls **abort()** if it discovers that the arena has been corrupted, which most often occurs by storing past the bounds of an allocated block. **realloc()** behaves unpredictably if handed an incorrect *ptr*.

## reboot — Command

Reboot the COHERENT system
**/etc/reboot [ -p ]**

**reboot** reboots the COHERENT system. The option **-p** prompts the user if she really wishes to reboot before executing the reboot.

**reboot** can be run only by the superuser **root**.

The COHERENT system can be rebooted only from the console. *It should be rebooted only while in single-user mode. Failure to return to single-user mode before rebooting could damage the* COHERENT *file system and destroy data.*

### See Also

**commands, shutdown**

### Notes

No message is broadcast unless the command **shutdown** had been executed before invoking **reboot**.

## recursion — Definition

*Recursion* is the technique by which a program or function calls itself. Because under C, all variables in a function function have local scope; therefore, when a function calls itself, it in effect recreates itself but with a fresh copy of each of its variables. The "states" of the previous call or calls to that function are stored on the stack, and are not modified when the function calls itself.

Recursion is a useful way to loop through a complex procedure. Be careful, however, that you do not lose track of how the number of times you have called a given function; and be careful not to pile the stack too high, or you may have problems.

### Example

The following program demonstrates recursion. In it, the function **recur()** calls itself ten times.

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

```
main()
{
      printf("Before recursion ... \n");
      recur(1);
      printf("After recursion ... \n");
}

recur(level)
int level;
{
      printf("Entering call to recur() number %d\n", level);

      if (level < 10)
            recur(level+1);

      printf("Leaving call to recur() number %d\n", level);
}
```

### See Also

**programming COHERENT**

## *recv()* — Sockets Function (libsocket)

Receive a message from a connected socket
**#include <sys/types.h>**
**#include <sys/socket.h>**
**int recv(***socket***,** *buffer***,** *length***,** *flags***)**
**int** *socket***;**
**char** **buffer***;*
**int** *length*, *flags***;**

The function **recv()** receives messages from a connected socket.

*socket* is the socket from which the messages are received. It must have been created by the function **socket()**, and connected with the function **connect()**. *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.

*flags* ORs together either or both of the following flags:

**MSG_OOB**
Read any out-of-band data present on *socket*, rather than the regular, in-band data.

**MSG_PEEK**
"Peek" at the data present on the socket: the data are copied but not erased from the socket, so another call to **recv()** or **recvfrom()** retrieves the same data.

If all goes well, **recv()** returns the number of bytes it read from *socket*. If something went wrong, it returns -1 and sets **errno** to one of the following values:

**EAGAIN**
If no message is queued at *socket*, **recv()** normally waits for a message to arrive (which is a blocking operation). *socket*, however, is marked as non-blocking.

**EBADF** *socket* does not identify a valid socket.

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

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

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

### See Also

**connect(), libsocket, recvfrom(), send(), socket()**

## *recvfrom()* — Sockets Function (libsocket)

Receive a message from a socket
**#include <sys/compat.h>**
**#include <sys/socket.h>**
**#include "socketvar.h"**
**int recvfrom (***socket***,** *buffer***,** *length***,** *flags***,** *address***,** *addrlen***)**
**int** *socket***; char** **buffer***; **int** *length***; int** *flags***;**
**sockaddr_t** **from***; **int** **alen***;

**recvfrom()** receives messages from another socket. Unlike the related function **recv()**, **recvfrom()** receives data regardless of whether the socket is connected or not.

*socket* is the file descriptor of the socket from which data are to be received. It may or may not be connected. *buffer* is the chunk of memory in user space into which the data are to be written; it is *length* bytes long. If a received message is longer than *length* bytes, excess bytes can be discarded, depending on the type of socket from which the message is received. If *from* is not NULL, **recvfrom()** 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.

If no messages are waiting at *socket*, **recvfrom()** waits for a message to arrive, unless the socket is nonblocking. In this case, it returns -1 and sets **errno**, as described below.

*flags* ORs together either or both of the following flags:

**MSG_OOB**
Read any out-of-band data present on *socket*, rather than the regular, in-band data.

**MSG_PEEK**
"Peek" at the data present on *socket*. The data are returned but remain on *socket*; therefore, another call to **recvfrom()** or **recv()** retrieves the same data.

If all goes well, **recvfrom()** returns the number of bytes received. If an error occurs, it returns -1 and sets **errno** to one of the following values:

**EBADF** *socket* is an invalid descriptor.

**ENOTSOCK**
*socket* is the descriptor of a file, not a socket.

**EINTR** The operation was interrupted by delivery of a signal before any data was available to be received.

**EAGAIN**
*socket* is marked non-blocking, but the requested operation would block.

**ENOMEM**
Too little user memory is available to complete the operation.

### *See Also*

**libsocket, recv()**

### *Notes*

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

## *ref* — Command

Display a C function header
**ref** *function*

**ref** looks up the function header of *function* in any of a series of reference files built by the command **ctags**. It is used by the **elvis** editor's **<shift-K>** command. This command checks the file **refs** in the current directory.

### *See Also*

**commands, ctags, elvis**

### *Notes*

Release 1.7 of **ref** tells you which source file it is looking in. It does not show argument lines for macros, because it now knows that they do not have any.

**ref** is copyright © 1990 by Steve Kirkendall, and was written by Steve Kirkendall (kirkenda@cs.pdx.edu) assisted by numerous volunteers. It is freely redistributable, subject to the restrictions noted in included documentation. Source code for **ref** is available through the Mark Williams bulletin board, USENET, and numerous other outlets.

## *regcomp()* — Regular-Expression Function (libc)

Compile a regular expression into a structure
**#include <regexp.h>**
**regexp** ***regcomp(***expression**)**
**char** **expression**;**

Function **regcomp()** compiles *expression* into a structure of type **regexp**, and returns a pointer to it. For details on the structure **regexp**, see the Lexicon entry for **regexp.h**. *expression* must be a regular expression; the rules that define a regular expression are described in the Lexicon entry **regexp.h**.

### *See Also*

**libc, regexp.h**

### *Notes*

**regcomp()** calls **malloc()** to allocate the memory that holds the structure it creates. To free this structure, your program must call **free()**.

## *regerror()* — Regular-Expression Function (libc)

Return an error message from a regular-expression function
**#include <regexp.h>**
**void regerror(***message***)**
**char** **message*;**

Function **regerror()** is the function that is called by default when an error is detected in any of the regular-expression functions **regcomp()**, **regexec()**, or **regsub()**. It prints *message* onto the standard-error device, plus some text to indicate whence the message originates; then calls **exit()** to abort the program in which the error occurred.

You are not obliged to use **regerror()** to report an error with a regular-expression function. You can substitute another function of your choosing, should you prefer.

### *See Also*

**libc, regexp.h**

## *regexec()* — Regular-Expression Function (libc)

Compare a string with a regular expression
**#include <regexp.h>**
**int regexec(***expression***,** *string***)**
**regexp** **expression*;**
**char** **string*;**

Function **regexec()** compares *string* with *expression*, which is a regular expression compiled by function **regcomp()**. If *string* matches *expression*, **regexec()** returns one; otherwise, it returns zero and readjusts the sub-string pointers within *expression*.

For details on the structure **regexp** and its sub-string pointers, see the Lexicon entry for **regexp.h**. In brief, if **regexec()** successfully matches *string* with *expression*, it initializes arrays **startp[]** and **endp[]** within *expression*. Each **startp/endp** pair point to one substring within *string*: the **startp** element points to the first character of the substring and the **endp** element points to the first character that follows the substring. The pair **startp[0]/endp[0]** points to the substring of *string* that matched the whole of *expression*. The other pairs point to the substrings within *string* that matched parenthesized expressions within *expression*, with parenthesized expressions numbered in left-to-right order of their opening parentheses.

### *See Also*

**libc, regexp.h**

## regexp.h — Header File

Header file for regular-expression functions
**#include <regexp.h>**

Header file **<regexp.h>** is used with regular-expression function **regcomp()**, **regexec()**, and **regsub()**. These functions manipulate a regular expression, which is stored in structure **regexp**. **<regexp.h>** defines this structure as follows:

```
typedef struct regexp {
      char *startp[NSUBEXP];
      char *endp[NSUBEXP];
      char regstart;
      char reganch;
      char *regmust;
      int regmlen;
      char program[1];
} regexp;
```

Fields **regstart** through **program** are used internally, and should not be manipulated by a user's program. Fields **startp[]** and **endp[]** are arrays of pointers to sub-strings within the expression. For details on how these pointers are used, see the Lexicon entry for **regexec()**. **NSUBEXP** gives the number of sub-strings that can be addressed at one time; as of this writing, it is set to ten.

### Syntax of a Regular Expression

The following describes the rules with which the **regexp** functions define a regular expression.

A regular expression consists of zero or more *branches*. Branches are separated from each other by a pipe character '|'. A string matches an expression when it matches any branch within the expression.

A branch, in turn, consists of zero or more *pieces*, which are concatenated. Each piece is a string, or *atom*, which can be followed by '*', '+', or '?'. An atom followed by '*' can be matched with a sequence of zero or more matches of the atom. An atom followed by '+' can be matched with a sequence of one or more matches of the atom. An atom followed by '?' can be matched with either the atom or the null string.

An atom, in turn, is built from the following:

**(***expression***)**
A regular expression between parentheses This matches a match for the regular expression.

**[***string***]** Match any character within *string*. If string contains a hyphen '-', this represents a range of characters. For example, "0-9" represents all digits; or "a-z" represents all lower-case characters. To include a literal '-' within *string*, make it the first or last character within *string*. To include a literal ']' in the sequence, make it the first character, after a possible '^'.

**[^***string***]**
Match any character that is *not* in *string*.
a *range* (see below), '.'

**^** Match the null string at the beginning of the input string.

**$** Match the null string at the end of the input string.

*****c* Match the single character *c* literally; ignore any special significance that *c* might have.

### Ambiguity

A string can match more than one part of an regular expression. The following rules describe how to choose which part to match.

The basic rule is that if a regular expression could match two parts of a string, it matches the one that begins earlier.

If both parts begin in the same place but match different lengths of the expression, or match the same length in different ways, life gets messier, as follows.

In general, the possibilities in a list of branches are considered in left-to-right order, the possibilities for '*', '+', and '?' are considered longest-first, nested constructs are considered from the outermost in, and concatenated constructs are considered leftmost-first. The match that is chosen is the one that uses the earliest possibility in

the first choice that has to be made. If there is more than one choice, the next will be made in the same manner (earliest possibility) subject to the decision on the first choice.

For example, "(ab|a)b*c" could match "abc" in one of two ways. The first choice is between "ab" and 'a'; since "ab" is earlier, and lead to a successful overall match, it is chosen. Since the 'b' is already spoken for, the "b*" must match its last possibility — the empty string — because it must respect the earlier choice.

In the particular case where no '|'s are present and there is only one '*', '+', or '?', the net effect is that the longest possible match will be chosen. So "ab*", presented with "xabbbby", will match "abbbb". Note that if "ab*" is tried against "xabyabbbz", it will match "ab" just after 'x', due to the begins-earliest rule. In effect, the decision on where to start the match is the first choice to be made, hence subsequent choices must respect it even if this leads them to less-preferred alternatives.

### See Also

**header files, regcomp(), regerror(), regexec(), regsub()**

### Notes

The code used for the **regexp()** was written by Harry Spencer at the University of Toronto. It is copyright © 1986 by the University of Toronto. These routines are intended to be compatible with the Bell System-8 **regexp()** but are not derived from Bell code. The above description of regular expressions is derived from the manual page written by Harry Spencer.

## *register* — C Keyword

Storage class

**register** is a C keyword that declares a class of data storage. A variable so declared may be stored in a register, which may increase the speed with which it is read by a program.

### See Also

**auto, C keywords, extern, register variable, static**
ANSI Standard, §6.5.1

## *register variable* — Definition

**register** is a C storage class. A **register** declaration tells the compiler to try to keep the defined local data item in a machine register. Under COHERENT C, the **int foo** can be declared to be a register variable with the following statement:

```
register int foo;
```

The COHERENT C compiler makes three registers available for variables: **ESI**, **EDI**, and **EBX**. Subsequent **register** declarations are ignored, because no registers are left to hold them.

By definition of the C language, registers have no addresses, so you cannot pass the address of register variable as an argument to a function. For example, the following code will generate an error message when compiled:

```
register int i;
  . . .
dosomething(&i);  /* WRONG */
```

This rule applies whether or not the variable is actually kept in a register.

Placing heavily-used local variables into registers often improves performance, but in some cases declaring **register** variables can degrade performance somewhat.

### See Also

**auto, extern, Programming COHERENT, static, storage class**

## *regsub()* — Regular-Expression Function (libc)

Use regular expression to build a string
**"#include <regexp.h>"**
**regsub(***expression***,** *source***,** *dest***)**
**regexp** **expression***;*
**char** **source***,* **dest***;*

Function **regsub()** builds a string from a string and a regular expression.

*source* is the source string that is being interpreted. *expression* is the regular expression through which *source* is being interpreted; it must have been built by a call to **regcomp()**. Before you call **regsub()**, you must first have called **regexec()** to compare them and initialize the sub-string pointers within *expression*.

*dest* points to the memory into which **regsub()** writes its substituted string. It replaces each instance of '&' within *source* with the substring indicated by **startp[0]** and **endp[0]**. It also replaces each instance of '*n*', where *n* is a digit, with the substring **startp**[*n*] and **endp**[*n*].

For details on how these pointers are initialized, see the Lexicon entry for **regsub()**. For more details on the structure **regexp**, see the Lexicon entry for **regexp.h**. The rules that describe a regular expression also appear in function **regexp.h**.

### See Also

**libc, regexp.h**

## *remove()* — General Function (libc)

Remove a file
**#include <stdio.h>**
**int**
**remove(***filename***)**
**const char** **filename***;*

**remove()** breaks the link between between *filename* and the actual file that it represents. In effect, it removes a file. Thereafter, any attempt to use *filename* to open that file will fail. It is equivalent to the system call **unlink()**.

**remove()** will remove a file that is currently open. **remove()** returns zero if it could remove *filename*, and nonzero if it could not.

### Example

This example removes the file named on the command line.

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

main(argc,argv)
int argc, char *argv[])
{
      if(argc != 1) {
            fprintf(stderr, "usage: remove filename\n");
            exit(EXIT_FAILURE);
      }

      if(remove(argv[1])) {
            perror("remove failed");
            exit(EXIT_FAILURE);
      }
      return(EXIT_SUCCESS);
}
```

### See Also

**libc, unlink()**
ANSI Standard, §7.9.4.1
POSIX Standard, §8.1

## *rename()* — System Call (libc)

Rename a file
**#include <stdio.h>**
**int rename(***old***;** *new***)**
**char** **old***,* **new***;*

The COHERENT system call **rename()** changes the name of a file, from the name pointed to by *old* to that pointed to by *new*. Both *old* and *new* must point to a valid file name. If *new* names a file that already exists, the old file is replaced by the file being renamed.

**rename()** returns zero if it could rename *old*, and nonzero if it could not. If **rename()** could not rename *old*, its name remains unchanged.

### Example

This example renames the file named in the first command-line argument to the name given in the second argument.

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

main(argc, argv)
int argc; char *argv[];
{
      if (argc != 3) {
            fprintf(stderr, "usage: rename from to\n");
            exit(EXIT_FAILURE);
      }

      if(rename(argv[1], argv[2])) {
            perror("rename failed");
            exit(EXIT_FAILURE);
      }
      exit(EXIT_SUCCESS);
}
```

### See Also

**libc, link(), stdio.h, unlink()**
ANSI Standard, §7.9.4.2
POSIX Standard, §5.5.3

### Notes

The ANSI Standard states that **rename()** fails if *old* is open, or if its contents must be copied in order to rename it. Under COHERENT, it also fails if *new* is already open.

## *reprint* — Command

Reprint a spooled print job
**reprint [***job* [*page* **[***page***] ] ]**

The command **reprint** reprints each spooled *job*, where *job* identifies a job spooled with the command **lp**. If you do not specify a *job*, **reprint** reprints the job that you spooled most recently.

If you specify a *page*, **reprint** will attempt to reprint the document from that page to its end. If you specify two *page*s, **reprint** will attempt to reprint the document from the first *page* to the second.

Note that the printer daemon **lpsched** identifies pages by counting lines of input, so this feature works only with straight text. It does *not* work correctly with "cooked" input, such as files of PostScript or PCL.

### See Also

**commands, lp, printer**

### Notes

You should be *very* careful that jobs to print sensitive information — e.g., the payroll checks or your resume — do not linger in spool directory where other users can reprint them. For information on resetting a job's lifetime, see the Lexicon entries for **chreq**, **printer**, and **MLP_LIFE**. The article **controls** describes how to change the default life expectancies for spooled jobs.

## *resetterm()* — terminfo Function

Reset the terminal to its previous settings
**#include <curses.h>**
**resetterm()**

COHERENT comes with a set of functions that let you use **terminfo** descriptions to manipulate a terminal. **resetterm()** restores the terminal to the condition it was in when before the current program began to manipulate its settings. Your program should call **resetterm()** before it calls **system()** or **exit()**.

### See Also

**curses.h, fixterm(), terminfo**

## restor — Command

Restore file system
**restor** *command* **[***dump_device***] [***filesystem***] [***file ...***]**

**restor** copies to the hard disk one or more files from floppy disks or tapes written by the command **dump**.

**restor** recognizes the following *commands*:

**r** Mass restore both full and incremental dump disks/tapes into the *filesystem*. The target file system must have enough data blocks and i-nodes to hold the dump.

The mass restoration is performed in three phases. In phase 1, **restor** clears all i-nodes that were either clear at dump time or are going to be restored. Any allocated blocks are released. Second, it restores all files on the disk. The i-numbering is preserved; however, data blocks are allocated in the standard fashion. Third, a pass is made over the i-nodes and the list of free i-nodes in the superblock is updated.

Restoration begins immediately, using the currently mounted disk or tape.

**R** Like the **r** command, except that it pauses to ask for numbers of disks or reels.

**t** Read the header from the dump. Display the date the dump was written and the "dump since" date that produced the dump.

**x** Extract each *file* from the dump and restore it to the hard disk. All file names are absolute path names starting at the root of the dump (the first directory dumped, which is always the root directory of the file system). A numeric file name is taken to be an i-number on the dumped file system, permitting restore by i-number.

**restor** looks up each *argument* file in the directories of the dumped file system and prints out each name and associated i-number. **restor** extracts the files from the currently mounted dump disk or tape, and writes the extracted files into the current directory. Extracted files are named after their i-numbers.

**X** Like command x, except that before it begins, it asks you for the number of the disk (or the reel number of the dump tape). It continues asking for dump disks until all files have been extracted or you types **<ctrl-D>**.

Each of the above commands can be modified either or both of the following modifiers:

**f** Tell **restor** to take the next *argument* as the path name of the dump device (floppy-disk drive or tape drive). If the **f** modifier is not specified, **restor** uses the device **/dev/dump**.

**v** Verbose output. Tell **restor** to print a step-by-step trace of its actions when restoring an entire file system. This is for discovering what went wrong when a mass restore runs into trouble.

### Restoring from a Damaged Medium

As noted below, **dump** requires that its output be written to disks or tapes that are free of defects. Restoring a file system from a damaged medium is difficult and is not associated with a high probability of success; if, however, you must try to do so, the following directions will give you a fighting chance of restoring your data.

1. Use the command **fdformat** to format a blank disk. Use the command **badscan** to examine it for bad sectors; if it does have bad sectors, put it aside and try another.
2. Use the command **dd** to copy the bad disk to directory **/tmp***foo1* **dd** should die at the bad sector in the disk.
3. **dd** again to directory **/tmp/***foo2* using that command's **skip=***n* to skip past the bad sector (or sectors).
4. Repeat step 3 (if it died too) until the end of the disk is reached. Now you have a set of directories named **/tmp/***foo[1...n]* that contain parts of the bad disk.
5. Use the command

```
dd if=/tmp/foo1 of=/dev/fha0
```

with the new, defect-free disk.

6. Now, use the command

   ```
   dd if=/tmp/foo2 of=/dev/fha0 seek=whatever
   ```

   to place *foo2* into the right place on the new disk.

7. Repeat 6 for each directory *foo3* through *fooN*.

8. Create a 512-byte file that contain the string

   ```
   GARBAGE\n
   ```

   repeated 64 times. Use **dd** to copy it into new disk where the bad sectors were.

Now, you *should* have a disk that is a mirror image of the old, damaged dump disk. Each bad sectors will have been replaced by 64 iterations of the string **GARBAGE\n**. As noted, there is no guarantee that this scheme will work in every instance (the chances of error are quite high), but it will give you a fighting chance to save your data.

### Files

**/dev/dump** — Dump device
**/etc/ddate** — Dump date file

### See Also

**commands, dump, dumpdir**

### Diagnostics

Most of the diagnostics produced by **restor** are self-explanatory, and are caused by internal table overflows or I/O errors on the dump medium or file system.

If the dump spans multiple disks or reels, **restor** asks you to mount the next disk at the appropriate time. Type a newline when the disk has been mounted. **restor** verifies that this is the correct disk, and gives you another chance if the disk number in the dump header is incorrect.

### Notes

You cannot perform a mass restore onto a live root partition. Instead, boot a stand-alone version of COHERENT on a floppy-disk drive, or boot from an alternative COHERENT file system on another hard-disk partition before you attempt to do a mass restoration.

The handling of tapes with multiple dumps on them (created by dumping to the no rewind special files) is not very general. Basically, **restor** assumes that tapes holding multiple dumps and tapes holding dumps that span multiple reels are mutually exclusive. You can restore from any file on a reel by positioning the tape and then restoring with the **x** or **r** commands, which do not reposition the tape. It is (almost) impossible to use the **X** or **R** commands, as the position of the dump tape will be lost when **restor** closes it.

**dump** requires that its output be written to disks that are free of bad sectors. If you write a dump to a disk with bad sectors, you will not be able to restore files from that disk. See **dump** for directions on processing disks to ensure that they are free of bad sectors.

## *return* — C Keyword

Return a value and control to calling function

**return** is a C statement that returns a value from a function to the function that called it. **return** can be used without a value, to return control of the program to the calling function; also, the calling function is free to ignore the value **return** hands it. Note that it is good programming practice to declare functions that return nothing to be of type **void**.

A function can return only one value to the function that called it. Most often, this value is used to signal whether the function performed successfully or not.

### See Also

**C keywords**
ANSI Standard, §6.6.6.4

## rev — Command

Print text backwards
**rev [***file ...***]**

**rev** reverses the order of the characters in each line of each input *file* and writes the result to the standard output. If no *file* is specified, the standard input is used instead.

### Example

The following allows you to give a command like Mandrake the Magician. Typing

```
rev
Rocks break down wall!
<ctrl-D>
```

displays:

```
!llaw nwod kaerb skcoR
```

on your screen.

### See Also

**commands**

## rewind() — STDIO Function (libc)

Reset file pointer
**#include <stdio.h>**
**void rewind(***fp***)**
**FILE** **fp*;**

**rewind()** resets the file pointer to the beginning of stream *fp*. It is a synonym for **fseek(***fp***, 0L, 0**).

### Example

For an example of this routine, see the entry for **fscanf()**.

### See Also

**fseek(), ftell(), libc, lseek()**
ANSI Standard, §7.9.9.5
POSIX Standard, §8.1

### Notes

Release 4.2 of COHERENT has changed **rewind()** to conform to the ANSI Standard. Prior to release 4.2, **rewind()** returned EOF if an error occurs, and otherwise returned zero. **rewind()** now returns nothing. Programs that depend upon the return value of **rewind()** should be modified to conform to this change.

## rewinddir() — General Function (libc)

Rewind a directory stream
**#include <dirent.h>**
**void rewinddir(***dirp***)**
**DIR** **dirp*;**

The COHERENT function **rewinddir()** 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 the beginning of the directory.

**rewinddir()** discards all buffered data for its data stream. This ensures that your program knows about all modifications to the directory that occurred since the last time the directory stream was opened or rewound.

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

### See Also

**closedir(), dirent.h, getdents(), libc, opendir(), readdir(), seekdir(), telldir()**
POSIX Standard, §5.1.2

### Notes

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.

## *rindex()* — String Function (libc)

Find rightmost occurrence of a character in a string
**#include <string.h>**
**char *rindex(***string***, *c*) char ****string***; char *c*;**

**rindex()** scans *string* for the last occurrence of character *c*. If *c* is found, **rindex()** returns a pointer to it. If it is not found, **rindex()** returns NULL.

### Example

This example uses **rindex()** to help strip a sample file name of the path information.

```
#include <stdio.h>
#include <string.h>
#include <misc.h>
#define PATHSEP '/'           /* path name separator */

main()
{
      char *testpath = "/foo/bar/baz";
      printf("Before massaging: %s\n", testpath);
      printf("After massaging: %s\n", basename(testpath));
      return(EXIT_SUCCESS);
}

char *basename(path)
char *path;
{
      char *cp;
      return (((cp = rindex(path, PATHSEP)) == NULL)
            ? path : ++cp);
}
```

### See Also

**libc, strchr(), strrchr(), string.h**

### Notes

You *must* include header file **string.h** in any program that uses **rindex()**, or that program will not link correctly.

**rindex()** is now obsolete. You should use **strrchr()** instead.

## *rm* — Command

Remove files
**rm [ -firtv ]** *file ...*

**rm** removes each *file*. If no other links exist, **rm** frees the data blocks associated with the file.

To remove a file, a user must have write and execute permission on the directory in which the file resides, and must also have write permission on the file itself. The force option **-f** forces the file to be removed if the user does not have write permission on the file itself. It suppresses all error messages and prompts.

The interactive option **-i** tells **rm** to prompt for permission to delete each *file*.

The recursive removal option **-r** causes **rm** to descend into every directory, search for and delete files, and descend further into subdirectories. Directories are removed if the directory is empty, is not the current directory, and is not the root directory.

The test option **-t** performs all access testing but removes no files.

The verbose option **-v** tells **rm** to print each file rm and the action taken. In conjunction with the **-t** option, this allows the extent of possible damage to be previewed.

### See Also

**commands, ln, rmdir**

### Notes

Absence of delete permission in parent directories is reported with the message *file*: **permission denied**. Write protection is not inherited by subdirectories; they must be protected individually.

Note that unlike the similarly named command under MS-DOS, COHERENT's version of **rm** will *not* prompt you if you request a mass deletion. Thus, the command

```
rm *
```

will silently and immediately delete all files in the current directory. *Caveat utilitor!*

## rmail — Command

Receive mail from remote sites
**rmail [-LlRr] -q** *num* **-u** *uuxflags address ...*

Command **rmail** receives and processes mail from remote sites. It reads and interprets the address on the mail. If the mail is addressed to a user on your local system, it hands the mail to the local-mail deliverer **lmail** for delivery; if the mail is addressed to a remote system, it queues the mail for forwarding to that system.

It is very unusual for a user to invoke **rmail** from the command line. **rmail** usually is invoked by another program; in particular, the command **uuxqt** invokes it to process mail uploaded from another machine via UUCP.

### Options

The command **uux** can pass options to **rmail** to control its behavior. **rmail** recognizes the following command-line options:

**-L** Hand all mail that whose address includes a UUCP path to the local mailer **lmail** for processing, presumably to make use of other transport mechanisms (e.g., Ethernet). This option, and option **-l**, defers all routing until **lmail** has re-forwarded the mail to **smail** for further processing.

**-l** Hand all mail whose address includes a domain name to the local mailer **lmail** for processing, so they can be processed for non-UUCP domains.

**-q** *number*
Set the queuing threshold to *number*. When routing mail to a given host, **rmail** checks the "cost" of contacting the host; this cost set in file **/usr/lib/mail/paths**. If the cost is less the queueing threshold, then **rmail** sends the mail immediately; otherwise, it queues the mail for later shipment. Under COHERENT, default queueing threshold is 100.

**-R** Reroute UUCP paths, trying successively larger righthand substrings of a path until a component is recognized.

**-r** Route the first component of a UUCP path (**host!address**) in addition to routing domain addresses (**user@domain**).

**-u** *uuxflags*
Pass all *uuxflags* to the command **uux** for inclusion in the remote-mail command. This overrides any of the default values and other queueing strategies.

### Files

**/usr/lib/mail/aliases** — File from which aliases data base is built
**/usr/lib/mail/paths** — File from which paths data base is built
**/usr/spool/uucp/.Log/mail/mail** — Log of mail
**/bin/lmail** — Local mailer
**/bin/mail** — Mail user agent

### See Also

**aliases, commands, lmail, mail [overview], paths, smail**

### Notes

**rmail** is a link to command **smail**. For information on how **rmail** parses addresses and constructs headers, see the Lexicon entry for **smail**.

Because **rmail** is a link to **smail**, it actually recognizes all of **smail**'s command-line options; however, it ignores all except those listed above.

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**.

## *rmdir* — Command

Remove directories
**rmdir [ -f ]** *directory ...*

**rmdir** removes each *directory*. This will not be allowed if a *directory* is the current working directory or is not empty. The force option **-f** allows the superuser to override these restrictions. **rmdir** removes the '.' and '..' entries automatically. Note that using the **-f** option on a directory that is not empty will damage the file system, and require that it be fixed with **fsck**.

### *See Also*

**commands, mkdir, rm**

### *Notes*

**rmdir -f** does *not* remove files from a nonempty directory: it simply orphans them. To remove a nonempty directory and its contents, use **rm -r** instead.

## *rmdir()* — System Call (libc)

Remove a directory
**#include <unistd.h>**
**int rmdir(***path***)**
**char** **path***path***;**

The COHERENT system call **rmdir()** removes the directory specified by argument *path*. To remove the directory, the following conditions must apply:

- *path* must exist and be accessible, it must be empty (i.e., contain only entries for '.' and '..').
- You must have permission to remove the directory.
- The file system that contains *path* must not be mounted "read only".
- The directory must not be the current directory for any process.
- The directory must not be a mount point for another file system.

If the directory is successfully removed, **rmdir()** returns zero. If an error occurs, it returns -1 and sets **errno** to an appropriate value.

### *See Also*

**libc, mkdir, mkdir(), rmdir, unistd.h, unlink()**
POSIX Standard, §5.5.2

## *root* — Definition

**root** is the login name for the superuser.

### *See Also*

**superuser, Using COHERENT**

## *route* — Command

Show or reset a user's default printer
**route [***printer***]**

The command **route** shows or resets your default printer. When invoked without an argument, it displays your default printer, plus a list of available alternative printers. When invoked with the argument *printer*, it changes your default printer to that printer. *printer* must name a printer that has been described to the spooler by the command **lpadmin**.

Note that this feature is an extension to the version of **lp** that is included with UNIX System V.

### See Also

**commands, lp, lpadmin, printer**

### Notes

**route** is unique to the MLP implementation of the **lp** spooler. Scripts that use it will not be portable to other implementations of **lp**.

**route** is a link to **lpstat**.

## routers — System Administration

Rules for resolving mail addresses to remote systems
**/usr/lib/mail/routers**

File **/usr/lib/mail/routers** defines one or more *routers*. Each router defines a method by which **smail** routes mail to a remote system.

Each entry within **routers** names a router and sets its attributes. The order of entries is important, because **smail** invokes routers in the order in which they appear in this file. Each entry consists of the following information:

- The name of the router. This attribute begins the definition of a router. The name must be unique, it must appear flush with the left margin, and must be followed by a single colon ':'.
- The name of the *driver*, or program that implements the router. This can be a command that is part of **smail**'s suite of utilities (which are contained in directory **/usr/lib/mail**), or can be an ordinary COHERENT command. If the latter, then the full name of the command that implements the driver is given with a **cmd** attribute; this is shown below.
- A set of generic attributes for the router. These attributes are "generic" because they can come from a set that can be applied to any router.
- A set of driver-specific attributes. These can be applied only to routers that use this driver.

To extend an entry across multiple lines, begin successive lines with white space.

For example, the following entry gives the attributes for a director that reads aliases from a file named **/private/usr/lib/aliases**:

```
# read aliases from a file private to one machine on the network
private_aliases:
      driver=aliasfile, owner=owner-$user ;
      file=/private/usr/lib/aliases
```

This entry is named **private_aliases**. It depends upon the low-level director-driver routine named **aliasfile**, which is built into **smail**, and which implements a general mechanism for looking up aliases within a data base. By default, the driver **aliasfile** reads file **/usr/lib/mail/aliases** (which is simply a file that contains ASCII records in no particular order); this routers tells it instead to read file **/private/usr/lib/mail/aliases**. (For details on the format of an aliases file, see the Lexicon entry **aliases**). Finally, this router tells **smail** that if this director discovers an error while it is processing its input, then it (**smail**) should sends a mail message to an address formed by prefixing the string "owner-" onto the name of the alias.

### Attributes of a Router

The following gives the generic attributes can be used in router entry. Each attribute is followed by its type (Boolean or string). To set a string attribute, its name should be followed by an '=', then the value to which you are setting it. To set a Boolean attribute, prefix it with a '+'; to unset a Boolean attribute, prefix it with a '-'.

**always** (Boolean)

A router will not always find a complete match for a particular host name. For example, if a routing data base has a route to the domain **amdahl.com** but not to the host name **futatsu.uts.amdahl.com**, then the routing driver might return the route to **amdahl.com**.

In general, **smail** uses the route that matches the largest "chunk" of the target host. However, if you set the attribute **always**, then **smail** uses any match found by this router in preference to any route returned by any router that appears below it within **routers.**

This attribute is useful for hard-wiring a certain number of routes within a small data base. For example, this is useful for an Internet site that is the gateway for a small number of UUCP sites within the UUCP zone.

**driver** (string)
This attribute gives the set of low-level functions that do the work of routing remote mail. This attribute is required.

**method** (string)
**transport** (string)
A router driver can internally set the transport it uses to deliver mail to a remote site. If it does not do so, then you must set either a **method** or a **transport** attribute, to specify how the mail is to be delivered. The attribute **method** names the file whose contents relate host names to transports. The attribute **transport** specifies a particular transport that is defined in file **/usr/lib/mail/transports**. If the file named in a **method** attribute does not contain a match for all hosts, then **smail** uses the transport named with the **transport** attribute. The format of a method file is given in the next section.

### Method Files

Method files relates a set of host names with the set of transports to be used to deliver mail to those hosts. Each entry should have the form:

*hostname transport-name*

which states that **smail** should use *transport-name* to deliver mail to *hostname*. As a special case, if *hostname* is the special string '*', the entry matches any host. You should use this catch-all feature only in the last entry in a method file.

You can associate an entry in a method file with a particular grade of message. This lets you assign each grade of mail its own transport; for example, you may wish to use non-demand UUCP for messages with a "bulk" or "junk" precedence. To specify a range of grades, append the range of grade-letters to the host name, separated by '/'. Entries with grades can be in any of the forms:

*hostname/X transport-name*
*hostname/X-* transport-name*
*hostname/*-Y transport-name*
*hostname/X-Y transport-name*

For a discussion of grade letters and their correlation with message-precedence strings, see the description of attribute **grades** in the Lexicon entry for **config (smail).** In the first form, the transport is used for an exact match of the grade letter. In the second form, a match requires a grade a character value of at least *X*. In the third, form a match requires a grade character value of at most *Y*. The final form specifies a range of grades from character value *X* to character value *Y*.

### The Default Configuration

The following gives the routers defined in the default version of file **/usr/lib/mail/routers** that is included with COHERENT.

The first router is named **paths**. It processes the contents of file **/usr/lib/mail/paths**:

```
# paths - route using a paths file, like that produced by the pathalias program
paths:      driver=pathalias,    # general-use paths router
      transport=uux;       # for matches, deliver over UUCP

      file=paths,          # sorted file containing path info
      proto=dbm,           # use a DBM-style data base
      optional,            # ignore if the file does not exist
      -required,           # no required domains
      domain=uucp,         # strip ending ".uucp" before searching
```

The command **pathalias**, which this router uses to read file **paths**, is described in its own Lexicon entry; as is command **uux**, which this router invokes to transport the files to the remote site.

The next router, named **uucp_neighbors**, matches nearby systems that are accessible via UUCP:

```
# uucp_neighbors - match neighbors accessible over UUCP
uucp_neighbors:
      driver=uuname,          # use a program which returns neighbors
      transport=uux;

      cmd=/usr/bin/uuname,    # specifically, use the uuname program
      domain=uucp,            # strip ending ".uucp" before searching
```

Command **uuname** is part of the Taylor UUCP package that is included with COHERENT. It is described in its own Lexicon entry. Under COHERENT, this command always returns the name of your local host.

The final router describes how to route mail to the "smart host." This is a system that knows how to access more remote systems than your system does, and that you trust to handle mail correctly. **smail** forwards to the smart host all mail that it does not know how to route, in the hope that the smart host will know what to do with it.

```
# smart_host - a partically specified smarthost director
#
# If the config file attribute smart_path is defined as a path from the
# local host to a remote host, then host names not matched otherwise will
# be sent off to the stated remote host.  The config file attribute
# smart_transport can be used to specify a different transport.
#
# If the smart_path attribute is not defined, this router is ignored.
smart_host:
      driver=smarthost, # special-case driver
      transport=uux;          # by default deliver over UUCP

      -path,                  # use smart_path config file variable?
```

### See Also

**Administering COHERENT, config [smail], directors, mail [overview], smail, transports**

### Notes

For information on how the configuration files **directors**, **routers**, and **transports** relate to each other, see the Lexicon entry for **directors**.

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**.

## *rpow()* — Multiple-Precision Mathematics (libmp)

Raise multiple-precision integer to power
**#include <mprec.h>**
**void rpow(***a, b, c***)**
**mint** **a, *b, *c**;**

**rpow()** sets the multiple-precision integer (or **mint**) pointed to by *c* to the value pointed to by *a* raised to power of the value pointed to by *b*.

### See Also

**libmp**

## *RS-232* — Technical Information

Serial port wiring

This article details the connections (pinouts) of EIA standard RS-232C. This connector consists of a D-shaped plug with 25 pins in two rows: 13 pins in the upper row and 12 in the lower. This interface is commonly used by devices that require a serial interface to a computer; these devices include modems, terminals, serial printers, and such specialized devices as bar-code scanners. In addition, this articles gives the pinouts of the nine-pin DB-9P connector, which is a nine-pin version of the RS-232 that is commonly used in AT and AT-compatible computers.

### RS-232 Pinout

The following table gives the 25-pin EIA standard RS-232C pinouts. It also gives:

- Nine-pin DB-9P convention
- Common abbreviations of signal names
- Abbreviations of RS-232 signal names
- Equivalent CCITT signal-number designations
- Signal direction (as appropriate)
- Signal description

Please note that in this table, **DTE** stands for "data terminal equipment" and refers to terminal-type equipment such as a PC or a terminal, whereas **DCE** stands for "data communications equipment" and refers to modems and modem-type equipment.

| *DB-25 Pin #* | *DB-9 Pin #* | *Common Name* | *EIA* | *CCITT* | *DTE-DCE* | *Description* |
|---|---|---|---|---|---|---|
| 1 | | FG | AA | 101 | — | Frame ground |
| 2 | 3 | TD | BA | 103 | → | Transmitted data |
| 3 | 2 | RD | BB | 104 | ← | Received data |
| 4 | 7 | RTS | CA | 105 | → | Request to send |
| 5 | 8 | CTS | CB | 106 | ← | Clear to send |
| 6 | 6 | DSR | CC | 107 | ← | Data set ready |
| 7 | 5 | SG | AB | 102 | — | Signal ground |
| 8 | 1 | DCD | CF | 109 | ← | Data carrier detect |
| 9 | | — | — | — | — | Positive DC test voltage |
| 10 | | — | — | — | — | Negative DC test voltage |
| 11 | | QM | — | — | ← | Equalizer mode |
| 12 | | SDCD | SCF | 122 | ← | Secondary carrier detect |
| 13 | | SCTS | SCB | 121 | ← | Secondary clear to send |
| 14 | | STD | SBA | 118 | → | Secondary transmitted data |
| 15 | | TC | DB | 114 | ← | Transmitter clock |
| 16 | | SRD | SBB | 119 | ← | Secondary receiver clock |
| 17 | | RC | DD | 115 | → | Receiver clock |
| 18 | | DCR | — | — | ← | Divided clock receiver |
| 19 | | SRTS | SCA | 120 | → | Secondary request to send |
| 20 | 4 | DTR | CD | 108.2 | → | Data terminal ready |
| 21 | | SQ | CG | 110 | ← | Signal quality |
| 22 | 9 | RI | CE | 125 | ← | Ring indicator |
| 23 | | — | CH | 111 | → | Data rate selector |
| 24 | | TC | DA | 113 | ← | Transmitted clock |
| 25 | | | | | | |

### Files

**/usr/pub/rs232** — On-line version of above table

### See Also

**Administering COHERENT, asy, modem, terminal**

Seyer, M.D.: *RS-232 Made Easy: Connecting Computers, Printers, Terminals, and Modems*. Englewood Cliffs, NJ, Prentice-Hall Inc., 1984.

### Notes

Serial ports on the back of the PC use either a 25-pin male (DB-25P) or a nine-pin male (DB-9P) connector. Due to what can only be regarded as extreme stupidity, the 25-pin female (DB-25S) connector was chosen for the parallel (printer) port, rather than using the usual 36-pin parallel connector. Do not confuse these ports when wiring custom-cable assemblies, as you can damage your equipment!

## rsmtp — Command

Run batched SMTP mail

**/bin/rsmtp**

Command **rsmpt** reads and executes Simple Mail Transfer Protocol (SMTP) commands from the standard input. It normally is used to execute a batched form of SMTP between machines via a remote execution service, e.g., UUCP. **rsmtp** reports failures through return mail.

### See Also

**commands, mail [overview], smail**

### Notes

**rsmtp** is a link to **smail**.

## *rubik* — Command

Play Rubik's cube
**/usr/games/rubik**

The command **rubik** lets you fiddle with an electronic version of Rubik's cube. By issuing commands, you can "rotate" the segments of the virtual cube and, with some agony, align all the "colors".

**rubik** is written in **m4**, and is a good example of extended programming in this utility.

### See Also

**commands, m4**

## *runq* — Command

Periodically process the mail queue
**/bin/runq**

Command **runq** checks the spool directory that holds incoming mail, and processes it. It is equivalent to the command **smail -q**.

### See Also

**commands, mail [overview], smail**

### Notes

**runq** is a link to **smail**.

## *rvalue* — Definition

An **rvalue** is the value of an expression. The name comes from the assignment expression **e1=e2;**, in which the right operand is an rvalue.

Unlike an lvalue, an rvalue can be either a variable or a constant.

### See Also

**lvalue, Programming COHERENT**
ANSI Standard, §6.2.2.1