fabs() — Mathematics Function (libm)

Compute absolute value
#include <math.h>
double fabs(*z***) double** *z***;**

fabs() implements the absolute value function. It returns *z* if *z* is zero or positive, or -*z* if *z* is negative.

Example

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

See Also

abs(), ceil(), floor(), frexp(), libm
ANSI Standard, §7.5.6.2
POSIX Standard, §8.1

factor — Command

Factor a number
factor [*number ...* **]**

factor computes and prints the prime factorials for each of a list of given *number*s. If no *number*s are given on the command line, **factor** reads numbers from the standard input.

See Also

commands

false — Command

Unconditional failure
false

false does nothing. It is guaranteed to fail. It can be useful in shell scripts, to force certain situations to occur.

See Also

commands, ksh, sh, true

Notes

Under the Korn shell, **false** is an alias for its built-in command **let**.

fc — Command

Edit and re-execute one or more previous commands
fc [-e *editor***] [-ln] [***first* **[***last***]]**

fc, the "fix command", is a command built into the Korn shell **ksh**. It permits you to edit and re-execute one or more commands that have been executed previously.

fc selects commands *first* through *last* and inserts them into a text editor. You can edit the commands in the editor; exiting from the editor re-executes the commands. *first* and *last* can be addressed either by the command's number (the first command issued to the shell is number one, the second is number two, and so on), or by a string that matches the beginning of the command. When called without a *last* variable, the command selects just *first*. Option **-l** prints the commands on the standard output rather than buffering the commands for editing and re-execution. Option **-n** suppresses the default command numbers.

fc uses the editor named in the environmental variable **FCEDIT**; if this variable is not set, it uses MicroEMACS. The option **-e** lets you select another editor.

See Also

commands, FCEDIT, ksh

FCEDIT — Environmental Variable

Editor used by fc command

The Korn shell's command **fc** reads the environmental variable **FCEDIT** to see which editor it should use to edit commands.

See Also

environmental variables, ksh

fclose() — STDIO Function (libc)

Close a stream
#include <stdio.h>
int fclose(*fp***) FILE ****fp***;**

fclose() closes the stream **fp**. It calls **fflush()** on the given *fp*, closes the associated file, and releases any allocated buffer. The function **exit()** calls **fclose()** for open streams.

Example

For examples of how to use this function, see the entries for **fopen()** and **fseek()**.

See Also

libc
ANSI Standard, §7.9.5.1
POSIX Standard, §8.1

Diagnostics

fclose() returns **EOF** if an error occurs.

fcntl() — System Call (libc)

Control open files
#include <fcntl.h>
int fcntl(*fd***,** *command***,** *arg***)**
int *fd***,** *cmd***,** *arg***;**

The COHERENT system call **fcntl()** manipulates an open file.

fd is the file descriptor; this description must have been obtained from a call to **creat()**, **dup()**, **fcntl()**, **open()**, or **pipe()**.

command identifies the task that you want **fcntl()** to perform. The value **fcntl()** returns varies, depending on what command you ask it to perform. *arg* is an argument specific to the given *command*.

fcntl() commands **F_GETLK**, **F_SETLK**, and **F_SETLKW** (described in detail below) implement file-record locking. File-record locks use the **flock** structure, which is defined in header file **<fcntl.h>** as follows:

```
typedef struct flock {
	short		l_type;			/* F_RDLCK, F_WRLCK, or F_UNLCK*/
	short		l_whence;		/* SEEK_SET, SEEK_CUR, SEEK_END */
	long		l_start;		/* location */
	long		l_len;			/* 0 is through EOF */
	short		l_sysid;		/* system id of lock (for GETLK) */
	short		l_pid;			/* process id of owner (for GETLK) */
};
```

You can lock a section of a file for reading (excluding subsequent write locks) or for writing (excluding all subsequent locks). The locked section begins at the specified location **l_start** and can extend backwards (when **l_len** is negative) or forwards (when it is positive). If **l_len** is zero, the lock extends to the end of the file. A lock may extend past the current end of file, but may not extend to before the beginning of the file.

fcntl() Commands

fcntl() recognizes the following commands:

F_DUPFD Duplicate file descriptor *fd* onto the first available file descriptor greater than or equal to *arg*. **fcntl()** returns the new file descriptor.

F_GETFD Get the current value of the close-on-exec flag **FD_CLOEXEC** for the file. If the low-order bit of the return value of **fcntl()** is zero, the file descriptor remains open if the process uses **exec()** to execute another process, If the low-order bit of the return value is one, the file descriptor is closed upon **exec()**.

F_GETFL Get the file flags for the file specified by *fd*. With this option, **fcntl()** returns the file flags.

F_GETLK *arg* must point to a **struct flock** that describes a section of the file to lock. If the system does not have any locks on the specified section, **fcntl()** sets the lock type of *arg* to **F_UNLCK** and leaves the other members unchanged. Otherwise, it sets the contents of *arg* to the first existing lock that blocks the requested lock.

F_SETFD Set the close-on-exec flag of the file to the value of the low bit of *arg*.

F_SETFL Set file flags for file descriptor *fd* to the value specified by *arg*. Here, **fcntl()** returns the new file flags.

F_SETLK Set or clear a file-record lock. *arg* must point to a **struct flock**. Set member **l_type** to **F_RDLCK** to request a read lock, to **F_WRLCK** to request a write lock, or to **F_UNLCK** to unlock a previously locked section. If the requested lock cannot be set, **fcntl()** returns with an error value of -1 and sets **errno** to **EACCES**.

F_SETLKW is just like **F_SETLK** unless the requested lock is not available, in which case **F_SETLKW** causes the current process to sleep until the requested lock becomes available. If sleeping would cause a deadlock, **fcntl()** returns -1 and sets **errno** to **EDEADLK**.

Upon failure, each *cmd* returns -1 and sets **errno** to an appropriate value. Possible **errno** values include the following:

EAGAIN Section already locked.

EBADF Bad file desciptor.

EINVAL Invalid command.

EMFILE Too many files open.

ENOLCK No more locks available.

EDEADLK Deadlock would result.

See Also

close(), creat(), dup(), exec(), fcntl.h, file, file descriptor, libc, lockf(), open(), pipe()
POSIX Standard §6.5.2

Notes

Use **fcntl()** with the unbuffered I/O routines (**open()**, **write()**, and so on) rather than with standard I/O library routines (**fopen()**, **fprintf()**, **fwrite()**, and so on). The buffering used by the standard I/O library may cause unexpected behavior with file locking.

fcntl.h — Header File

Manifest constants for file-handling functions
#include <fcntl.h>

fcntl.h declares manifest constants that are used by the file-handling functions **open()**, **creat()**, and **fcntl()**.

See Also

header files
POSIX Standard, §6.5.1

fcvt() — General Function (libc)

Convert floating-point numbers to strings
char *
fcvt(*d*, *w*, **dp*, **signp***)**
double *d*; **int** *w*, **dp*, **signp*;

fcvt() converts floating point numbers to ASCII strings. Its operation resembles that of **printf()**'s operator **%f**.

fcvt() converts *d* into a NUL-terminated string of decimal digits. The argument *w* sets the precision of the string, i.e., the number of characters to the right of the decimal point.

fcvt() rounds the last digit and returns a pointer to the result. On return, **fcvt()** sets *dp* to point to an integer that indicates the location of the decimal point relative to the beginning of the string: to the right if positive, and to the left if negative. Finally, it sets *signp* to point to an integer that indicates the sign of *d*: zero if positive, and nonzero if negative. **fcvt()** rounds the result to the FORTRAN F-format.

Example

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

See Also

libc

Notes

fcvt() performs conversions within static string buffers that it overwrites on each execution.

fd — Device Driver

Floppy disk driver

The files **/dev/f*** and **/dev/rf*** are entries for the floppy-disk driver **fd**. Each entry is assigned major device number 4, is accessed as a block-special device, and has a corresponding character-special device entry. **fd** handles up to four 5.25-inch floppy-disk drives, each in one of several formats.

The least-significant four bits of an entry's minor device number identify the type of drive. The next least-significant two bits identify the drive.

The following table summarizes the name, minor device number, sectors per track, partition sector size, characteristics, and addressing method for each device entry of floppy-disk drive 0.

9 sectors/track					
fqa0	13	9	1440	DSQD	cylinder (3.25 inch — 720K)
f9a0	12	9	720	DSDD	cylinder (5.25 inch — 360K)
15 sectors/track					
fha0	14	15	2400	DSHD	cylinder (5.25 inch — 1.2MB)
18 sectors/track					
fva0	15	18	2880	DSHD	cylinder (3.5 inch — 1.44MB

Prefixing an **r** to a device name given above gives the name of the corresponding character-device entry. Corresponding device entries for drives 1, 2, and 3 have minor numbers with offsets of 16, 32, and 48 from the minor numbers given above, and have 1, 2, or 3 in place of 0 in the names given above.

For device entries whose minor number's fourth least-significant bit is zero (minor numbers 0 through 7 for drive 0), the driver uses surface addressing rather than cylinder addressing. This means that it increments tracks before heads when computing sector addresses and the first surface is used completely before the second surface is accessed. For devices whose minor number's fourth least significant bit is 1 (minor numbers 8 through 15 for drive 0), the driver uses cylinder addressing.

For a floppy disk to be accessible from the COHERENT system, a device file must be present in directory **/dev** with the appropriate type, major and minor device numbers, and permissions. The command **mknod** creates a special file for a device.

The following table gives the all floppy-disk devices that COHERENT recognizes, by minor number. Note that some specialized devices skip the first cylinder on the disk, to support some third-party program that requires this feature:

Minor Number	*Drive*	*Diameter*	*Density*	*Cylinders*
0	0	Both	Any	1-39/79
1	0	Both	Any	0-39/79
4	0	5.25”	360KB	1-39
5	0	3.5”	720KB	1-79
6	0	5.25”	1.2MB	1-79
7	0	3.5”	1.44MB	1-79
12	0	5.25”	360KB	0-39
13	0	3.5”	720KB	0-79
14	0	5.25”	1.2MB	0-79
15	0	3.5”	1.44MB	0-79
16	1	Both	Any	1-39/79
17	1	Both	Any	0-39/79
20	1	5.25”	360KB	1-39
21	1	3.5”	720KB	1-79
22	1	5.25”	1.2MB	1-79
23	1	3.5”	1.44MB	1-79
28	1	5.25”	360KB	0-39
29	1	3.5”	720KB	0-79
30	1	5.25”	1.2MB	0-79
31	1	3.5”	1.44MB	0-79

Example

The following program examines a COHERENT floppy-disk device and prints its size in bytes. It was written by Sanjay Lal (sanjayl@tor.comm.mot.com):

```
#include <errno.h>
#include <fcntl.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/stat.h>

#define BLOCK 512

struct FDATA {
      int fd_size; /* Blocks per diskette */
      int fd_nhds; /* Heads per drive */
      int fd_trks; /* Tracks per side */
      int fd_offs; /* Sector base */
      int fd_nspt; /* Sectors per track */
      char fd_GPL[4]; /* Controller gap param (indexed by rate) */
      char fd_N; /* Controller size param */
      char fd_FGPL; /* Format gap length */
};

/* Parameters for each kind of format */
struct FDATA fdata [] = {
/* 8 sectors per track, surface by surface seek. */
      {  320, 1, 40, 0, 8, { 0x00, 0x20, 0x20 }, 2, 0x58 }, /* Single sided */
      {  640, 2, 40, 0, 8, { 0x00, 0x20, 0x20 }, 2, 0x58 }, /* Double sided */
      { 1280, 2, 80, 0, 8, { 0x00, 0x20, 0x20 }, 2, 0x58 }, /* Quad density */

/* 9 sectors per track, surface by surface seek. */
      {  360, 1, 40, 0, 9, { 0x00, 0x20, 0x20 }, 2, 0x50 }, /* Single sided */
      {  720, 2, 40, 0, 9, { 0x00, 0x20, 0x20 }, 2, 0x50 }, /* Double sided */
      { 1440, 2, 80, 0, 9, { 0x00, 0x20, 0x20 }, 2, 0x50 }, /* Quad density */

/* 15 sectors per track, surface by surface seek. */
      { 2400, 2, 80, 0, 15, { 0x1B, 0x00, 0x00 }, 2, 0x54 }, /* High capacity */

/* 18 sectors per track, surface by surface seek. */
      { 2880, 2, 80, 0, 18, { 0x1B, 0x00, 0x00 }, 2, 0x6C }/* 1.44 3.5" */
};
```

```
#define funit(x) (minor(x) >> 4)   /* Unit/drive number */
#define fkind(x) (minor(x) & 0x7)  /* Kind of format */

static int ctrl;

int main(argc, argv)
int argc; char **argv;
{
      int size;
      struct stat sbuf;
      struct FDATA *fdp;

      if (argc!=2) {
            fprintf(stderr, "usage : %s /dev/fd...\n",argv[0]);
            exit(EXIT_FAILURE);
      }

      if (strcmp(argv[1], "conv")==0) {
            /*special case*/
            size = getchar() + getchar() * 256;
            printf("%ld\n", (long)((long)size * (long)512) );
            return (EXIT_SUCCESS);
      }

      if (ctrl = stat(argv[1], &sbuf)) {
            fprintf (stderr,"%s : error stating %s.\n", argv[0], argv[1]);
            exit(EXIT_FAILURE);
      }

      fdp = & fdata [fkind (sbuf.st_rdev)];
      printf("%ld\n", (long)((long)fdp->fd_size * (long)512) );

      return (EXIT_SUCCESS);
}
```

Files

<fdioctl.h> — Driver command header file
/dev/fd* — Block-special files
/dev/rfd* — Character special files

See Also

device drivers, fdformat, floppy disk, ft, mkfs, mknod

Diagnostics

The driver reports any error status received from the controller and retries the operation several times before it reports an error to the program that initiated an operation.

Notes

The floppy-tape driver **ft** also works through major-device number 4.

fd assumes that the disk is formatted with eight, nine, 15, or 18 sectors of 512 bytes each per track, depending upon the **/dev** entry. Cylinder addressing is the norm for COHERENT.

Programs that use the raw device interface must read whole sectors into buffers that do not straddle DMA boundaries.

fd.h — Header File

Declare file-descriptor structure
#include <sys/fd.h>

fd.h declares the file-descriptor structure **fd**, plus associated constants.

See Also

header files

fdformat — Command

Low-level format a floppy disk
/etc/fdformat [*option ...* **]** *special*

fdformat formats a floppy disk. The given *special* should be the name of the special file that correspond to the floppy disk drive.

fdformat recognizes the following options:

-a Print information on the standard output device during format. As it formats a cylinder, it will print a line of the form

```
hd=0 cyl=25
```

on your screen.

-i *number*
Use *number* (0 through 7) as the interleave factor in formatting. Note that the default interleave is six.

-o *number*
Use *number* (default, 0) as the skew factor for sector numbering.

-v Verify formatting and verify data written with the **-w** option.

-w *file* Format the floppy disk and then copy *file* to it track by track. The raw device should be used.

The command **mkfs** builds a COHERENT file system on a formatted floppy disk. The command **dosformat** builds a DOS file system on a formatted floppy disk. The command **mount** mounts a floppy disk containing a file system to allow access to it through the COHERENT directory structure. The command **umount** unmounts a floppy disk.

Examples

The following command formats a 2880-block (1.44-megabyte), 3.5-inch floppy disk in drive 1 (otherwise known as drive B):

```
/etc/fdformat -v /dev/rfva1
```

The following command formats a 2400-block (1.2-megabyte), 5.25-inch floppy disk in drive 0 (otherwise known known as drive A):

```
/etc/fdformat -v /dev/rfha0
```

Note that using the raw device (**/dev/rfha0**) speeds up formatting noticeably.

See Also

commands, dosformat, fd, mkfs, mount, umount

Diagnostics

When errors occur on floppy-disk devices the driver prints on the system console an error message that describes the error.

Notes

fdformat formats a track at a time. **fdformat** can be interrupted between tracks, which may result in a partially formatted floppy disk.

fdioctl.h — Header File

Control floppy-disk I/O
#include <sys/fdioctl.h>

fdioctl.h declares constants and structures used to control floppy-disk I/O.

See Also

header files

fdisk — Command

Hard-disk partitioning utility
/etc/fdisk [-r] [-c] [-b *mboot***]** *xdev ...*

The command **fdisk** lets you view how a hard disk is partitioned, alter how it is partitioned, and mark a partition so that the COHERENT bootstrap will automatically boot the operating system it contains. If you wish, you can use **fdisk** to assign partitions to different operating systems, e.g., MS-DOS, CP/M, Windows NT, COHERENT, and XENIX.

fdisk recognizes the following command-line options:

-b Use the first 446 bytes of the file *mboot* to replace the bootstrap information in *xdev*. Use this option to overwrite the COHERENT bootstrap with another bootstrap.

-c Specify the disk geometry (i.e., number of cylinders, heads, sectors) for disk drives that your system's BIOS does not support.

-r Read-only access. **fdisk** reads the partition table and displays its contents, but does not let you change how a disk is partitioned. This is the "safe" option.

-V Display the version number of **fdisk**. PP When you invoke **fdisk**, it reads the first block from the special device *xdev*, which holds the partitioning information for that disk. *xdev* is the device whose name ends in **x**; for example, if you have one SCSI hard disk and one AT-style hard disk installed in your machine, *xdev* would be either **/dev/sd0x** or **/dev/at0x**. If you use **fdisk** with a device other than the **x** device (e.g., with device **/dev/at0a**), **fdisk** displays values for your partitions that are totally bogus — and probably quite alarming.

After you invoke **fdisk**, it displays a warning message, then the layout of the disk whose partition-table device you named on the command line. The following gives an example layout, for a 33-megabyte AT disk:

```
Drive 0 Currently has the following logical partitions:
              [In Cylinders] [  In Tracks  ]
Number  Type  Start End Size Start  End Size Mbyte Blocks Name
0 Boot MS-DOS    0  149  150     0  899  900  7.83  15300 /dev/at0a
1     EXT-DOS  150  614  464   900 3684 2784 24.28  47430 /dev/at0b
2      UNUSED    0    0    0     0    0    0     0      0 /dev/at0c
3      UNUSED    0    0    0     0    0    0     0      0 /dev/at0d
```

In this example, partition 1 (which is accessed via device **/dev/at0a**) holds an MS-DOS file system. It is marked as the "Boot" partition, which means that the COHERENT bootstrap will boot its operating system automatically when you reboot your computer. The other columns show the size of each partition, and its beginning and end points in both cylinders and tracks.

If you invoked **fdisk** with its option **-r**, the program exits at this point. If you did *not* invoke it with option **-r**, it displays the following menu of actions:

```
Possible actions:
   0 = Quit
   1 = Change active partition (or make no partition active)
   2 = Change one logical partition
   3 = Change all logical partitions
   4 = Delete one logical partition
   5 = Change drive characteristics
   6 = Display drive information
   7 = Proceed to next drive
```

The following describes each action in detail:

0. Quit **fdisk**.

1. Change which partition is the active partition. You can also say that your system has *no* active partition. If you do so, the COHERENT bootstrap will prompt you at boot time to enter the number of the partition whose operating system you wish to boot. **fdisk** will let you set only one active partition at a time.

2. Change the dimensions (i.e., the size, beginning point, or end point) of one partition. Doing this destroys the data on that partition.

3. Change the dimensions of every partition. Doing this destroys the data on your hard disk.

4. Delete a partition.

5. Change the parameters of the drive. Use this option if COHERENT somehow has a faulty notion of your disk's size. You should never have to use this option; using it will wipe out all data on your hard disk.

6. Give summary information about the disk — that is, re-display the table shown above.

7. This option appears only if you have more than one hard disk drive. Use this option to display information about another hard disk on your system.

Before you change the dimensions of any partition on your system, read the warnings given in the notes below. When you have finished modifying your disk, **fdisk** then writes your changes into *xdev*.

Files

<fdisk.h>

See Also

commands, hard disk, ideinfo

Notes

If you change a device's partition table, reboot your system. Most device drivers will not recognize the revised partition information until a reboot occurs.

As the **-r** and **-b** options are contradictory, attempting to use them together triggers an error message.

Note that many operating systems implement a program named **fdisk**. Each manipulates a hard disk's partition table, but not all respect the fact that a disk may hold more than one operating system. In particular, the MS-DOS edition of **fdisk** can rearrange the order of entries in the partition table. If this happens, you may lose the ability to run COHERENT until the table is restored to its previous order. A sign of this problem is seeing the prompt **AT boot?** when you try to start COHERENT after running any **fdisk** program, and not being able to get past it.

Computer systems that use older releases of a BIOS may report incorrect disk parameters. Users of such systems should change the CMOS setup values if possible, but the BIOS on some older systems will not allow you to specify arbitrary values for disk parameters. Users with such systems can use the option **fdisk -c** option instead.

If you plan to install and run COHERENT and MS-DOS on the same hard disk, note the following:

- If you wish to install COHERENT and MS-DOS on the same hard drive, you *must* run the MS-DOS **fdisk** first!
- If you plan on running both operating systems, you *must* install MS-DOS first and leave some free cylinders on the disk for COHERENT as well as a free partition. You can have both primary as well as extended MS-DOS partitions on the same drive as COHERENT, but COHERENT cannot use a sub-partition of the MS-DOS extended partition. COHERENT must have one of the four *real* partitions. Failure to observe these rules will result in loss of data! *Caveat utilitor.*

fdisk.h — Header File

Fixed-disk constants and structures
#include <sys/fdisk.h>

fdisk.h declares structures and constants used to manipulate a fixed (hard) disk.

See Also

header files

fdopen() — STDIO Function (libc)

Open a stream for standard I/O
#include <stdio.h>
FILE *fdopen(*fd, type***) int** *fd***; char ****type***;**

fdopen() allocates and returns a **FILE** structure, or *stream*, for the file descriptor *fd*, as obtained from **open()**, **creat()**, **dup()**, or **pipe()**. *type* is the manner in which you want *fd* to be opened, as follows:

r	Read a file
w	Write into a file
a	Append onto a file

Example

The following example obtains a file descriptor with **open()**, and then uses **fdopen()** to build a pointer to the **FILE** structure.

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

void adios(message)
char *message;
{
      fprintf(stderr, "%s\n", message);
      exit(1);
}

main(argc, argv)
int argc; char *argv[];
{
      extern FILE *fdopen();
      FILE *fp;
      int fd;
      int holder;

      if (--argc != 1)
            adios("Usage: example filename");

      if ((fd = open(argv[1], 0)) == -1)
            adios("open failed.");
      if ((fp = fdopen(fd, "r")) == NULL)
            adios("fdopen failed.");

      while ((holder = fgetc(fp)) != EOF) {
            if ((holder > '\177') || (holder < ' '))
                  switch(holder) {
                  case '\t':
                  case '\n':
                        break;
                  default:
                        fprintf(stderr, "Seeing char %d\n", holder);
                        exit(1);
                  }
            fputc(holder, stdout);
      }
}
```

See Also

creat(), dup(), fopen(), libc, open()
POSIX Standard, §8.2.2

Diagnostics

fdopen() returns NULL if it cannot allocate a **FILE** structure. Currently, only 20 **FILE** structures can be allocated per program, including **stdin**, **stdout**, and **stderr**.

feof() — STDIO Function (libc)

Discover stream status
#include <stdio.h>
int feof(*fp*) FILE ****fp*;**

feof() tests the status of the argument stream *fp*. It returns a number other than zero if *fp* has reached the end of file, and zero if it has not. One use of **feof()** is to distinguish a value of -1 returned by **getw()** from an **EOF**.

Example

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

See Also

EOF, libc
ANSI Standard, §7.9.10.2
POSIX Standard, §8.1

ferror() — STDIO Function (libc)

Discover stream status
#include <stdio.h>
**int ferror(*fp*) FILE ****fp*;

ferror() tests the status of the file stream *fp*. It returns a number other than zero if an error has occurred on *fp*. Any error condition that it discovers persists until you either close the stream or call **clearerr()** to clear it. For write routines that employ buffers, call **fflush()** before you call **ferror()**, in case an error occurs on the last block written.

Example

This example reads a word from one file and writes it into another.

```
#include <stdio.h>

main()
{
	FILE *fpin, *fpout;
	int inerr = 0;
	int outerr = 0;
	int word;
	char infile[20], outfile[20];

	printf("Name data file you wish to copy:\n");
	gets(infile);
	printf("Name new file:\n");
	gets(outfile);

	if ((fpin = fopen(infile, "r")) != NULL) {
		if ((fpout = fopen(outfile, "w")) != NULL) {

			for (;;) {
				word = fgetw(fpin);
				if (ferror(fpin)) {
					clearerr(fpin);
					inerr++;
				}

				if (feof(fpin))
					break;
				fputw(word, fpout);
				if (ferror(fpout)) {
					clearerr(fpout);
					outerr++;
				}
			}

		} else {
			printf
				("Cannot open output file %s\n",
					outfile);
			exit(1);
		}

	} else {
		printf("Cannot open input file %s\n", infile);
		exit(1);
	}

	printf("%d - read error(s)  %d - write error(s)\n",
		inerr, outerr);
	exit(0);
}
```

See Also

libc
ANSI Standard, §7.9.10.3
POSIX Standard, §8.1

fetch() — DBM Function (libgdbm)

Fetch a record from a DBM data base
#include <dbm.h>
datum fetch (*key***)**
datum *key***;**

Function **fetch()** retrieves the record with *key* from the currently opened DBM data base. The data base must first have opened by a call to function **dbminit()**.

fetch() returns a pointer to the retrieved record. If no record is available, or if an error occurred, field **dptr** within the returned record is initialized to NULL.

See Also

Notes

For a statement of copyright and permissions on this routine, see the Lexicon entry for **libgdbm**.

fflush() — STDIO Function (libc)

Flush output stream's buffer
#include <stdio.h>
int fflush(*fp***) FILE ****fp***;**

fflush() flushes any buffered output data associated with the file stream *fp*. The file stream stays open after **fflush()** is called. **fclose()** calls **fflush()**, so there is no need for you to call it when normally closing a file or buffer.

Example

This example demonstrates **fflush()**. When run, you will see the following:

```
Line 1
-----
Line 1
-----
Line 1
Line 2
-----
```

The call

```
fprintf(fp, "Line 2\n");
```

goes to a buffer and is not in the file when file **foo** is listed. However if you redirect the output of this program to a file and list the file, you will see:

```
Line 1
Line 1
Line 1
Line 2
-----
-----
-----
```

because the line

```
printf("-----\n");
```

goes into a buffer and is not printed until the program is over and all buffers are flushed by **exit()**.

Although the COHERENT screen drivers print all output immediately, not all operating systems work this way, so when in doubt, **fflush()**.

```
#include <stdio.h>

main()
{
	FILE *fp;
```

```
	if (NULL == (fp = fopen("foo", "w")))
		exit(1);
	fprintf (fp, "Line 1\n");
	fflush (fp);
	system ("cat foo"); /* print Line 1 */

	printf("-----\n");
	fprintf(fp, "Line 2\n");
	system("cat foo"); /* print Line 1 */
	printf("-----\n");

	fflush(fp);
	system("cat foo"); /* print Line 1 Line 2 */
	printf("-----\n");
}
```

See Also

fclose(), libc, setbuf(), write()
ANSI Standard, §7.9.5.2
POSIX Standard, §8.1

Diagnostics

fflush() returns **EOF** if it cannot flush the contents of the buffers; otherwise it returns a meaningless value.

Note, also, that all STDIO routines are buffered. **fflush()** should be used to flush the output buffer if you follow a STDIO routine with an unbuffered routine.

ffs() — Sockets Function (libsocket)

Translate a bit mask into an integer value
int mask (*mask***);**

Function **ffs()** translates the bit mask *mask* into an integer value. It returns the integer value of the first bit to be turned on (i.e., one, two, three, etc.). If no bit is turned on within *mask*, it returns zero.

See Also

libsocket

Notes

This function is used by a number of X programs that manipulate fonts. COHERENT includes it for compatibility with X11R6.

fgetc() — STDIO Function (libc)

Read character from stream
#include <stdio.h>
int fgetc(*fp***) FILE ****fp***;**

fgetc() reads characters from the input stream *fp*. In general, it behaves the same as the macro **getc()**: it runs more slowly than **getc()**, but yields a smaller object module when compiled.

Example

This example counts the number of lines and “sentences” in a file.

```
#include <stdio.h>

main()
{
	FILE *fp;
	int filename[20];
	int ch;
	int nlines = 0;
	int nsents = 0;

	printf("Enter file to test: ");
	gets(filename);
```

```
	if ((fp = fopen(filename,"r")) == NULL) {
		printf("Cannot open file %s.\n", filename);
		exit(1);
	}

	while ((ch = fgetc(fp)) != EOF) {
		if (ch == '\n')
			++nlines;

		else if (ch == '.' || ch == '!' || ch == '?') {
			if ((ch = fgetc(fp)) != '.')
				++nsents;

			else
				while((ch=fgetc(fp)) == '.')
					;
			ungetc(ch, fp);
		}
	}

	printf("%d line(s), %d sentence(s).\n",
		nlines, nsents);
}
```

See Also

getc(), libc
ANSI Standard, §7.9.7.1
POSIX Standard, §8.1

Diagnostics

fgetc() returns EOF at end of file or on error.

fgetpos() — STDIO Function (libc)

Get value of file-position indicator
#include <stdio.h>
int
fgetpos(*fp*, *position***)**
FILE **fp*; **fpos_t** **position*;

fgetpos() copies the value of the file-position indicator for the file stream pointed to by *fp* into the area pointed to by *position*. *position* is of type **fpos_t**, which is defined in the header **stdio.h**.

The function **fsetpos()** can use the information written into *position* to return the file-position indicator to where it was when **fgetpos()** was called.

fgetpos() returns zero if all went well. If an error occurred, it returns nonzero and sets **errno** to an appropriate value.

Example

This example seeks to a random line in a very large file.

```
#include <math.h>
#include <stdarg.h>
#include <stddef.h>
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

void
fatal(message)
char *message;
{
	fprintf(stderr, "%s\n", message);
	exit(1);
}
```

```
main(argc, argv)
int argc; char *argv[];
{
	int c;
	long count;
	FILE *ifp, *tmp;
	fpos_t loc;

	if (argc != 2)
		fatal("usage: fscanf inputfile\n");
	if ((ifp = fopen(argv[1], "r")) == NULL)
		fatal("Cannot open %s\n", argv[1]);
	if((tmp = tmpfile()) == NULL)
		fatal("Cannot build index file");

	/* seed random-number generator */
	srand ((unsigned int)time(NULL));

	for (count = 1;!feof(ifp); count++) {
		/* for monster files */
		if (fgetpos(ifp, &loc))
			fatal ("fgetpos error");

		if (fwrite(&loc, sizeof(loc), 1, tmp) != 1)
			fatal("Write fail on index");
		rand();
		while('\n' != (c = fgetc(ifp)) && EOF != c)
			;
	}

	count = rand() % count;
	fseek(tmp, count * sizeof(loc), SEEK_SET);

	if(fread(&loc, sizeof(loc), 1, tmp) != 1)
		fatal("Read fail on index");

	fsetpos(ifp, &loc);
	while((c = fgetc(ifp)) != EOF) {
		if(c =='@')
			putchar('\n');
		else
			putchar(c);

		if(c == '\n')
			break;
	}
}
```

See Also

fseek(), fsetpos(), ftell(), libc, rewind()
ANSI Standard, §7.9.9.1

Notes

The ANSI Standard introduced **fgetpos()** and **fsetpos()** to manipulate a file whose file-position indicator cannot be stored within a **long**. Under COHERENT **fgetpos()** behaves the same as the function **ftell()**.

fgets() — STDIO Function (libc)

Read line from stream
#include <stdio.h>
char ***fgets(***s*, *n*, *fp***)**
char **s*; **int** *n*; **FILE** **fp*;

fgets() reads characters from the stream *fp* into string *s* until either *n*-1 characters have been read, or a newline or EOF is encountered. It retains the newline, if any, and appends a null character at the end of of the string. **fgets()** returns the argument *s* if any characters were read, and NULL if none were read.

Example

This example looks for the pattern given by **argv[1]** in standard input or in file **argv[2]**. It demonstrates the functions **pnmatch()**, **fgets()**, and **freopen()**.

```
#include <stdio.h>
#define MAXLINE 128
char buf[MAXLINE];

void fatal(s) char *s;
{
	fprintf(stderr, "pnmatch: %s\n", s);
	exit(1);
}

main(argc, argv)
int argc; char *argv[];
{
	if (argc != 2 && argc != 3)
		fatal("Usage: pnmatch pattern [ file ]");

	if (argc==3 && freopen(argv[2], "r", stdin)==NULL)
		fatal("cannot open input file");

	while (fgets(buf, MAXLINE, stdin) != NULL) {
		if (pnmatch(buf, argv[1], 1))
			printf("%s", buf);
	}

	if (!feof(stdin))
		fatal("read error");
	exit(0);
}
```

See Also

fgetc(), gets(), libc
ANSI Standard, §7.9.7.2
POSIX Standard, §8.1

Diagnostics

fgets() returns NULL if an error occurs, or if **EOF** is seen before any characters are read.

fgetw() — STDIO Function (libc)

Read integer from stream
#include <stdio.h>
int fgetw(*fp*) FILE *_fp_;

fgetw() reads an integer from the stream *fp*.

Example

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

See Also

fputw(), libc

Notes

fgetw() returns EOF on errors. A call to **feof()** or **ferror()** may be necessary to distinguish this value from a genuine end-of-file signal.

field — Definition

A **field** is an area that is set apart from whatever surrounds it, and that is defined as containing a particular type of data. In the context of C programming, a field is either an element of a structure, or a set of adjacent bits within an **int**.

See Also

bit map, data formats, Programming COHERENT, structure

file — Definition

The way to access bits

The term **file** is used throughout the world of computing. Because there are several distinct types of COHERENT "files," understanding what COHERENT means by a "file" can help you grasp how COHERENT works.

A file is a mass of bits that is given a name and is stored on some physical medium (e.g., floppy disk, hard disk, RAM disk, or CD-ROM). These bits may represent data (e.g., ASCII or EBCDIC characters) or machine-executable instructions. COHERENT defines a number of different types of files. A file's type defines its behavior. Some common file types include the following:

regular This file points to a location on a disk, which can be read or written. The location pointed to can contain data (e.g., text) or executable instructions in the form of shell commands or binary instructions. Regular files are sometimes called *ordinary* files.

directory
A directory holds the names and addresses of other files, including other directories.

special Special files designate COHERENT devices. A device can represent a physical device, such as a floppy disk drive, a printer port, or a serial port. It can also represent a part of a physical device, such as a RAM disk (representing part of memory) or one partition of a hard disk. It can also represent a logical device that has no physical counterpart, like the bit bucket **/dev/null**.

Special files come in two flavors: *character special* and *block special*. The former access data in streams (that is, one character at a time), and so access devices like tape drives and serial ports. The latter access one block at a time, and so access disk drives and other devices that return their data in block-sized chunks. (COHERENT defines a block as being 512 characters.)

FIFO This is a variety of regular file that contains semantics to hook together two processes, just like a pipe '|' in the COHERENT shell. See the Lexicon article **named pipe** for details on this variety of file.

process
This kind of file corresponds one-to-one with the existence of a process on a system. It tends to be short-lived.

Files live with a *file system*, which organizes the files hierarchically within directories. The Lexicon entry for the command **mkfs** gives some technical information on how a file system is constructed. The Lexicon entry for the command **mount** gives some information on how a file system relates to device on which it lives, and how different file systems from different partitions are hooked together to form one large file system for the entire computer.

The same file can have (and be accessed by) more than one name. The Lexicon entry for the command **ln** shows how you can link additional names to a file. The entry for the system call **unlink()** gives some details on the relationship between a file and its names.

Finally, a file has *permissions* associated with it. Every file is owned by someone; and the owner can restrict access to the file if she wishes. The Lexicon entry for the command **ls** describes what permissions are available for a file. The entry for the command **chmod** shows how you can change permissions on a file. The entry for the command **umask** shows how you can change the permissions that COHERENT gives by default to any files that you create.

See Also

chgrp, chmod, chown, directory, FILE, device drivers, ls, mkfs, named pipe, open(), Programming COHERENT, stream, umask, Using COHERENT
ANSI Standard §4.9.3

file — Command

Guess a file's type
file *file ...*

file examines each *file* and takes an educated guess as to its type. **file** recognizes the following classes of text files: files of commands to the shell; files containing the source for a C program; files containing **yacc** or **lex** source; files containing assembly language source; files containing unformatted documents that can be passed to **nroff**; and plain text files that fit into none of the above categories.

file recognizes the following classes of non-text or binary data files: the various forms of archives, object files, and link modules for various machines, and miscellaneous binary data files.

See Also

commands, ls, size

Notes

Because **file** only reads a set amount of data to determine the class of a text file, mistakes can happen.

FILE — Definition

Descriptor for a file stream
#include <stdio.h>

FILE describes a *file stream* which can be either a file on disk or a peripheral device through which data flow. It is defined in the header file **stdio.h**.

A pointer to **FILE** is returned by **fopen()**, **freopen()**, **fdopen()**, and related functions.

The **FILE** structure is as follows:

```
typedef struct    FILE
{
      unsigned char *cp,
                    *dp,
                    *bp;
      int   cc;
      int   (*gt)(),
            (*pt)();
      int   ff;
      char  fd;
      int   uc;
} FILE;
```

cp points to the current character in the file. **dp** points to the start of the data within the buffer. **bp** points to the file buffer. **cc** is the number of unprocessed characters in the buffer. **gt** and **pt** point, respectively, to the functions **getc()** and **putc()**. **ff** is a bit map that holds the various file flags, as follows:

_FINUSE	0x01	Unused
_FSTBUF	0x02	Used by macro **setbuf()**
_FUNGOT	0x04	Used by **ungetc()**
_FEOF	0x08	Tested by macro **feof()**
_FERR	0x10	Tested by macro **ferror()**

fd is the file descriptor, which is used by low-level routines like **open()**; it is also used by **reopen()**. Finally, **uc** is the character that has been “ungotten” by the function **ungetc()**, should it be used.

See Also

fopen(), freopen(), Programming COHERENT, stdio.h, stream
ANSI Standard, §7.9.1

file descriptor — Definition

A **file descriptor** is an integer that indexes an area in COHERENT’s internal list of file descriptors. COHERENT system calls, including **open()**, **close()**, and **lseek()**, use it to describe a file.

Please note that a file descriptor is *not* the same as a **FILE** structure, which is used by the STDIO routines **fopen()**, **fclose()**, or **fread()**.

See Also

file, FILE, Programming COHERENT

fileno() — STDIO Function (libc)

Get file descriptor
#include <stdio.h>
int fileno(*fp***) FILE ***fp***;**

fileno() returns the file descriptor associated with the file stream *fp*. The file descriptor is the integer returned by **open()** or **creat()**; it corresponds to a **FILE** structure, as returned by the STDIO function **fopen()**.

Example

This example reads a file descriptor and prints it on the screen.

```
#include <stdio.h>

main(argc,argv)
int argc; char *argv[];
{
      FILE *fp;
      int fd;

      if (argc !=2) {
            printf("Usage: fd_from_fp filename\n");
            exit(0);
      }

      if ((fp = fopen(argv[1], "r")) == NULL) {
            printf("Cannot open input file\n");
            exit(0);
      }

      fd = fileno(fp);
      printf("The file descriptor for %s is %d\n",
            argv[1], fd);
}
```

See Also

FILE, file descriptor, libc
POSIX Standard, §8.2.1

filsys.h — Header File

Structures and constants for super block
#include <sys/filsys.h>

filsys.h declares structures and constants used by functions that manipulate a file system's super block.

See Also

header files

filter — Definition

A *filter* is a program that reads a stream of input, transforms it in a precisely defined manner, and writes it to another stream. Two or more filters can be coupled with *pipes* to perform a complex transformation on a stream of input.

See Also

pipe, Using COHERENT

find — Command

Search for files satisfying a pattern
find *directory ...* **[***expression ...***]**

find traverses each given *directory*, testing each file or subdirectory found with the *expression* part of the command line. The test can be the basis for deciding whether to process the file with a given command.

If the command line specifies no *expression* or specifies no execution or printing (**-print**, **-exec**, or **-ok**), by default **find** prints the pathnames of the files found.

In the following, *file* means any file: directory, special file, ordinary file, and so on. Numbers represented by *n* may be optionally prefixed by a '+' or '-' sign to signify values greater than *n* or less than *n*, respectively.

find recognizes the following *expression* primitives:

-**atime** *n* Match if the file was accessed in the last *n* days.

-**ctime** *n* Match if the i-node associated with the file was changed in the last *n* days, as by **chmod**.

-**exec** *command*
Match if *command* executes successfully (has a zero exit status). The *command* consists of the following arguments to **find**, terminated by a semicolon ';' (escaped to get past the shell). **find** substitutes the current pathname being tested for any argument of the form '{}'.

-**group** *name*
Match if the file is owned by group *name*. If *name* is a number, the owner must have that group number.

-**inum** *n* Match if the file is associated with i-number *n.*

-**links** *n* Match if the number of links to the file is *n.*

-**mtime** *n* Match if the most recent modification to the file was *n* days ago.

-**name** *pattern*
Match if the file name corresponds to *pattern*, which may include the special characters '*', '?', and '[...]' recognized by the shell **sh**. The *pattern* matches only the part of the file name after any slash ('/') characters.

-**newer** *file* Match if the file is newer than *file.*

-**nop** Always match; does nothing.

-**ok** *command*
Same as **-exec** above, except prompt interactively and only executes *command* if the user types response **'y'**.

-**perm** *octal* Match if owner, group, and other permissions of the file are the *octal* bit pattern, as described in **chmod**. When *octal* begins with a **'-'** character, more of the permission bits (setuid, setgid, and sticky bit) become significant.

-**print** Always match; print the file name.

-**size** *n* Match if the file is *n* blocks in length; a block is 512 bytes long.

-**type** *c* Match if the type of the file is *c*, chosen from the set **bcdfmp** (for block special, character special, directory, ordinary file, multiplexed file, or pipe, respectively).

-**user** *name* Match if the file is owned by user *name.* If *name* is a number, the owner must have that user number.

exp1 exp2 Match if both expressions match. **find** evaluates *exp2* only if *exp1* matches.

exp1 **-a** *exp2*
Match if both expressions match, as above.

exp1 **-o** *exp2*
Match if either expression matches. **find** evaluates **exp2** only if **exp1** does not match.

! *exp* Match if the expression does *not* match.

(*exp*) Parentheses are available for expression grouping.

Examples

A **find** command to print the names of all files and directories in user **fred**'s directory is:

```
find /usr/fred
```

The following, more complicated **find** command prints out information on all **core** and object (**.o**) files that have not been changed for a day. Because some characters are special both to **find** and **sh**, they must be escaped with '\' to avoid interpretation by the shell.

```
find / \( -name core -o -name \*.o \) -mtime +1 \
-exec ls -l {} \;
```

Finally, the following example implements a simple tool for keeping files on two COHERENT systems in synch with each other. **find** reads directory **src** and passes to **uucp** the names of all files that are newer than file **last_upload**. It then uses the command **touch** to update the date on **last_upload**, to use it as a marker of when the last upload was performed.

```
find $HOME/src -type f -newer last_upload | while read filename
do
      uucp -r -nyou $filename yoursystem!~/
      echo Queued file $filename to yoursystem ...
done | mail somebodyorother
touch last_upload
```

See Also

chmod, commands, ls, sh, srcpath, test

findmouse — Command

Examine a port to see if a mouse is plugged into it
/usr/local/bin/findmouse *port*

The command **findmouse** opens *port* so you can examine whether a mouse is plugged into it.

port must be the full path name of the local, polled, serial-port device. For example, to check whether a mouse is plugged into serial port 1, use the command:

```
/usr/local/bin/findmouse /dev/com1pl
```

When you invoke **findmouse**, it opens *port*, then asks you to “wiggle” your mouse. As you move the mouse around your desk, **findmouse** polls the port and display on the screen any data read from it. You should see the mouse data on your screen in the form of two-digit hexadecimal numbers.

To exit from **findmouse**, press any key.

findmouse prints an error message an exits should use incorrect command syntax, or if it cannot open a requested port.

See Also

asy, commands, poll()

firstkey() — DBM Function (libgdbm)

Retrieve the first record from a DBM data base
#include <dbm.h>
datum firstkey()

Function **firstkey()** retrieves the first record from the currently open DBM data base. The data base must have been opened by a call to function **dbminit()**.

firstkey() returns a pointer to the retrieved record. If no record is available (i.e., the data base is empty), or if an error occurred, field **dptr** within the returned record is initialized to NULL.

Please note that the hashing algorithm used the DBM functions dictates which record is “first” within the data base. A loop that uses this function plus the function **nextkey()** will retrieve every record from the data base; however, the records probably will not be in the order you expect.

See Also

Notes

For a statement of copyright and permissions on this routine, see the Lexicon entry for **libgdbm**.

fixterm() — terminfo Function

Set the terminal into program mode
#include <curses.h>
fixterm()

COHERENT comes with a set of functions that let you use **terminfo** descriptions to manipulate a terminal. **fixterm()** restores the terminal to its internal conditions, as set by the **curses/terminfo** library. Your program should call **fixterm()** after it returns from a shell escape.

See Also

curses.h, resetterm(), terminfo

float — C Keyword

Data type

Floating point numbers are a subset of the real numbers. Each has a built-in radix point (or "decimal point") that shifts, or "floats", as the value of the number changes. It consists of the following: one sign bit, which indicates whether the number is positive or negative; bits that encode the number's *exponent*; and bits that encode the number's *fraction*, or the number upon which the exponent works. In general, the magnitude of the number encoded depends upon the number of bits in the exponent, whereas its precision depends upon the number of bits in the fraction.

The ranges of values that can be held by a COHERENT **float** are set in header file **float.h**.

The exponent often uses a *bias*. This is a value that is subtracted from the exponent to yield the power of two by which the fraction will be increased.

Floating point numbers come in two levels of precision: single precision, called **float**s; and double precision, called **double**s. With most microprocessors, **sizeof(float)** returns four, which indicates that it is four **char**s (bytes) long, and **sizeof(double)** returns eight.

Several formats are used to encode **float**s, including IEEE, DECVAX, and BCD (binary coded decimal).

The following describes DECVAX, IEEE, and BCD formats, for your information.

DECVAX Format

The 32 bits in a **float** consist of one sign bit, an eight-bit exponent, and a 24-bit fraction, as follows. Note that in this diagram, 's' indicates "sign", 'e' indicates "exponent", and 'f' indicates "fraction":

seee eeee	*Byte 4*
efff ffff	*Byte 3*
ffff ffff	*Byte 2*
ffff ffff	*Byte 1*

The exponent has a bias of 129.

If the sign bit is set to one, the number is negative; if it is set to zero, then the number is positive. If the number is all zeroes, then it equals zero; an exponent and fraction of zero plus a sign of one ("negative zero") is by definition not a number. All other forms are numeric values.

The most significant bit in the fraction is always set to one and is not stored. It is usually called the "hidden bit".

The format for **double**s simply adds another 32 fraction bits to the end of the **float** representation, as follows:

seee eeee	*Byte 8*
efff ffff	*Byte 7*
ffff ffff	*Byte 6*
ffff ffff	*Byte 5*
ffff ffff	*Byte 4*
ffff ffff	*Byte 3*
ffff ffff	*Byte 2*
ffff ffff	*Byte 1*

IEEE Format

The IEEE encoding of a **float** is the same as that in the DECVAX format. Note, however, that the exponent has a bias of 127, rather than 129.

Unlike the DECVAX format, IEEE format assigns special values to several floating point numbers. Note that in the following description, a *tiny* exponent is one that is all zeroes, and a *huge* exponent is one that is all ones:

- A tiny exponent with a fraction of zero equals zero, regardless of the setting of the sign bit.
- A huge exponent with a fraction of zero equals infinity, regardless of the setting of the sign bit.
- A tiny exponent with a fraction greater than zero is a denormalized number, i.e., a number that is less than the least normalized number.
- A huge exponent with a fraction greater than zero is, by definition, not a number. These values can be used to handle special conditions.

An IEEE **double**, unlike DECVAX format, increases the number of exponent bits. It consists of a sign bit, an 11-bit exponent, and a 53-bit fraction, as follows:

seee eeee	*Byte 8*
eeee ffff	*Byte 7*
ffff ffff	*Byte 6*
ffff ffff	*Byte 5*
ffff ffff	*Byte 4*
ffff ffff	*Byte 3*
ffff ffff	*Byte 2*
ffff ffff	*Byte 1*

The exponent has a bias of 1,023. The rules of encoding are the same as for **float**s.

BCD Format

The BCD format ("binary coded decimal", also called "packed decimal") is used to eliminate rounding errors that alter the worth of an account by a fraction of a cent. It consists of a sign, an exponent, and a chain of four-bit numbers, each of which is defined to hold the values zero through nine.

A BCD **float** has a sign bit, seven bits of exponent, and six four-bit digits. In the following diagrams, 'd' indicates "digit":

seee eeee	*Byte 4*
dddd dddd	*Byte 3*
dddd dddd	*Byte 2*
dddd dddd	*Byte 1*

A BCD **double** has a sign bit, 11 bits of exponent, and 13 four-bit digits, as follows:

seee eeee	*Byte 8*
eeee dddd	*Byte 7*
dddd dddd	*Byte 6*
dddd dddd	*Byte 5*
dddd dddd	*Byte 4*
dddd dddd	*Byte 3*
dddd dddd	*Byte 2*
dddd dddd	*Byte 1*

Passing the hexadecimal numbers A through F in a digit yields unpredictable results.

The following rules apply when handling BCD numbers:

- A tiny exponent with a fraction of zero equals zero.
- A tiny exponent with a fraction of non-zero indicates a denormalized number.
- A huge exponent with a fraction of zero indicates infinity.
- A huge exponent with a fraction of non-zero is, by definition, not a number; these non-numbers are used to indicate errors.

COHERENT Floating Point

COHERENT 286 uses DECVAX floating-point format. COHERENT 386 uses IEEE floating-point format. Please note that this does *not* mean that the COHERENT-386 floating-point software fully implements the IEEE standard; for example, it does not support denormals.

To allow you to convert binary data from one floating-point format to another, COHERENT comes with four functions with which you can convert DECVAX-format floating-point numbers to IEEE format, and vice versa. They are as follows:

decvax_d() Convert an IEEE **double** to DECVAX format.

decvax_f() Convert an IEEE **float** to DECVAX format.

ieee_d() Convert a DECVAX **double** to IEEE format.

ieee_f() Convert a DECVAX **float** to IEEE format.

For details, see their respective entries in the Lexicon.

See Also

C keywords, data formats, decvax_d, decvax_f, double, ecvt(), em87, fcvt(), float, float.h, gcvt(), ieee_d, ieee_f
The Art of Computer Programming, vol. 2, page 180*ff*
ANSI Standard, §6.1.2.5

Notes

The COHERENT-386 preprocessor implicitly defines the macro **_IEEE**, whereas the COHERENT-286 preprocessor implicitly defines the macro **_DECVAX**. These can be used to conditionally include code that applies to a specific edition of COHERENT. If you were writing code that intensively used floating-point numbers and you want to compile the code under both editions of COHERENT, you can write code of the form:

```
#ifdef _DECVAX
          ...
#elif _IEEE
          ...
#endif
```

The C preprocessor under each edition of COHERENT will ensure that the correct code is included for compilation.

float.h — Header File

Define constants for floating-point numbers

The header file **float.h** defines the following manifest constants, which mark the limits for computation of floating-point numbers. The prefixes **DBL**, **FLT**, and **LDBL** refer, respective, to **double**, **float**, and **long double**:

DBL_DIG
: Number of decimal digits of precision.

DBL_EPSILON
: Smallest possible floating-point number *x*, such that 1.0 plus *x* does not test equal to 1.0.

DBL_MANT_DIG
: Number of digits in the floating-point mantissa for base **FLT_RADIX**.

DBL_MAX
: Largest number that can be held by type **double**.

DBL_MAX_EXP
: Largest integer such that the value of **FLT_RADIX** raised to its power minus one is less than or equal to **DBL_MAX**.

DBL_MAX_10_EXP
: Largest integer such that ten raised to its power is less than or equal to **DBL_MAX**.

DBL_MIN
: Smallest number that can be held by type **double**.

DBL_MIN_EXP
: Smallest integer such that the value of **FLT_RADIX** raised to its power minus one is greater than or equal to **DBL_MIN**.

DBL_MIN_10_EXP
: Smallest integer such that ten raised to its power is greater than or equal to **DBL_MAX**.

FLT_DIG
: Number of decimal digits of precision.

FLT_EPSILON
: Smallest floating-point number *x*, such that 1.0 plus *x* does not test equal to 1.0.

FLT_MANT_DIG
: Number of digits in the floating-point mantissa for base **FLT_RADIX**.

FLT_MAX
: Largest number that can be held by type **float**.

FLT_MAX_EXP
: Largest integer such that the value of **FLT_RADIX** raised to its power minus one is less than or equal to **FLT_MAX**.

FLT_MAX_10_EXP
: Largest integer such that ten raised to its power is less than or equal to **FLT_MAX**.

FLT_MIN
: Smallest number that can be held by type **float**.

FLT_MIN_EXP
: Smallest integer such that the value of **FLT_RADIX** raised to its power minus one is greater than or equal to **FLT_MIN**.

FLT_MIN_10_EXP
: Smallest integer such that ten raised to its power is greater than or equal to **FLT_MIN**.

FLT_RADIX
: Base in which the exponents of all floating-point numbers are represented.

FLT_ROUNDS
Manner of rounding used by the implementation. The ANSI Standard defines the rounding codes as follows:

-1 Indeterminable, i.e., no strict rules apply
0 Toward zero, i.e., truncation
1 To nearest, i.e., rounds to nearest representable value
2 Toward positive infinity, i.e., always rounds up
3 Toward negative infinity, i.e., always rounds down

COHERENT uses type-1 rounding.

LDBL_DIG
Number of decimal digits of precision.

LDBL_EPSILON
Smallest floating-point number *x*, such that 1.0 plus *x* does not test equal to 1.0.

LDBL_MANT_DIG
Number of digits in the floating-point mantissa for base **FLT_RADIX**.

LDBL_MAX
Largest number that can be held by type **long double**.

LDBL_MAX_EXP
Largest integer such that the value of **FLT_RADIX** raised to its power minus one is less than or equal to **LDBL_MAX**.

LDBL_MAX_10_EXP
Largest integer such that ten raised to its power is less than or equal to **LDBL_MAX**.

LDBL_MIN
Smallest number that can be held by type **long double**. Must be no greater than 1E-37.

LDBL_MIN_EXP
Smallest integer such that the value of **FLT_RADIX** raised to its power minus one is greater than or equal to **LDBL_MIN**.

LDBL_MIN_10_EXP
Smallest integer such that ten raised to its power is greater than or equal to **LDBL_MIN**.

See Also

double, float, header files
ANSI Standard, §5.2.4.2.2

Notes

COHERENT's C compiler does not yet implement type **long double**.

floor() — Mathematics Function (libm)

Set a numeric floor
#include <math.h>
double floor(*z*) double *z*;

floor() sets a numeric floor. It returns a double-precision floating point number whose value is the largest integer less than or equal to *z*.

Example

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

See Also

abs(), ceil(), fabs(), frexp(), libm
ANSI Standard, §7.5.6.3
POSIX Standard, §8.1

floppy disks — Technical Information

The COHERENT system lets you read or write to floppy disks, using a variety of different formats. You can choose the format that best suits the task at hand.

Disks Supported

COHERENT lets you use either 3.5-inch or 5.25-inch disks, in either high or low density; what you use depends upon the type of hardware that you have. The following table gives some commonly used diskette device names and formats. The minor number of each device is also given; note that all floppy-disk devices have the major number of 4:

Device Name	Sectors/Track	Heads	Sectors	Bytes	Format	Minor Number
/dev/f9a0	9	2	720	360 KB	5.25"	12
/dev/f9a1	9	2	720	360 KB	5.25"	28
/dev/fqa0	9	2	1440	720 KB	3.5"	13
/dev/fqa1	9	2	1440	720 KB	3.5"	29
/dev/fha0	15	2	2400	1.2 MB	5.25"	14
/dev/fha1	15	2	2400	1.2 MB	5.25"	30
/dev/fva0	18	2	2880	1.44 MB	3.5"	15
/dev/fva1	18	2	2880	1.44 MB	3.5"	31

Device names ending in ‘0’ indicate drive A:, names ending in ‘1’ indicate drive B:. For a fuller description of COHERENT’s floppy-disk devices, see the Lexicon entry for **fd**.

MS-DOS Format

COHERENT lets you read or write to floppy disks that contain MS-DOS file systems. Both tasks use the commands **doscp** or **doscpdir**. These commands are discussed in full in their respective Lexicon entries.

To read files from an MS-DOS disk, use **doscp** with the name of the appropriate for the floppy-disk device that you will be using (as given in the above table). For example, to copy binary file **fred.exe** to the current directory from a low-density, 5.25-inch MS-DOS floppy disk in drive A, use the following command:

```
doscp /dev/f9a0:fred.exe .
```

The following command copies to the current directory all files on a high-density, 5.25-inch MS-DOS floppy disk in drive B:

```
doscp /dev/fha1:\* .
```

To write a file to a preformatted MS-DOS floppy disk, again use the **doscp** command, but invert the order of the arguments. For example, to write file **fred.ms**, which contains text, to a low-density, 5.25-inch MS-DOS floppy disk in drive A, use the following command:

```
doscp -a fred.ms /dev/f9a0:
```

Note that the ‘a’ flag in the command line tells COHERENT to convert linefeeds to the linefeed/carriage return combination, as used by MS-DOS. You will want to use this flag *only* when transferring text files to or from an MS-DOS floppy disk.

The following command copies all files in the current directory to a high-density, 3.5-inch MS-DOS floppy disk in drive B:

```
doscpdir . /dev/fva1:
```

Note that when you copy a file to an MS-DOS floppy disk, COHERENT observes the MS-DOS file-name conventions: it permits only eight characters to the left of the period, and only three characters to the right of it.

(It should be noted in passing that you can use the **doscp** or **doscpdir** to read files from or write files to an MS-DOS partition on your hard disk. All that is necessary is to replace the name of floppy-disk device with that of the hard-disk device for the partition in question. See the Lexicon entry for **at** for a list of hard-disk devices; see the entry for **fdisk** for information on how to read the layout of your hard disk; and see the entries for **doscp** and **doscpdir** for details of how to use these commands.)

Finally, COHERENT lets you format a floppy disk and create an MS-DOS file system on it. To do so, you must use the commands **fdformat** and **dosformat**. **fdformat** is described in detail in its Lexicon article.

To format a high-density, 5.25-inch floppy disk in drive B and write an MS-DOS file system onto it, use the following commands:

```
/etc/fdformat -av /dev/fha1
dosformat /dev/fha1:
```

COHERENT Format

If you wish, you can create a COHERENT file system on a floppy disk, mount it, and use standard COHERENT commands to manipulate the files on it. This illustrates well the fact that to COHERENT a file system is a file system, whether it resides on a hard, a floppy disk, or any other mass-storage device. You can use such mountable floppy disks as an easy method of backing up files, or as a flexible extension to any other file system that you have currently mounted.

To create a COHERENT file system on a floppy disk, you must use the commands **fdformat** and **mkfs**. Each is described in detail in its own Lexicon article. The following example creates a COHERENT file system on a high-density, 3.5-inch floppy disk placed in drive B:

```
/etc/fdformat -av /dev/rfva1
/etc/mkfs /dev/fva1 2880
```

In this example, command **fdformat** formatted the disk. The option **-v** tells **fdformat** to use its verification mode. This takes longer, but ensures that the disk is good. If this command fails, it means that the floppy disk has a bad block or sector: throw it away and try again.

Command **mkfs** builds a COHERENT file system on the disk. The file system has 2,880 blocks (1.44 megabytes) of space, which is appropriate for a high-density, 3.5-inch floppy disk.

Now that the file system is created on the disk, you must mount it. To do so, use the script **mount**; this is described in its Lexicon entry. This mounts the file system on directory **/f0** if the disk drive is drive 0 (A:); or **f1** if the disk drive is drive 1 (B:).

While it is customary to mount file systems under directory ‘/’, you are not required to do it. For example, if your login identifier is **fred** and your home directory is **/usr/fred**, you can mount the floppy disk’s file system onto a subdirectory of **/usr/fred** and so make the floppy disk, in effect, an extension of your home directory. To mount a floppy on a directory other than its default, use the command **/etc/mount**. The following command does this for the 3.5-inch disk we formatted in the above example:

```
/etc/mount /dev/fva1 /usr/fred/temp
```

Now, all files you copy into directory **/usr/fred/temp** using the **cp** command will be written directly onto the floppy disk. Note that you may need to log in as the superuser **root** and use the command **chown** to ensure that **fred** owns the file system on that floppy disk. For details on **chown**, see its entry in the Lexicon. For details on shorthand notations for **mount**, see its entry in the Lexicon.

One important point about mounting file systems: before you remove a COHERENT-formatted floppy disk from its drive, you **must** first use the command **/etc/umount** to unmount its file system. If you do not, all data that COHERENT has stored in its buffers will not be written to the disk, and may be lost. Worse, if you remove one COHERENT disk and insert another without unmounting the old disk and mounting the new one, COHERENT will write all data in its buffers onto the new disk without regard for what that disk contains; in all likelihood, this will trash the file system on the new disk and render its data unreadable. So, the lesson is: *always unmount a floppy disk before you remove it!*. To unmount the floppy disk we used in our previous example, use the command:

```
/etc/umount /dev/fva1
```

By the way, that’s not a misprint: the command is **umount**, not “unmount”.

Finally, please note that you can mount only a COHERENT file system. You *cannot* mount a file system created with MS-DOS, XENIX, or any other operating system.

You can, however *import* a set of files — including their directory structure — from UNIX, XENIX, or any other UNIX-like operating system by using the utilities. **cpio** or **tar**. Each of these utilities uses a backup algorithm that is implemented on many operating systems. To import files from another operating system, go to the machine that holds the files you want and use its version of **cpio** or **tar** to back up the files or directories to a set of floppy disks or cartridge tape. Then bring the floppy disks back to your COHERENT system and use COHERENT’s implementation of **cpio** to read the back-up disks. The following section gives directions on how to do this; or see the Lexicon entries for **cpio** and **tar** for more information.

Raw Format

Finally, COHERENT lets you use floppy disks in their raw form as a backup medium, much as you would use magnetic tape on a larger computer. You must first use the command **/etc/fdformat** with the **-v** option to format the floppy disks you will be using; it is also wise to label and number the disks so you can keep them in some reasonable order. Then you can use any of COHERENT's archiving utilities, such as **tar** or **cpio**, to archive directories or entire file systems onto the disks. It is recommended that you format a generous supply of floppy disks before you begin; if you run short of disks while archiving your files, you will have to abort, format more disks, and begin again. For details on how to use the archiving programs, see their respective entries in the Lexicon.

Interleave

The "interleave" of a disk device refers to the pattern with which blocks are scattered around a disk cylinder. It can have a drastic effect on the speed with which data are read from and written to a disk.

The interleave is set by the file system written onto that disk. Thus, under COHERENT the interleave is set by the command **/etc/mkfs**. By default, this command sets the interleave pattern to six. You can request a different interleave pattern; however, the proper interleave for a floppy disk can vary wildly, depending upon what disk drives you have, your CPU speed, amount of RAM, and several other variables. The best way to discover the interleave pattern is to experiment.

The following script, by Fred Smith (fredex%fcshome@merk.merk.com), formats a floppy disk to a specified set of factors, generates a file system, and runs a program to exercise it. By running this program with a number of different settings, you can find which is best for your system. You will find this to be especially helpful if you work frequently with floppy disks:

```
# usage: doit <interleave> <skew> <device name> <tracks (not sectors) per drive>
#   for a 3.5dshd in drive 1:  sh doit 3 6 fva1 2880
#   for a 5.25dshd in drive 0: sh doit 3 6 fha0 2400
# assumes that iozone is in the current directory, and that there is a
# subdirectory named 'test', over which the floppy can be mounted.

echo /etc/fdformat -a -i $1 -o $2 /dev/r$3
/etc/fdformat -a -i $1 -o $2 /dev/r$3
/etc/badscan -v -o flop /dev/$3 $4

# in case you want to modify the permissions of the new file system.
# if you don't want to do the vi, then run this as root.
#vi flop

      /etc/mkfs /dev/$3 flop
      /etc/mount /dev/$3 ./test
      cd test
      ../iozone
      cd ..
      /etc/umount /dev/$3
```

Debugging Floppy-Disk Problems

The COHERENT floppy-disk driver has been used frequently by tens of thousands of users over a number of years, and has been found to be sound. However, from time to time a problem can arise. This usually occurs when users install new equipment into their systems. If you continually see error messages that indicate a problem with the floppy-disk drive, e.g., **door open**, try the following steps to diagnose the problem:

1. Is CMOS configured for the floppy-disk drives? The CMOS on your machine may have been "clobbered" by an event that has nothing to do with COHERENT — e.g., a power surge.

   To check your CMOS, you can reboot your system; the BIOS on practically every computer includes a program for reading and resetting the CMOS. Or, you can read the output of device **/dev/cmos**. The Lexicon entry **cmos** describes how to interpret the output of this device.

2. If you have switched hard drives, did you change IDE controllers or alter any jumpers? If the same card controls both floppy and hard drives, you may have moved a jumper wrongly. It may also be that the new controller has a bug.

3. Try using the command **/etc/conf/bin/idtune** to change the value of variable **FL_DSK_CH_PROB**; then use the command **/etc/conf/bin/idmkcoh** to link a new kernel, and boot the new kernel. To check the current value of that variable (or of any tunable variable), use the command **idtune -p**.

4. Is any other equipment conflicting with the drive in question, such as a QIC-80 or QIC-40 tape drive? Try pulling the device in question, and see if that makes the problem go away.

5. Check that all cables are secure and all cards seated properly. If your machine is loaded with equipment, its interior can be a rat's nest of cables and connectors; and while installing new equipment, it is easy to loosen a cable or jar a card so that it no longer works.

6. Try the following command with a floppy disk in place, just after you have booted COHERENT and before any other access to the drive:

```
dd if=/dev/rdev of=/dev/null count=2 bs=30b
```

dev is names the floppy-disk device in question, e.g., **fha0** or **fva1**. This command may help if the driver is not getting the recalibration status it expects.

7. If all else fails, try swapping out the controller or drive. It may be that the device simply has failed.

See Also

Administering COHERENT, badscan, cpio, doscp, doscpdir, dosformat, fd, fdformat, gtar, mkfs, mount, umount

Notes

You can create a version of the COHERENT operating system that runs from a floppy disk. Such a version of COHERENT can be used to create test or backup systems for device drivers or other applications. For directions on how to make a version of COHERENT that boots from a floppy disk, see the Lexicon entry **booting**.

fmap — Command

Measure fragmentation of the free list
fmap *device*

Command **fmap** tests how fragmented the free list is on COHERENT file system *device*. It briefly displays its results and returns an exit status that is equivalent to the percent of fragmentation found on *device*.

You can use the amount of fragmentation of the free list to decide whether to de-fragment *device*. When the freelist is fragmented, writing a file creates a file that is not physically contiguous; and this, in turn, slows disk I/O.

device must be a partition on your hard disk or a floppy disk rather than an entire hard drive. It can be either the raw or the "cooked" (block) device. For example, the command

```
fmap /dev/rat0a
```

tells **fmap** to map the free list on the first partition on the first drive.

Because **fmap** returns an exit status equal to the integer portion of the percentage of fragmentation found, you can use it in a shell script to alert the system administrator when the file system needs attention. For example, the following shell script tests the level of fragmentation on the device given as an argument; if the fragmentation exceeds 5%, it sends mail to the superuser **root**:

```
fmap /dev/$1
a=$?
if expr $a > 5
then
      echo  -n "fmap of " >/tmp/rootmail
      echo -n $1 >>/tmp/rootmail
      echo -n " shows " >>/tmp/rootmail
      echo -n $a >>/tmp/rootmail
      echo " percent fragmentaion" >>/tmp/rootmail
      echo -n $a >>/tmp/rootmail
      echo " is greater than 5"  >>/tmp/rootmail
      echo -n "therefore, it is time to defrag " >>/tmp/rootmail
      echo $1 >>/tmp/rootmail
      echo "bye" >>/tmp/rootmail
      mail root </tmp/rootmail
      rm /tmp/rootmail
fi
exit 0
```

See Also

commands, dpac, fsck, qpac, upac

Notes

fmap was written by Randy Wright (rw@rwsys.wimsey.bc.ca).

fmod() — Mathematics Function (libm)

Calculate modulus for floating-point number
#include <math.h>
double
fmod(*number***,** *divisor***)**
double *number***,** *divisor***;**

The mathematics function **fmod()** divides *number* by *divisor* and returns the remainder. If *divisor* is nonzero, the return value will have the same sign as *divisor*. If *divisor* is zero, however, the COHERENT implementation of **fmod()** returns 0.0 and sets **errno EDOM**.

See Also

ceil(), fabs(), floor(), libm
ANSI Standard, §7.5.6.4
POSIX Standard, §8.1

fmt — Command

Adjust the length of lines in a file of text
fmt [-*width***] [***textfile ... textfile***]**

The command **fmt** reads each *textfile* named on its command line, and adjusts it so that each line is approximately *width* characters long. It preserves indentation and word spacing.

If you name no *textfile* on its command line, **fmt** reads the standard input. If you do not name a *width* on the command line, **fmt** adjusts each line to be approximately 72 characters long.

See Also

commands, elvis

Notes

fmt is part of the **elvis** package. Users usually do not run it on its own.

fnkey — Command

Set/print function keys for the console
fnkey [*n* **[** *string* **]]**

The console keyboard of a COHERENT system includes ten programmable function keys, labeled **F1** through **F10**. Initially, these are programmed to send the escape sequences set by the **nkb** keyboard driver.

The command **fnkey** programs function key **F***n* to send *string*, where *n* is a number from one through ten. If no *string* is given, **fnkey** resets **F***n* to send nothing.

With no argument, **fnkey** prints the current string for each programmed function key.

fnkey also lets you change the default bindings for other special or function keys. See Lexicon articles **keyboard tables** and **nkb** for details.

Example

To set function key **F2** to execute the COHERENT command **date**, use the following command:

```
fnkey 2 'date
'
```

If you type **fnkey** without any arguments, it displays the binding of all function keys including the following:

```
F2:  date\n
```

Files

/dev/console

See Also

commands, keyboard, vtnkb

Diagnostics

fnkey prints

```
cannot open /dev/console
```

if you lack permission to open **/dev/console**.

fnmatch() — String Function (libc)

Match a string with a normal expression
#include <fnmatch.h>
int fnmatch(*pattern***,** *string***,** *flags***)**
const char ****pattern*, ****string***;** *int flags***;**

The function **fnmatch()** checks whether the string to which *string* points matches the normal expression to which *pattern* points. A *normal expression* is one that uses wildcard characters to broaden the range of strings that it matches. For more information, see the Lexicon entry for **wildcards**.

flags is a bit map whose bits are defined in the header file **<fnmatch.h>**. **fnmatch()** recognizes any or all of following flags:

FNM_NOESCAPE
: Disable recognizing the backslash as an escape character.

  If this flag is not set, then prefixing a character in *pattern* with a backslash '\' matches that same character in *string*. For example, the pair '*' in *pattern* matches a literal '*' in *string*; and the pair '\\' in *pattern* matches '\' in *string*.

FNM_PATHNAME
: A slash '/' in *string* matches only a slash in *pattern*. If this flag is set, then a '/' in *string* will not match a wildcard character in *pattern*.

FNM_PERIOD
: A leading period '.' in *string* must be matched exactly by a period in *pattern*. If **FNM_PATHNAME** is set, then a "leading" period is one that occurs either at the beginning of *string* or immediately following a slash; if it is not set, then a "leading" period is one that appears at the beginning of *string*. If **FNM_PERIOD** is not set, then **fnmatch()** places no special restrictions on matching a period.

If *string* matches *pattern*, **fnmatch()** returns zero. If it does not match, **fnmatch()** returns **FN_NOMATCH**. If an error occurs, **fnmatch()** returns a value other than zero or **FN_NOMATCH**.

See Also

libc, pnmatch(), string.h, wildcards

fnmatch.h — Header File

Constants used with function fnmatch()
#include <fnmatch.h>

The header file **fnmatch.h** defines manifest constants used with the function **fnmatch()**.

See Also

fnmatch(), header files

fopen() — STDIO Function (libc)

Open a stream for standard I/O
#include <stdio.h>
FILE *fopen (*name, type***)**
char ****name*, ****type***;**

fopen() allocates and initializes a **FILE** structure, or *stream*; opens or creates the file *name*; and returns a pointer to the structure for use by other STDIO routines. *name* refers to the file to be opened.

type is a string that consists of one or more of the characters "rwa", to indicate the mode of the string, as follows:

r Read; error if file not found
w Write; truncate if found, create if not found

a Append to end of file; no truncation, create if not found
r+ Read and write; no truncation, error if not found

w+ Write and read; truncate if found, create if not found
a+ Append and read; no truncation, create if not found

The modes that contain ‘a’ set the seek pointer to point at the end of the file; all other modes set it to point at the beginning of the file. Modes that contain ‘+’ both read and write; however, a program must call **fseek** or **rewind** before it switches from reading to writing or vice versa.

Example

This example copies **argv[1]** to **argv[2]** using STDIO routines. It demonstrates the functions **fopen()**, **fread()**, **fwrite()**, **fclose()**, and **feof()**.

```
#include <stdio.h>
#include <stdlib.h>
/* BUFSIZ is defined in stdio.h */
char buf[BUFSIZ];

void fatal(message)
char *message;
{
      fprintf(stderr, "copy: %s\n", message);
      exit(1);
}

main(argc, argv)
int argc; char *argv[];
{
      register FILE *ifp, *ofp;
      register unsigned int n;

      if (argc != 3)
            fatal("Usage: copy source destination");
      if ((ifp = fopen(argv[1], "r")) == NULL)
            fatal("cannot open input file");
      if ((ofp = fopen(argv[2], "w")) == NULL)
            fatal("cannot open output file");

      while ((n = fread(buf, 1, BUFSIZ, ifp)) != 0) {
            if (fwrite(buf, 1, n, ofp) != n)
                  fatal("write error");
      }

      if (!feof(ifp))
            fatal("read error");
      if (fclose(ifp) == EOF || fclose(ofp) == EOF)
            fatal("cannot close");
      exit(0);
}
```

See Also

fclose(), fdopen(), freopen(), libc
ANSI Standard, §7.9.5.3
POSIX Standard, §8.1

Diagnostics

fopen() returns NULL if it cannot allocate a **FILE** structure, if the *type* string is nonsense, or if the call to **open()** or **creat()** fails.

The header file **stdio.h** defines the manifest constant **FOPEN_MAX**, which sets the maximum number of **FILE** structures that you can allocate per program, including **stdin**, **stdout**, and **stderr**. For release 4.2, **FOPEN_MAX** is set to 60.

Notes

Many operating systems recognize a 'b' modifier to the *type* argument; this indicates that the file contains binary information, and lets the operating system handle "funny characters" correctly. COHERENT has no need of such a modifier, so if you append 'b' to *type*, it will be ignored. This modifier, however, is recognized by numerous other operating systems, including MS-DOS, OS/2, and GEMDOS. If you expect to port developed code to any of these operating systems, files should append the 'b' to *type*.

for — Command

Execute commands for tokens in list
for *name* **[in** *token ...***] do** *sequence* **done**

The shell command **for** controls a loop. It assigns to the variable *name* each successive *token* in the list, and then executes the commands in the given *sequence*. If the **in** clause is omitted, **for** successively assigns *name* the value of each positional parameter to the current script ('$@'). Because the shell recognizes a reserved word only as the unquoted first word of a command, both **do** and **done** must either occur unquoted at the start of a command or be preceded by ';'.

The shell commands **break** and **continue** may be used to alter control flow within a **for** loop.

The shell executes **for** directly.

See Also

break, commands, continue, ksh, sh

for — C Keyword

Control a loop
for(*initialization***;** *endcondition***;** *modification***)**

for is a C keyword that introduces a loop. It takes three arguments, which are separated by semicolons ';'. *initialization* is executed before the loop begins. *endcondition* describes the condition that ends the loop. *modification* is a statement that modifies *variable* to control the number of iterations of the loop. For example,

```
for (i=0; i<10; i++)
```

first sets the variable **i** to zero; then it declares that the loop will continue as long as **i** remains less than ten; and finally, increments **i** by one after every iteration of the loop. This ensures that the loop will iterate exactly ten times (from **i==0** through **i==9**). The statement

```
for(;;)
```

will loop until its execution is interrupted by a **break**, **goto**, or **return** statement. Also, either or both of *initialization* and *modification* may consist of multiple statements that are separated by commas. For example,

```
for (i=0, j=0; i<10; i++, j++)
```

initializes both *i* and *j*, and increments both with each iteration of the loop.

See Also

break, C keywords, continue, while
ANSI Standard, §6.6.5.3

fork() — System Call (libc)

Create a new process
#include <unistd.h>
fork()

In the COHERENT system, many processes may be active simultaneously. **fork()** creates a new process; the new process is a duplicate of the requesting process. In practice, the new process often issues a call to execute yet another new program.

The process that issues the **fork()** call is termed the *parent* process, and the newly forked process is termed the *child* process. **fork()** returns the process id of the newly created child to the parent process, and returns zero to the child process. The parent may call **wait()** to suspend itself until the child terminates.

The following parts of the environment of a process are exactly duplicated by a **fork()** call:

- Open files and their seek positions
- Current working and root directories
- The file creation mask
- The values of all signals
- The alarm clock setting
- Code, data, and stack segments

The system normally makes a fresh copy of the code, data, and stack segments for the child process. One advantage of *shared text* processes is that they do not need to copy the code segment. It is write protected, and therefore may be shared.

Example

For examples of how to use this call, see **msgget()**, **pipe()**, and **signal()**.

See Also

alarm(), execl(), exit(), libc, sh, umask(), unistd.h, wait()
POSIX Standard, §3.1.1

Diagnostics

fork() returns -1 on failure, which usually involves insufficient system resources. On successful calls, **fork()** returns zero to the child and the process id of the child to the parent.

fortune — Command

Print randomly selected, hopefully humorous, text
/usr/games/fortune [*file* **]**

fortune prints a message that is randomly selected from the contents of a text file. **fortune** reads *file* if it is named on the command line; otherwise, it reads the default file **/usr/games/lib/fortunes**.

Files

/usr/games/lib/fortunes — Default fortunes

See Also

commands

Notes

The fortunes included in **/usr/games/lib/fortunes** were selected mainly because they would not give offense to anyone. We encourage you to update this file with your favorite witticisms.

.forward — System Administration

Set a forwarding address for mail

The file **$HOME/.forward** lets you automatically redirect your incoming mail. You can redirect mail to one or more other users, who are located either on your local machine or on a remote site; or you can redirect your mail to one or more programs on your local machine, for further processing; or both. As you can see, this feature of the mail system included with COHERENT gives you great flexibility in processing your mail.

For example, you may wish to forward to another user any mail that is sent to the superuser root, so you can handle it immediately. (If you don't, it will languish in **root**'s mailbox until someone logs in as **root**, which may not happen for days.) To forward **root**'s mail to user **fred**, place the following line into file **/.forward**:

```
fred
```

Thereafter, whenever mail is sent to **root**, it will be forwarded automatically to user **fred**.

For another example, suppose that you are going on vacation, and you want your mail to be forwarded both to user **fred** and to user **anne**. To do so, insert the following instruction into file **$HOME/.forward**:

```
fred, anne
```

Thereafter, the route-mail program **rmail** will send a copy of every mail message you receive to **fred** and to **anne**. Please note that **rmail** will *not* insert a copy into your mailbox: if you forward your mail, you will not see it.

For another example, suppose that user **fred** has an account on each of two systems: one called **acme.com** and the other **zenith.com**. Suppose, further, that he logs into **acme.com** regularly, but he logs into **zenith.com** only now and again. This user probably would want to route any mail he receives on **zenith.com** to **acme.com**, so he will see it immediately. To do so, he would put the following instruction into file **$HOME/.forward** on **zenith.com**:

```
fred@acme.com
```

Thereafter, all mail sent to address **fred@zenith.com** will be forwarded automatically to **fred@acme.com**.

Please note that it is illegal to include in **.forward** the name of the user whose mail is being forwarded, because it causes an infinite loop in the mail system. For example, writing

```
fred, anne, root
```

into **root**'s **.forward** file causes any message sent to **root** to be forwarded to **fred**, **anne**, and **root**; the copy forwarded to **root** is again forwarded to **fred**, **anne**, and **root**; and so on, *ad infinitum*.

You can also embed the name of a program with your **.forward** file. All mail sent to your account will be handed to this program for processing. For example, the **elm** mailer includes a program called **filter**, which a user can program to read his mail and throw away unwanted messages. If you have installed **elm** onto your system, you can turn on **filter** by embedding the following command into file **$HOME/.forward**:

```
"|/usr/local/bin/filter"
```

Note that the command must be preceded by a '|' symbol; this is because **filter** receives its input from the standard input, which is the standard method for programs that filter text or mail. Note, too, that the entire command must be enclosed within quotation marks.

See Also

Administering COHERENT, mail [overview], smail

fpathconf() — System Call (libc)

Get a file variable by file descriptor
#include <unistd.h>
long fpathconf(*fd*, *fd***)**
int *fd*, *name***;**

fpathconf() returns the value of a limit or option associated with the open file whose the file descriptor is *fd*. *name* is the symbolic constant (defined in **<unistd.h>**) that represents the limit or option to be returned. The value that **fpathconf()** returns depends upon the type of file that *fd* identifies.

fpathconf() can return information about the following constants:

_PC_LINK_MAX
The maximum value of a file's link count. If *fd* identifies a directory, the value returned applies to the directory itself.

_PC_MAX_CANON
The number of bytes in a terminal's canonical input queue. Behavior is undefined if *fd* does not identify a terminal file.

_PC_MAX_INPUT
The number of bytes for which space will be available in a terminal's input queue. Behavior is undefined if *fd* does not identify a terminal file.

_PC_NAME_MAX
The number of bytes in a file name. The behavior is refined if *fd* does not identify a directory. The value returned applies to the file names within the directory.

_PC_PATH_MAX
The number of bytes in a path name. Behavior is undefined if *fd* does not identify a directory. If *fd* identifies the current working directory, **fpathconf()** returns the maximum length of a relative path name.

_PC_PIPE_BUF
The number of bytes that can be written atomically when writing to a pipe. If *fd* identifies a pipe or FIFO, the value returned applies to the FIFO itself. If *fd* identifies a directory, the value returned applies to any FIFOs that exist or can be created within that directory. If *fd* identifies any other type of file, behavior is

undefined.

_PC_CHOWN_RESTRICTED

chown() can be used only by a process with appropriate privileges, and only to change the group ID of a file to either that process's effective group ID or one of its supplementary group IDs. If *fd* identifies a directory, the value returned applies to any file, other than a directory, that exists or can be created within the directory.

_PC_NO_TRUNC

Path-name components longer than **NAME_MAX** generate an error. The behavior isundefined if *fd* does not identify a directory. The value returned applies to the file names within the directory.

_PC_VDISABLE

If this value is defined, terminal-special characters can be disabled. Behavior is undefined if *fd* does not identify a terminal file.

The value of the system limit or option that *name* specifies does not change during the lifetime of the calling process.

fpathconf() fails and returns -1 if *name* is not set to a recognized constant. It fails, returns -1, and sets **errno** to an appropriate value if either of the following is true:

- *fd* is not a valid file descriptor. **fpathconf()** sets **errno** to **EBADF**.
- *name* is an invalid value. **fpathconf()** sets **errno** to **EINVAL**.

See Also

libc, pathconf(), unistd.h
POSIX Standard, §5.7.1

fperr.h — Header File

Constants used with floating-point exception codes
#include <fperr.h>

fperr.h declares constants used by routines that handle floating-point exceptions. It also defines the error messages they use.

See Also

header files

fprintf() — STDIO Function (libc)

Print formatted output into file stream
#include <stdio.h>
int fprintf(*fp, format,* **[***arg1, ... argN***])**
FILE **fp***; char** **format***;**
[*data type***]** *arg1, ... argN***;**

fprintf() formats and prints a string. It resembles the function **printf()**, except that it writes its output into the stream pointed to by *fp*, instead of to the standard output.

fprintf() uses the *format* to specify an output format for *arg1* through *argN*.

See **printf()** for a description of **fprintf()**'s formatting codes.

If it wrote the formatted string correctly, **fprintf()** returns the number of characters written. Otherwise, it returns a negative number.

Example

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

See Also

libc, printf(), sprintf(), vfprintf()
ANSI Standard, §7.9.6.1
POSIX Standard, §8.1

Notes

Because C does not perform type checking, it is essential that an argument match its specification. For example, if the argument is a **long** and the specification is for a **short**, **fprintf()** will peel off the first word of that **long** and present it as an **short**.

fputc() — STDIO Function (libc)

Write character into file stream
#include <stdio.h>
int fputc(*c*, *fp*)
char *c*; FILE **fp*;

fputc() writes the character *c* into the file stream pointed to by *fp*. It returns *c* if *c* was written successfully.

Example

The following example uses **fputc** to write the contents of one file into another.

```
#include <stdio.h>

void fatal(message)
char *message;
{
      fprintf(stderr, "%s\n", message);
      exit(1);
}

main()
{
      FILE *fp, *fout;
      int ch;
      int infile[20];
      int outfile[20];

      printf("Enter name to copy: ");
      gets(infile);
      printf("Enter name of new file: ");
      gets(outfile);

      if ((fp = fopen(infile, "r")) == NULL)
            fatal("Cannot write input file");

      if ((fout = fopen(outfile, "w")) != NULL)
            fatal("Cannot write output file");

      while ((ch = fgetc(fp)) != EOF)
            fputc(ch, fout);
}
```

See Also

libc
ANSI Standard, §7.9.7.3
POSIX Standard, §8.1

Diagnostics

fputc() returns EOF when a write error occurs, e.g., when a disk runs out of space.

fputs() — STDIO Function (libc)

Write string into file stream
#include <stdio.h>
int fputs(*string*, *fp*)
char **string*; FILE **fp*;

fputs() writes *string* into the file stream pointed to by *fp*. Unlike its cousin **puts()**, it does not append a newline character to the end of *string*.

fputs() returns a nonnegative value on success and **EOF** if a write error occurs.

Example

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

See Also

libc, puts()
ANSI Standard, §7.9.7.4
POSIX Standard, §8.1

fputw() — STDIO Function (libc)

Write an integer into a stream
#include <stdio.h>
int fputw(*word***,** *fp***)**
int *word***; FILE** **fp***;**

fputw() writes *word* into the file stream pointed to by *fp*, and returns the value written.

Example

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

See Also

fgetw(), libc

Diagnostics

fputw() returns **EOF** when an error occurs. A call to **ferror()** or **feof()** may be needed to distinguish this value from a valid end-of-file signal.

fread() — STDIO Function (libc)

Read data from file stream
#include <stdio.h>
int fread(*buffer, size, n, fp***)**
char **buffer***; unsigned** *size, n***; FILE** **fp***;**

fread() reads *n* items, each being *size* bytes long, from file stream *fp* into *buffer*.

Example

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

See Also

fwrite(), libc
ANSI Standard, §7.9.8.1
POSIX Standard, §8.1

Diagnostics

fread() returns zero upon reading EOF or on error; otherwise, it returns the number of items read.

free() — General Function (libc)

Return dynamic memory to free memory pool
#include <stdlib.h>
void free(*ptr***) char** **ptr***;**

free() helps you manage the arena. It returns to the free memory pool memory that had previously been allocated by **malloc()**, **calloc()**, or **realloc()**. **free()** marks the block indicated by *ptr* as unused, so the **malloc()** search can coalesce it with contiguous free blocks. *ptr* must have been obtained from **malloc()**, **calloc()**, or **realloc()**.

Example

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

See Also

libc
ANSI Standard, §7.10.3.2
POSIX Standard, §8.1

Diagnostics

free() prints a message and calls **abort()** if it discovers that the arena has been corrupted. This most often occurs by storing data beyond the bounds of an allocated block.

freemem — Device

Device that indicates amount of memory that is free
/dev/freemem

/dev/freemem is the device from which you can read the system's free memory at any given moment. It has major device 0, the same as **/dev/null** and **/dev/cmos**; and minor number 12.

This non-portable device node is used exclusively for tracking the amount of free memory in the system. Its driver recognizes the system calls **open()**, **close()**, **read()**, and **ioctl()**, but not **write()**.

Example

The following program prints the amount of free memory in your system.

```
#include <fcntl.h>
#include <sys/null.h>
#include <stdlib.h>

main()
{
	FREEMEM freemem;
	int fm_fd;

	fm_fd = open("/dev/freemem", O_RDONLY);

	if (fm_fd >= 0) {
		ioctl (fm_fd, NLFREE, &freemem);
		close (fm_fd);
		printf ("Available memory: %d kilobytes\n", freemem.avail_mem);
		printf ("Free memory: %d kilobytes\n", freemem.free_mem);
	} else
		printf("Cannot open /dev/freemem\n");
}
```

See Also

device drivers, hmon, ioctl(), null

freopen() — STDIO Function (libc)

Open file stream for standard I/O
#include <stdio.h>
FILE *freopen (*name***,** *type***,** *fp***)**
char ****name*,** **type***; FILE ****fp***;**

freopen() reinitializes the file stream *fp*. It closes the file currently associated with it, opens or creates the file *name*, and returns a pointer to the structure for use by other STDIO routines. *name* names a file.

type is a string that consists of one or more of the characters "**rwa**" (for, respectively, read, write, and append) to indicate the mode of the stream. For further discussion of the *type* variable, see the entry for **fopen()**. **freopen()** differs from **fopen()** only in that *fp* specifies the stream to be used. Any stream previously associated with *fp* is closed by **fclose()**. **freopen()** is usually used to change the meaning of **stdin**, **stdout**, or **stderr**.

Example

This example, called **match.c**, looks in **argv[2]** for the pattern given by **argv[1]**. If the pattern is found, the line that contains the pattern is written into the file **argv[3]** or to **stdout**.

```
#include <stdio.h>
#define MAXLINE 128
char buffer[MAXLINE];
```

```
void fatal(message)
char *message;
{
      fprintf(stderr, "match: %s\n", message);
      exit(1);
}

main(argc,argv)
int argc; char *argv[];
{
      FILE *fpin, *fpout;

      if (argc != 3 && argc != 4)
            fatal("Usage: match pattern infile [outfile]");
      if ((fpin = fopen(argv[2], "r")) == NULL)
            fatal("Cannot open input file");

      fpout = stdout;
      if (argc == 4)
            if ((fpout = freopen(argv[3], "w", stdout)) == NULL)
                  fatal("Cannot open output file");

      while (fgets(buffer, MAXLINE, fpin) != NULL) {
            if (pnmatch(buffer, argv[1], 1))
                  fputs(buffer, stdout);
      }
      exit(0);
}
```

See Also

fopen(), libc
ANSI Standard, §7.9.5.4
POSIX Standard, §8.1

Diagnostics

freopen() returns NULL if the *type* string is nonsense or if the file cannot be opened. Currently, only 20 **FILE** structures can be allocated per program, including **stdin**, **stdout**, and **stderr**.

frexp() — General Function (libc)

Separate fraction and exponent
#include <math.h>
double frexp(*real***,** *ep***)**
double *real***; int** **ep***;**

frexp() breaks double-precision floating point numbers into fraction and exponent. It returns the fraction *m* of its *real* argument, such that 0.5 <= *m* < 1 or *m*=0, and stores the binary exponent *e* in the location pointed to by *ep*. These numbers satisfy the equation $real = m * 2e$.

Example

This example prompts for a number, then uses **frexp()** to break it into its fraction and exponent.

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

main()
{
      double real, fraction;
      int ep;

      char string[64];

      for (;;) {
            printf("Enter number: ");
            if (gets(string) == NULL)
                  break;
```

```
            fraction = frexp(real, &ep);
            printf("%lf is the fraction of %lf\n",
                  fraction, real);
            printf("%d is the binary exponent of %lf\n",
                  ep, real);
      }
}
```

See Also

atof(), ceil(), fabs(), floor(), ldexp(), libc, modf()
ANSI Standard, §7.5.4.3
POSIX Standard, §8.1

from — Command

Generate list of numbers, for use in loop
from *start* **to** *stop* **[by** *incr* **]**

from prints a list of integers on the standard output, one per line. It prints beginning with *start*, and then prints successive numbers incrementing by *incr* (default, one) the previous number. It continues until the generated value matches or exceeds *stop*. Each of *start*, *stop*, and optional *incr* is a decimal integer with an optional leading '-' sign.

Typical uses of **from** include generating a file of numbers and generating a loop index for the shell. The following example creates special files for eight terminal ports:

```
for i in `from 0 to 7`
do
      /etc/mknod /dev/hs0$i c 7 $i
done
```

See Also

commands, ksh, sh

Diagnostics

from prints an error message if the generated list is empty.

fscanf() — STDIO Function (libc)

Format input from a file stream
#include <stdio.h>
int fscanf(*fp, format, arg1, … argN***)**
FILE **fp***; char** **format***;**
[*data type***]** **arg1, …* **argN***;**

fscanf() reads the file stream pointed to by *fp*, and uses the string *format* to format the arguments *arg1* through *argN*, each of which must point to a variable of the appropriate data type.

fscanf() returns either the number of arguments matched, or EOF if no arguments matched.

For more information on **fscanf()**'s conversion codes, see **scanf()**.

Example

The following example uses **fprintf()** to write some data into a file, and then reads it back using **fscanf()**.

```
#include <stdio.h>

main ()
{
      FILE *fp;
      char let[4];

      /* open file into write/read mode */
      if ((fp = fopen("tmpfile", "wr")) == NULL) {
            printf("Cannot open 'tmpfile'\n");
            exit(1);
      }
```

```
	/* write a string of chars into file */
	fprintf(fp, "1234");

	/* move file pointer back to beginning of file */
	rewind(fp);

	/* read and print data from file */
	fscanf(fp, "%c %c %c %c",
		&let[0], &let[1], &let[2], &let[3]);
	printf("%c %c %c %c\n",
		let[3], let[2], let[1], let[0]);
}
```

See Also

libc, scanf(), sscanf()
ANSI Standard, §7.9.6.2
POSIX Standard, §8.1

Notes

Because C does not perform type checking, it is essential that an argument match its specification. For that reason, **fscanf()** is best used only to process data that you are certain are in the correct data format, such as data previously written out with **fprintf()**.

fsck — Command

Check and repair file systems interactively
/etc/fsck [**-fnqsSy**] [**-t** *tempfile*] [*filesystem ...*]

fsck checks and interactively repairs file systems. If all is well, **fsck** merely prints the number of files used, the number of blocks used, and the number of blocks that are free. If the file system is found to be inconsistent in one of the aspects outlined below, **fsck** asks whether it should fix the inconsistency and waits for you to reply **yes** or **no**.

The following file system aspects are checked for consistency by **fsck**:

- If a block is claimed by more than one i-node, by an i-node and the free list, or more than once in the free list.
- Whether an i-node or the free list claims blocks beyond the file system's range.
- Link counts that are incorrect.
- Whether the directory size is not aligned for 16 bytes.
- Whether the i-node format is correct.
- Whether any blocks are not accounted for.
- Whether a file points to an unallocated i-node.
- Whether a file's i-node number is out of range.
- Whether the super block refers to more than 65,536 i-nodes.
- Whether the super block assigned more blocks to the i-nodes than the system contains.
- Whether the format of the free block list is correct.
- Whether the counts of the total free blocks and the free i-nodes are correct.

fsck prints a warning message when a file name is null, has an embedded slash '/', is not null-padded, or if '.' or '..' files do not have the correct i-node numbers.

When **fsck** repairs a file system, any file that is orphaned (that is, allocated but not referenced) is deleted if it is empty, or copied to a directory called **lost+found**, with its i-node number as its name. The directory **lost+found** must exist in the root of the file system being checked before **fsck** is executed, and it must have room for new entries without requiring that new blocks be allocated.

fsck recognizes the following options:

-f Fast check. **fsck** only checks whether a block has been claimed by more than one i-node, by an i-node and the free list, or more than once in the free list. If necessary, **fsck** will reconstruct the free list.

-n No option: a default reply of **no** is given to all of **fsck**'s questions.

-q Quiet option: run quietly. **fsck** automatically removes all unreferenced pipes, and automatically fixes list counts in the super block and the free list. File-name warning messages are suppressed, but **fsck** still prints the number of files used, the number of blocks used, and the number of blocks that remain free.

-s Sort the free lists, both free blocks and free i-nodes, based on the interleave number. This is useful in reducing fragmentation of a file system. This option ignores mounted file systems.

-S Same as **-s**, except that it also works on mounted file systems. Not recommended for the faint of heart.

-t Name the temporary file used by **fsck**.

-y Yes option: a default reply of **yes** is given to all of **fsck**'s questions.

If you do not name a file system in **fsck**'s command line, **fsck** checks the file systems named in the file **/etc/checklist**.

Files

/etc/checklist

See Also

clri, commands, icheck, ncheck, ram, sync, umount

Diagnostics

The following describes **fsck**'s error messages and questions. The error messages fall into two categories: *warnings*, which describe something possibly wrong with a file; and *fatals*, which indicate that something has gone wrong with a file system, or with **fsck** itself, with which **fsck** cannot cope. Each question describes the condition in question; here, it is followed by advice on what is probably the correct response.

Bad action in virtual system *(fatal)*
Bad block *number*, i-number = *number (warning)*
Number Bad blocks in Free List *(warning)*

Bad/Dup blocks in *i-node type file name* (Clear i-node) [yes/no] *(question)*
: The given i-node contains bad or duplicately referenced blocks. You are asked if you would like to clear the i-node completely. If you answer yes, then the file will be lost forever.

Bad entry in block *number* in directory *name*/*i-node (warning)*

Bad Free List (SALVAGE) [yes/no] *(question)*
: **fsck** is asking if you want it to salvage the free list automatically. This is almost certainly a good thing to do.

Bad or Dup blocks in *directory*/*file* (Remove) [yes/no] *(question)*
: The given file's i-node references bad or duplicately referenced blocks. **fsck** is asking if you wish to remove *file* from the directory.

Bad Super Block: *number (warning)*
Number Blocks missing *(warning)*

***** BOOT Coherent (NO SYNC!) ***** *(message)*
: Do as the message says: reboot COHERENT *without* running the command **sync**.

Cannot close Ram Disk Close /dev/rram1close *(fatal)*
Cannot create temp file *name (fatal)*
Cannot open Ram Disk Close /dev/rram1close *(fatal)*
Cannot open read/write Ram Disk /dev/rram1 *(fatal)*

Can not Read: Blk num: *number* (CONTINUE) [yes/no] *(question)*
: The given action could not be performed. If you choose to not continue, **fsck** will abort. If you choose to continue, the results may be unpredictable.

Can not Seek: Blk num: *number* (CONTINUE) [yes/no] *(question)*
The given action could not be performed. If you choose to not continue, **fsck** will abort. If you choose to continue, the results may be unpredictable.

Can not Write: Blk num: *number* (CONTINUE) [yes/no] *(question)*
The given action could not be performed. If you choose to not continue, **fsck** will abort. If you choose to continue, the results may be unpredictable.

Can't access ram disk /dev/rram1, use the -t option *(fatal)*
Can't malloc memory, phase 2 *(fatal)*
Can't malloc space for interleave table. Free-block list is not rebuilt. *(warning)*
Can't open: *file system (warning)*
Can't open checklist file: /etc/checklist *(fatal)*
Can't stat: *file system (warning)*
Can't stat temp file *name (fatal)*

Count = *count*, should be *count* (Adjust) [yes/no] *(question)*
The given i-node claims to have a different number of links than was actually found in the file system. You are asked if you wish to adjust the count found in the i-node. If you answer yes, then **fsck** will correct the i-node count.

Directory Misaligned i-number = *number (warning)*
Dir i-number = *number* connected. Parent was i-number = *number (warning)*
Dir i-number = *number* connected. It has bad/dup blocks. *(warning)*
Dir i-number = *number* connected. It has no .. entry. *(warning)*

Dup/Bad blocks in root i-node (Continue) [yes/no] *(question)*
The root i-node has bad or duplicate blocks. This may require a guru to fix properly. **fsck** is asking whether you want it to continue. If not, then **fsck** will abort.

Dup Block *number*, i-number = *number (warning)*
Number Dup blocks in Free List *(warning)*

DUP Table Overflow (Continue) [yes/no] *(question)*
The table of duplicately referenced disk blocks has overflowed. You can continue with the **fsck** (as best as it is able), or abort.

Embedded slashes in entry in block *number* in directory *name/i-node (warning)*
Error seeking tmp file *(fatal)*
Error writing tmp file *(fatal)*
Error writing to tmp file *(fatal)*

Excessive Bad Blocks i-number = *number* (Continue) [yes/no] *(question)*
The specified i-node references an excessive number of bad blocks. You can continue with the **fsck** (at the next i-node), or abort.

Excessive Dup Blocks i-number = *number* (Continue) [yes/no] *(question)*
The specified i-node references an excessive number of duplicate blocks. You can continue with the **fsck** (at the next i-node), or abort.

Excessive *bad/dup* blocks in free list (Continue) [yes/no] *(question)*
This indicates that there are excessive bad or duplicately referenced blocks in the free list off of the superblock. This is a very bad condition. You should choose to continue, which will fall to phase 6 to salvage the free list. If you answer no, then **fsck** will abort.

Expect roughly *number* missing blocks next time fsck is run as a result of i-nodes being cleared. *(message)*

file is not a block or character device; OK? [yes/no]: *(question)*
You are attempting to **fsck** a file that is not a block or character device. If you are certain it is a file system, then answer yes to continue.

File System Read-Only (NO WRITE) *(fatal)*
***** File System *system* was modified ***** *(message)*
Number files *number* blocks *number* free *(message)*
Fixblock error. *(fatal)*

Free Block count wrong in superblock. (FIX) [yes/no] *(question)*
The free block count in the superblock is incorrect. You should allow **fsck** to repair it unless you are a guru and have reason to believe that **fsck** should not use the redundancy in the file system (via all previously reported messages) to repair this crucial piece of data in the superblock.

Free i-node count wrong in superblock. (FIX) [yes/no] *(question)*
The free i-node count in the superblock is incorrect. You should allow **fsck** to repair it unless you are a guru and have reason to believe that **fsck** should not use the redundancy in the file system (via all previously reported messages) to repair this crucial piece of data in the superblock.

Inconsistent . entry in block *number* in directory *name/i-node (warning)*
Inconsistent .. entry in block *number* in directory *name/i-node (warning)*
i-number = *number* is in a bad inode block. *(warning)*

I-number is out of range I=*file name* (Remove) [yes/no] *(question)*
file has an i-node number that is out of range. **fsck** is asking if you wish to remove the stated file (which, after all, does not exist).

I-node *number* is a multiply referenced directory i-node. *(warning)*
internal linktable corruption. *(fatal)*
Invalid interleave factors in superblock. Default free-block list spacing assumed. *(warning)*
Invalid Response *(fatal)*
Link count discrepancy in *i-node type file name*
file system mounted on *point* as of *time (message)*
Name too long. *(warning)*
Non null padded entry in block *number* in directory *name/i-node (warning)*
Null name entry in block *number* in directory *name/i-node (warning)*

Out of Range Block number: *number* (CONTINUE) [yes/no] *(question)*
The given action could not be performed. If you choose to not continue, **fsck** will abort. If you choose to continue, the results may be unpredictable.

Possible Directory Size Error i-number = *number (warning)*
Possible File Size Error i-number = *number (warning)*
Possible file system on ram disk /dev/rram1, use the -t option *(fatal)*
Ram disk close /dev/rram1close not mknoded properly *(fatal)*
Ram disk /dev/rram1 not mknoded properly *(fatal)*

Root i-node is not a directory (FIX) [yes/no] *(question)*
The root i-node must be a directory. **fsck** is asking whether you wish to fix this. If not, then **fsck** will abort.

Root i-node is unallocated. Terminating *(fatal)*
Size check: fsize *blocks* isize *first non-i-node block (warning)*
Sorry. No lost+found directory. *(warning)*
Sorry. No space in lost+found directory. *(warning)*
Temp File must not be on file system to fsck *(fatal)*
Too many file systems in checklist file: /etc/checklist *(fatal)*
Too large free block count *(warning)*
Too large free i-node count *(warning)*
Too many links in i-node *number (fatal)*
Tried to checkpath i-node *number* which is not dir. *(fatal)*

Unallocated *file* (Remove) [yes/no] *(question)*
file's i-node is unallocated. **fsck** is asking if you wish to remove the stated file (which, after all, does not exist).

Unknown File Type i-number = *number* (Clear) [yes/no]: *(question)*
The mode field in the specified i-node is unknown. If you wish, you can clear the named i-node.

file system unmounted. Last mounted on *point*. *(message)*

Unref Dir *name* (Reconnect) [yes/no] *(question)*
The given directory's i-node is unreferenced. You are asked if you would like to reconnect the stated directory. If you answer yes, then the directory will be reconnected in directory **/lost+found** in the given file system. If not, it will remain unreferenced and you will be asked later if you would like to remove it.

Unref *i-node type file name* (Reconnect) [yes/no] *(question)*
The given i-node is unreferenced. **fsck** is asking if you wish to reconnect it to the stated file. If you answer yes, then the file will be reconnected in directory **/lost+found** in the given file system. If not, it will remain unreferenced and you will be asked later if you would like to remove it.

Unref *i-node type file name* (Clear i-node) [yes/no] *(question)*
The given i-node is unreferenced. **fsck** asks if you wish to clear the i-node completely. If you answer yes, the file is lost forever. You have already decided not to reconnect it, so there seems to be no reason to keep it anyway.

Notes

The correction of file systems almost always involves the destruction of data.

You should run **fsck** only when the COHERENT system is in single-user mode.

fsck cannot modify a file system during its work. This rule was adopted to prevent **fsck** from attempting to modify a corrupt file system, and so making matters worse. However, this means that **fsck** cannot change the size of directory **lost+found**. Thus, if more files are detached from the file system than **lost+found** can hold, **fsck** must delete them outright. If you are running an application that uses large numbers of transient files (e.g., a news system), you should increase the size of **lost+found** so that it has a fighting chance of holding all detached files that **fsck** finds. To do so, use the command **/etc/mklost+found**. For details, see its entry in the Lexicon.

fseek() — STDIO Function (libc)

Seek on file stream
#include <stdio.h>
int fseek(*fp, where, how***)**
FILE **fp*; **long** *where*; **int** *how*;

fseek() changes where the next read or write operation will occur within the file stream *fp*. It handles any effects the seek routine might have had on the internal buffering strategies of the system. The arguments *where* and *how* specify the desired seek position. *where* indicates the new seek position in the file. It is measured from the start of the file if *how* equals **SEEK_SET** (zero), from the current seek position if *how* equals **SEEK_CUR** (one), and from the end of the file if *how* equals two **SEEK_END** (two).

fseek() differs from its cousin **lseek()** in that **lseek()** is a COHERENT system call and takes a file number, whereas **fseek()** is a STDIO function and takes a **FILE** pointer.

Example

This example opens file **argv[1]** and prints its last **argv[2]** characters (default, 100). It demonstrates the functions **fseek()**, **ftell()**, and **fclose()**.

```
#include <stdio.h>
extern long atol();

void fatal(message)
char *message;
{
      fprintf(stderr, "tail: %s\n", message);
      exit(1);
}

main(argc, argv)
int argc; char *argv[];
{
      register FILE *ifp;
      register int c;
      long nchars, size;

      if (argc < 2 || argc > 3)
            fatal("Usage: tail file [ nchars ]");
      nchars = (argc == 3) ? atol(argv[2]) : 100L;
```

```
	if ((ifp = fopen(argv[1], "r")) == NULL)
		fatal("cannot open input file");
	/* Seek to end */
	if (fseek(ifp, 0L, 2) == -1)
		fatal("seek error");

	/* Find current size */
	size = ftell(ifp);
	size = (size < nchars) ? 0L : size - nchars;

	/* Seek to point */
	if (fseek(ifp, size, 0) == -1)
		fatal("seek error");
	while ((c = getc(ifp)) != EOF)
		/* Copy rest to stdout */
		putchar(c);
	if (fclose(ifp) == EOF)
		fatal("cannot close");
	exit(0);
}
```

See Also

fsetpos(), ftell(), libc, lseek()
ANSI Standard, §7.9.9.2
POSIX Standard, §8.1

Diagnostics

For any diagnostic error, **fseek()** returns -1; otherwise, it returns zero. If **fseek()** goes beyond the end of the file, it will not return an error message until the corresponding read or write is performed.

fsetpos() — STDIO Function (libc)

Set file-position indicator
#include <stdio.h>
int
fsetpos(*fp***,** *position***)**
FILE **fp***; fpos_t** **position***;**

fsetpos() resets the file-position indicator. *fp* points to the file stream whose indicator is being reset. *position* is a value that had been returned by an earlier call to **fgetpos()**. It is of type **fpos_t**, which is defined in the header **stdio.h**.

Like the related function **fseek()**, **fsetpos()** clears the end-of-file indicator and undoes the effects of a previous call to **ungetc()**. The next operation on *fp* may read or write data.

fsetpos() returns zero if all goes well. If an error occurred, it returns nonzero and sets **errno** to an appropriate value.

Example

For an example of this function, see **fgetpos()**.

See Also

fgetpos(), fseek(), ftell(), libc, rewind()
ANSI Standard, §7.9.9.3

Notes

The ANSI Standard designed **fsetpos()** to be used with files whose file position cannot be represented within a **long**. Under COHERENT, it behaves the same as **fseek()**.

If you wish to use **fsetpos()**, you should first call the function **fgetpos()** to obtain value of the file-position indicator.

You can also use the functions **ftell()** and **fset()**, respectively, to read and set the file-position indicator. However, code that uses these function may not be portable to operating systems other than COHERENT or UNIX.

fstat() — System Call (libc)

Find attributes of an open file
#include <sys/stat.h>
int fstat(*fd, statptr***)**
int *fd***; struct stat ****statptr***;**

fstat() examines the attributes of an open file. *fd* is the descriptor of the open file or pipe you wish to examine. *statptr* points to a structure of type **stat**, which is defined in the header file **<stat.h>**; **fstat()** writes into it the attributes of the file or pipe to which *fd* points, including protection information, file type, and file size.

fstat() returns zero if all goes well. If an error occurs (e.g., *fd* is not found or *statptr* is invalid), it returns -1.

Example

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

See Also

chmod(), chown(), libc, ls, open(), stat(), stat.h
POSIX Standard, §5.6.2

Notes

fstat() differs from the related function **stat()** mainly in that it accesses a file through its descriptor, which was returned by a successful call to **open()**, whereas **stat()** takes the file's path name and opens the file itself before it checks its status.

fstatfs() — System Call (libc)

Get information about a file system
#include <sys/types.h>
#include <sys/statfs.h>
int fstatfs (*filedes***,** *buffer***,** *length***,** *fstype***)**
int *filedes***;**
struct statfs ****buffer*;**
int *length***,** *fstype***;**

The COHERENT system call **fstatfs()** 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 **fstatfs()** can write its output. Always set this to **sizeof(struct statfs)**.

filedes and *fstype* identify the file system. If the file system is unmounted, then *filedes* should give the file descriptor for the device by which the file system is accessed, as returned by a call to **creat()**, **dup()**, **open()**, or **pipe()**; and *fstype* contains the type of the file system. If the file system is mounted, then *filedes* should give the file descriptor of a file on the file system in question, and *fstype* must be set to zero.

fstatfs() returns zero if all went well. If something went wrong, it returns -1 and sets **errno** to an appropriate value.

See Also

libc, mkfs, statfs(), statfs.h, types.h, ustat()

ft — Device Driver

Floppy-tape driver
/dev/ft

The device driver **ft** supports floppy-tape drives. It has major number 4. Minor-number assignments are documented in the header file **/usr/include/sys/ft.h**.

ft works with QIC-40 and QIC-80 drives from Colorado, Archive, Mountain, and Summit. It offers the following features:

- It uses the bad-block bitmap that is written into the first two 32-kilobyte segments of tape at format time.
- It uses standard QIC-40/QIC-80 Reed-Solomon error-correcting code (ECC). This technique uses three of every 32 blocks for error checking. A tape block is one kilobyte long.
- It supports no-rewind-on-close. This feature permits you to concatenate several archives onto a single tape cartridge.
- It performs auto-configuration for users who do not know if their drives use soft select A or soft select B, or hard select on unit 0, 1, 2, or 3, with manual override.
- It lets you configure the size of the tape buffer, from 64 through 4,064 kilobytes.
- It reads from and writes to buffer space rather than going to tape whenever possible.
- It works with partially formatted tapes. Some formatting utilities let you select the number of tape tracks to format, in case you do not want to take the time to format an entire cartridge.
- It recognizes both 205-foot and 307.5-foot tapes.
- It works with the COHERENT command **tape** with the following arguments: **rewind**, **retension**, **seek**, **status**, and **tell**.

Please note that release 1.0 of **ft** has the following limitations:

- It does not format tapes. For now, we suggest that you buy pre-formatted tapes, or use formatting utilities available under other operating systems.
- It does not support the QIC-80 formats for MS-DOS or UUCP file systems on tape. These features do not need to be part of the device driver, and can be implemented by user-level applications.
- It does not perform data compression, as documented in QIC-122. Other forms of data compression are presently available under COHERENT, such as the **-z** option supported by the tape-archive command **gtar**.
- The device driver is character-only: there is no corresponding block device for floppy tape.
- It does not support 1,100-foot tapes. Although the QIC-80 standard mentions this length, it is not in common use.
- You cannot access a floppy-disk drive from COHERENT while a floppy-tape drive is in use. Likewise, if a floppy disk is in use — for example, if it is mounted — you cannot access the floppy-tape drive.
- Although a QIC-80 drive can read a tape that was formatted for QIC-40, it cannot write to such a tape. The cartridge must be reformatted for QIC-80 before a QIC-80 drive can write to it.

See Also

device drivers, fd, ftbad, gnucpio, gtar, tape

Notes

ft reports any error that may affect integrity of the data. If the same block number appears repeatedly in **ft**'s warning messages, it is a problem on the tape and the block should be in the bad block list. Because the Reed-Solomon ECC used in **ft** allows the physical medium to spoil up to three of every 32 one-kilobyte blocks yet recover all data, your data set may still be recoverable despite these errors; but you should consider using the command **ftbad** to add such blocks to your cartridge's list of bad blocks before you again write data onto that cartridge.

The message:

```
Get Reference Burst Failed
```

can occur if you attempt to back up to an unformatted tape, or one who's format is unrecognizable. If a backup fails with this message, try using another, formatted cartridge.

Systems with a very slow CPU (e.g., a 16-megahertz 80386SX) may have trouble running **ft** in multi-user mode. The reason is that floppy-tape hardware does not have much intelligence built into it, so the driver must consume many CPU cycles. In such instances, we suggest that you back up your system while in single-user mode (which is a good idea in any case).

ftbad — Command

Manipulate bad-block list on a floppy-tape cartridge
ftbad [-rw] [*device***]**

The command **ftbad** lets you manipulate the list of bad blocks on a floppy-tape cartridge. It recognizes the following options:

-r Read the list of bad blocks from floppy-tape cartridge, and write them to the standard-output device. The output will appear something like the following:

```
557
1033
89640
```

-w Read a list of bad blocks from the standard-input device, and write it onto the floppy-tape cartridge.

device
The floppy-tape device to manipulate. If you do not name a device on **ftbad**'s command line, by default it uses **/dev/ft**, which rewinds the tape upon close. For a list of tape devices that you can use the Lexicon entry for **tape**.

Example

To modify the bad block list for a cartridge, do the following:

- First, use the command:

```
ftbad -r > badlist
```

This reads the list of bad blocks and writes it into file **badlist**.

- Second, edit **badlist**. Each line in this file will name only one bad block, in decimal notation.
- Finally, write the edited list back onto the tape cartridge with the command:

```
ftbad -w < badlist
```

See Also

commands, ft, tape

Notes

Do not change the bad block list of a tape that contains data you wish to retrieve. You should use **ftbad** only when you see repeated I/O errors at the same block on a tape and wish to mark that block as being bad before you reuse the tape. *Caveat utilitor!*

ftell() — STDIO Function (libc)

Return current position of file pointer
#include <stdio.h>
long ftell(*fp***) FILE ***fp***;**

ftell() returns the current position of the seek pointer. Like its cousin **fseek()**, **ftell()** takes into account any buffering that is associated with the stream *fp*.

Example

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

See Also

fgetpos(), fseek(), libc, lseek(), rewind()
ANSI Standard, §7.9.9.4
POSIX Standard, §8.1

ftime() — System Call (libc)

Get the current time from the operating system
#include <sys/timeb.h>
int ftime(*tbp***)**
struct timeb ***tbp*;**

ftime() fills the structure **timeb**, which is pointed to *tbp*, with COHERENT's representation of the current time. Header file **timeb.h** defines **timeb** as follows:

```
struct timeb {
      time_t time;
      unsigned short millitm;
      short timezone;
      short dstflag;
}
```

The member **time** is the number of seconds since January 1, 1970, 0h00m00s GMT. **millitm** is a count of milliseconds. **timezone** and **dstflag** are obsolete; they have been replaced by the environmental variable **TIMEZONE**.

ftime() does not return a meaningful value.

See Also

date, libc, time, timeb.h, TIMEZONE, types.h

Notes

The ANSI Standard eliminates **ftime()** from the set of standard time functions. COHERENT includes it only to support existing software. Users are well advised to modify their time routines to eliminate **ftime()**.

ftok() — General Function (libc)

Generate keys for interprocess communication
#include <sys/types.h>
#include <sys/ipc.h>
key_t ftok(*filename***,** *procid***)**
char ***filename*;**
char *procid***;**

The COHERENT system implements three methods by which one process can communicate with another: semaphores, messages, and shared memory. In each case, a process must use a key of type **key_t** (which is defined in header file **<sys/types.h>**) to identify itself.

One problem is that each process generates its own key, by its own method. Therefore, two processes could independently generate the same key, which could create serious problems for interprocess communication.

The function **ftok()** generates keys for processes that perform interprocess communication. *filename* is the full path name of a file. This can be the full path name of the file in which the program resides on disk. The file named in *filename* must exist and be accessible for the system call **stat()**, or **ftok()** will fail. *procid* is a one-character identifier with which this process distinguishes itself from all other processes that are pegged to *filename*. How a process generates *procid* is up to the program itself.

For example, the program **myproc** can generate a unique key for itself with the call:

```
key_t mykey;
mykey = ftok("/usr/bin/myproc", 'A');
```

Note the following caveats:

- Because **ftok()** generates its key from a file's i-node major and minor numbers rather than its name, it generates the same key for two files that are linked. For example, if files **/usr/henry/foo** and **/usr/henry/bar** are linked to each other, then the calls

  ```
  ftok("/usr/henry/foo", 'A');
  ```

  and

  ```
  ftok("/usr/henry/bar", 'A');
  ```

will generate the same key.

- If the file named by *filename* is destroyed and then recreated, the call to **ftok()** generates a different key than it did before *filename* was destroyed.
- If the file named by *filename* does not exist, **ftok()** returns **(key_t) -1**.

Example

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

See Also

ipc.h, libc, msgget(), semget(), shmget()

function — Definition

A **function** is the C term for a portion of code that is named, can be invoked by name, and that performs a task. Many functions can accept data in the form of arguments, modify the data, and return a value to the statement that invoked it.

See Also

data types, library, portability, Programming COHERENT

fwrite() — STDIO Function (libc)

Write into file stream
#include <stdio.h>
int fwrite(*buffer, size, n, fp***)**
char **buffer***; unsigned** *size, n***; FILE** **fp***;**

fwrite() writes *n* items, each of *size* bytes, from *buffer* into the file stream pointed to by *fp*.

Example

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

See Also

fread(), libc
ANSI Standard, §7.9.8.2
POSIX Standard, §8.1

Diagnostics

fwrite() normally returns the number of items written. If an error occurs, the returned value will not be the same as *n*.

fwtable — Command

Build font-width table
fwtable [-ptv] [*infile* **[** *outfile* **]]**

For the typesetting program **troff** to use a font, it must know the width of each character in the font, and it must know how to tell the printer to select the font. All of this information is built into a *font-width table*, which **troff** reads when you run it.

COHERENT comes with font-width tables for a selected set of fonts: for a handful of scalable fonts that are included with standard PostScript cartridges, for a few bit-mapped fonts, and for some fonts that are built into the Hewlett-Packard LaserJet III. For a list of the font-width tables that are included with COHERENT, and for further information on how to manage fonts, see the Lexicon entry for **troff**.

The command **fwtable** can read a font, and build a new font-width table for it. It reads the font information from *infile* (or the standard input) and writes a font-width table for the font to *outfile* (or the standard output). It can understand fonts in the following formats:

- PCL (Printer Control Language) bitmap fonts, which have the suffix **.usp**.
- Fonts that are built into the Hewlett-Packard LaserJet III and IV, which have the suffix **.tfm**.
- AFM (Adobe Font Metric) descriptions of PostScript fonts, which have the suffix **.afm**.

fwtable recognizes the following command-line options:

-p *infile* is an AFM (Adobe Font Metric) description for a PostScript font. By default, **fwtable** assumes that *infile* is a a bit-mapped soft font (that is, a font with the suffix **.usp**).

Please note that if the AFM font you will be using is downloadable rather than built into a cartridge, you must also use the command **PSfont** to "cook" that font's **.pfb** file into downloadable form. For more information, see the Lexicon entry **PSfont**.

-t *infile* is a Hewlett-Packard **.tfm** file, which describes a font that is built into the Hewlett-Packard LaserJet III, rather than a bit-mapped soft font.

-v Print a brief font description to the standard error file.

Files

/usr/lib/roff/troff_pcl/fwt/ — Directory for PCL font-width tables
/usr/lib/roff/troff_ps/fwt/ — Directory for PostScript font-width tables

See Also

commands, hpr, PSfont, troff

Notes

fwtable does not understand Intellifont scalable fonts, or TrueType fonts.