COHERENT

System Manual

Table of Contents

Introduction

This manual contains a concise description of each facility available to a programmer in the COHERENT™ operating system. It is a reference work rather than a tutorial; its descriptions use terminology unfamiliar to the beginner. The *Introduction to the COHERENT System* is the document a novice should read first.

The companion volume to this manual is the *COHERENT Command Manual*, which describes each command available to a COHERENT system user. The descriptions in this manual often refer to commands described in the *COHERENT Command Manual*. It should be used in conjunction with this manual.

Almost all programming on the COHERENT system is written in the C programming language. A familiarity with C is essential to understanding most of the information in this manual. The basic C reference work is *The C Programming Language*, by Kernighan and Ritchie (Prentice-Hall, 1978). Section **as** provides a brief summary of assembly language calling sequences.

This manual consists of many sections, ordered alphabetically. Each section describes a single feature or a few related features. New material is inserted into this manual as new features become available on the system.

The features described in this manual fall into several distinct categories, as indicated by the heading of each section. The headings are: Convention, Device Driver, File Format, **libc** Library, **libm** Library, Maintenance, STDIO Library, and System Call. Sections immediately following this introduction list which sections fall under each heading.

A comprehensive **Index** makes it easy to find any information desired. It lists the name of each feature included in this manual as well as describing the function of each feature.

For most programmers, the most useful information in this manual is in the sections describing *libraries*. The standard I/O library (STDIO) provides simple but powerful buffered input and output facilities. The standard C library (**libc**) provides routines to perform common tasks which are not part of the C language itself, such as string manipulation. The mathematics library (**libm**) provides mathematical functions, such as trigonometric functions. The **cc** command loads the **libc** library automatically when it compiles a

C program. Programs using the STDIO library should **#include <stdio.h>**. Programs using the **libm** library should **#include <math.h>** and should use the **-lm** option to the **cc** or **ld** command to obtain the mathematics library.

Programmers who require access to the primitive building blocks of the COHERENT system will find the descriptions of system calls useful. Most programmers should use the libraries rather than using system calls directly.

Convention sections describe COHERENT system conventions, such as the ASCII character set.

Device driver sections describe specific COHERENT devices. File format sections describe the format of specific COHERENT files or file types. This specialized information is useful primarily for programmers involved in system development or modification.

The maintenance sections describe commands generally used by the system administrator rather than by most users. Unlike the maintenance commands described in the *COHERENT Command Manual*, the commands in this manual are usually executed only during system startup. The *COHERENT Administrator's Guide* gives additional maintenance information.

The USAGE lines of each section describe how a feature is used. For file formats, the **USAGE** names the appropriate header file. For maintenance commands, it describes commands in the format of the *COHERENT Command Manual*. For library routines and system calls, the **USAGE** describes the C calling sequence of each routine. Boldface type indicates characters which are literally part of the calling sequence. Italic type indicates parts of the calling sequence which are actually replaced by other text when the routine is invoked. The call

```
acct("/usr/adm/acct");
```

is an example of the **USAGE** of the system call **acct**, with the character string **"/usr/adm/acct"** as the *file* argument.

Many sections include a **FILES** subsection describing the names of files used. References to header files such as **<stdio.h>** are enclosed in angle brackets to indicate that the header file resides in the *include* directory, which is usually **/usr/include**.

Most sections include cross references in a **SEE ALSO** subsection. Many refer to other sections of this manual. Others refer to the *COHERENT Command Manual*, which provides detailed information on COHERENT commands. Some refer to separate related documents.

The COHERENT system is available on a wide variety of computers. In almost all cases, the operation of the features described in this manual is identical on different machines; the system looks the same, regardless of which processor it actually uses. However, because of hardware limitations, a few features may not be implemented on some systems. A few sections of this manual note that the features described do not exist on all COHERENT systems.

The information in this manual is also available online, through the **man** command, on most COHERENT systems. For example,

```
man abort
```

prints the section of this manual describing the **abort** system call.

Conventions

The following sections of this manual describe system conventions, indicated with the heading "Convention".

as
ascii
environ
man
ms

Device Drivers

The following sections of this manual describe device drivers, indicated with the heading "Device Driver".

ct
mem
null
tape

Most device drivers handle a particular hardware device. **mknod** creates a special file to provice access to the driver; device special files generally reside in directory /**dev**. **sload** and **suload** load and unload a device driver. **ioctl** controls device-dependent characteristics of a driver.

Because of hardware restrictions, the COHERENT system does not support loadable device drivers on systems based on the 8086 or 8088 processors (such as the IBM Personal Computer). The /**drv** directory and the **sload** and **suload** calls do not exist on such systems.

File Formats

The following sections of this manual describe file formats, indicated with the heading "File Format".

acct.h
ar.h
canon.h
core
dir.h
group
l.out.h
mtab.h
passwd
ttys
utmp.h

libc Library

The following sections of this manual describe parts of the **libc** library, indicated with the heading "**libc** Library".

abort
abs
assert
atof
crypt
ctime
ctype
ecvt
end
exit
frexp
getenv
getgrent
getlogin
getpass
getpw
getpwent
getwd
l3tol
malloc
mktemp
mtype
nlist
perror
pnmatch
qsort
rand
setjmp
signame
sleep
string
swab
system
ttyname

libc Library

The **cc** command loads the **libc** library automatically when it compiles a C program. The **libc** library resides in the archive file **/lib/libc.a**.

libm Library

The following sections of this manual describe parts of the **libm** library, indicated with the heading "**libm** Library".

floor
hypot
j0
log
sin
sinh

Programs using the **libm** library should **#include <math.h>**, and the **cc** or **ld** command should use the **-lm** option to obtain the library. The **libm** library resides in the archive file **/lib/libm.a**.

When an error occurs, these routines set the external variable **errno** to either **EDOM** (argument out of domain of function) or **ERANGE** (returned value too large for floating point representation of the machine). The **math.h** header file defines these values, and section **errno** describes errors in detail. The actual returned value may be a very large number or may be indistinguishable from normal returned values.

Section **mp** of this manual describes the **libmp** library for multiple precision arithmetic. Programs using the **libmp** library should **#include <mprec.h>**, and the **cc** or **ld** command should use the **-lmp** option to obtain the library. The **libmp** library resides in the archive file **/usr/lib/libmp.a**.

Maintenance

The following sections of this manual provide system maintenance information, indicated with the heading ''Maintenance''.

cron
getty
init
lpd
swap
sysgen
update

These sections are of particular interest to the system administrator. The *COHERENT Command Manual* and the *COHERENT Administrator's Guide* give additional information on system maintenance.

STDIO Library

The following sections of this manual describe parts of the standard I/O library STDIO, indicated with the heading ''STDIO Library''.

fclose
ferror
fflush
fileno
fopen
fread
fseek
fwrite
getc
gets
getw
popen
printf
putc
puts
putw
scanf
setbuf
ungetc

Programs using the STDIO library should **#include <stdio.h>**. These routines perform efficient buffered I/O operations. A buffered file is called a *stream*, and has type **FILE ***. When an error occurs, these routines generally return the value **NULL** for a pointer or **EOF** for an **int**. **NULL** and **EOF** are defined in the **stdio.h** header file. **stdio.h** also defines the standard input, standard output, and standard error streams, called **stdin, stdout** and **stderr**.

System Calls

The following sections of this manual describe system calls, indicated with the heading "System Call".

access
acct
alarm
brk
chdir
chmod
chown
chroot
close
creat
dup
errno
exec
exit
fork
ftime
getpid
getuid
ioctl
kill
link
lock
lseek
mknod
mount
open
pause
pipe
profil
ptrace
read
setuid
signal
sload
stat
stime
sync

System Calls

times
umask
unlink
utime
wait
write

When an error occurs in a system call, the call normally returns −1 and sets the external variable **errno** to a code identifying the error. Section **errno** describes error numbers.

libc Library

NAME
abort – terminate process with core dump

USAGE
abort()

DESCRIPTION
abort terminates a process with a core dump, creating a file called **core**. It is normally invoked in situations that ''should not happen'' in a program. For example, the memory allocator **malloc** invokes **abort** when it discovers a corrupt storage arena.

Where possible, **abort** executes a machine instruction which causes the processor to trap. Implementation on specific machines is described below. If the signal associated with the trap is caught or ignored, the dump will not be produced.

FILES
core

SEE ALSO
exit, malloc, signal
COHERENT Command Manual: **db**

DIAGNOSTICS
On the PDP-11, **abort** executes an IOT instruction; ''IOT trap—core dumped'' is printed.

On the Zilog Z8000, **abort** executes a halt instruction; ''privileged instruction—core dumped'' is printed.

On the Intel 8086, **abort** executes an invalid system call; ''bad system call—core dumped'' is printed.

libc Library

NAME
abs – absolute value of an integer

USAGE
abs(*n*)
int *n*;

DESCRIPTION
abs returns the absolute value of the integer *n*.

SEE ALSO
floor

NOTES
On two's complement machines, **abs** of the most negative integer is itself.

System Call

NAME
access – test access to a file

USAGE
#include <access.h>

access(*file, how***)**
char ***file*;**
int *how***;**

DESCRIPTION
access tests whether the given *file* may be accessed in a particular way, without actually performing an **open**, **creat**, or **exec** call. The *how* argument specifies types of accessibility to be tested. The parameters **AREAD**, **AWRITE**, **AEXEC**, and **AAPPND** test access for reading, writing, execution, and appending, respectively. The header file **access.h** defines these values, which may be logically combined to produce the *how* argument. If *how* is 0, **access** tests only the existence of *file* and the ability to search all directories leading to it.

access uses the *real* user id and *real* group id (rather than the *effective* user id and *effective* group id), so set user id programs can use it.

For directories, the above permissions are interpreted differently. For example, directories can never be explicitly opened for writing, but write permission on a directory as indicated by **access** means the ability to create or unlink files in the directory. The parameters **ALIST**, **ADEL**, **ASRCH**, and **ACREAT** test access for listing a directory, deleting a directory entry, searching a directory, and creating a directory entry, respectively.

FILES
<access.h>

SEE ALSO
creat, exec, open, stat

DIAGNOSTICS
access returns 0 if *file* exists and is accessible as requested. It returns –1 if *file* does not exist or is inaccessible under the given modes.

System Call

NAME

acct – enable/disable process accounting

USAGE

acct(*file*)
**char ****file*;

DESCRIPTION

acct enables or disables process accounting. A nonnull *file* argument enables accounting; the specified *file* must exist. When enabled, the system appends a raw accounting data record in the format described by **acct.h** to *file* as each process terminates.

A *file* argument of **NULL** disables process accounting.

acct is restricted to the superuser.

SEE ALSO

acct.h, exit, times
COHERENT Command Manual: **ac, sa**

DIAGNOSTICS

Successful calls return the value 0. **acct** returns −1 for errors, such as nonexistent *file* or invocation by a user other than the superuser.

NOTES

The system writes accounting records for a process only when it exits. Processes that never terminate and processes running at the time of a system crash do not produce accounting information.

NAME
acct.h – process accounting file format

USAGE
#include <acct.h>

DESCRIPTION
After **acct** enables process accounting, the system writes raw process accounting information into an accounting file as each process terminates. Each entry in the accounting file, normally **/usr/adm/acct**, has the following form, as defined in the **acct.h** header file:

```
struct  acct    {
        char    ac_comm[10];
        comp_t  ac_utime;
        comp_t  ac_stime;
        comp_t  ac_etime;
        time_t  ac_btime;
        short   ac_uid;
        short   ac_gid;
        short   ac_mem;
        comp_t  ac_io;
        dev_t   ac_tty;
        char    ac_flag;
};

/* Bits from ac_flag */
#define AFORK   01      /* has done fork, but not exec */
#define ASU     02      /* has used superuser privileges */
```

Every time a process does an **exec** call, the contents of **ac_comm** are replaced with the first 10 characters of the file name. The fields **ac_utime** and **ac_stime** represent CPU time used in the user program and in the system, respectively. **ac_etime** represents the elapsed time since the process started running, while **ac_btime** is the time the process started. The effective user id and group id are **ac_uid** and **ac_gid**. **ac_mem** gives the average memory usage of the process. **ac_io** gives the number of blocks of input-output. **ac_tty** gives the controlling typewriter device major and minor numbers.

For some of the above times, the **acct** structure uses the special representation **comp_t,** defined in the **types.h** header file. It is a floating point representation with 3 bits of base 8 exponent and 13 bits of fraction, so it fits in a **short** integer.

FILES
<acct.h>
/usr/adm/acct

SEE ALSO
acct
COHERENT Command Manual: **accton, sa**

System Call

NAME
alarm – set a timer

USAGE
alarm(*n***)**
unsigned *n***;**

DESCRIPTION
alarm sets a timer associated with the requesting process to go off in *n* seconds. After *n* seconds, the system sends the signal **SIGALRM** to the process. An argument of 0 turns off the alarm timer.

By default, the receipt of the **SIGALRM** signal terminates the process. However, it may be caught or ignored by using **signal**. Because of scheduling variation and the one second granularity, the action of **alarm** is only predictable to within one second.

alarm is useful for such things as timeouts. For example, the login process on a dial-in port might hang up the line after a sufficient elapsed time with no user response.

alarm returns the previous alarm value, which represents the time remaining from the previous call. Time remaining is superseded by the new alarm value.

SEE ALSO
signal, sleep

NAME
ar.h – archive file format

USAGE
#include <ar.h>

DESCRIPTION
An *archive* is a single file built from a number of constituent files and maintained by the **ar** command. Usually an archive is a library of object files used by the **ld** command.

All archives start with a magic number **ARMAG**, which identifies the file as an archive. The members of the archive follow the magic number, each preceded by an **ar_hdr** structure, as defined in the **ar.h** header file:

```
#define DIRSIZ  14                /* from <dir.h> */
#define ARMAG   0177535           /* magic number */

struct ar_hdr {
        char    ar_name[DIRSIZ];/* member name */
        time_t  ar_date;          /* time inserted */
        short   ar_gid;           /* group owner */
        short   ar_uid;           /* user owner */
        short   ar_mode;          /* file mode */
        size_t  ar_size;          /* file size */
};
```

The structure at the head of each member is immediately followed by **ar_size** bytes, which are the data of the file.

To enhance the performance of **ld**, the **ranlib** command provides a random library facility. **ranlib** produces archives which contain a special entry named **__.SYMDEF** at the beginning. The header file **ar.h** contains structure definitions and other information describing the format of its tables.

All integer members of the structure (everything but the **ar_name**) are in canonical form to ease portability. See **canon.h** for more information.

FILES
<ar.h>

SEE ALSO
canon.h
COHERENT Command Manual: **ar, ld, ranlib**

NAME
as – assembly language calling sequences

USAGE
as /usr/include/sys.s *file ...*

DESCRIPTION
The descriptions of routines in this manual use the C language. The following C call and corresponding assembly language illustrate the assembly language calling sequence for a C routine.

```
struct  item {
        int     x_type;
        char    *x_name[8];
}       items[30];

n = fread((char *)items, sizeof(items[0]), 30, fp);
```

Assembly language for the PDP-11:

```
mov     $fp_, -(sp)
mov     $30, -(sp)
mov     $10, -(sp)
mov     $items_, -(sp)
jsr     pc, fread_
add     $8, sp
mov     r0, n_
```

Assembly language for the Z8002:

```
push    (r15), fp_
push    (r15), $30
push    (r15), $10
push    (r15), $items_
call    fread_
inc     r15, $8
ld      n_, r1
```

Convention

Assembly language for the Z8001:

```
pushl   (rr14), fp_
push    (rr14), $30
push    (rr14), $34
ldl     rr0, $items_
pushl   (rr14), rr0
calr    fread_
inc     r15, $12
ld      n_, r1
```

Assembly language for the (small model) i8086:

```
push    fp_
mov     ax,$30
push    ax
mov     ax,$10
push    ax
mov     ax,$items_
push    ax
call    fread_
add     sp, $8
mov     n_, ax
```

For more detailed information, such as passing and returning **long** and **double** types to routines, the user should consult the assembler output of the C compiler (obtained with the **−S** option to the **cc** command).

It is recommended that assembly language programs call routines from the C library, rather than coding system calls directly. For those who need to know, the format of a system call conforms exactly to the C function calling conventions for a particular machine. The include file **<sys.s>** gives the system call numbers, and may be included with source files to the assembler.

On the PDP-11, system calls are implemented as trap instructions. The arguments are at the correct place on the stack after the C compiler has pushed them for the interface routine. The system ignores the pushed **pc** for the call of the routine. Following the C calling conventions, the return value is in **r0** for an **int** and the **r0-r1** pair for a **long**. The location **errno** is at the top of the stack

Convention

segment, at virtual address 0177776. For example, the interface routine for the **write** system call is:

```
write_:
        sys     4
        rts     pc
```

On the Zilog Z8001 and Z8002 processors, system calls conform to C calling conventions, with the arguments on the stack and the return value in **r1** for an **int** and in **rr0** for a **long**. The location of **errno** is 0xFFFE, on the stack. For example, the interface routine for the **write** system call is:

```
write_:
        sys     4
        ret
```

For the Intel 8086 and 8088 processors, as well as the IBM Personal Computer, the calling sequences for the system calls is as for C, with arguments on the stack. The return value is in register **ax** for an **int** and in the pair **dx-ax** for a **long**. The location of **errno** is 0x0002. For example, the interface routine for the **write** system call is:

```
write_:
        sys     4
        ret
```

FILES
<sys.s>

SEE ALSO
COHERENT Command Manual: **as, cc**

Convention

NAME

ascii – ASCII character table

DESCRIPTION

The file **/usr/pub/ascii** gives a table of codes in the ISO 7-bit (ASCII) character set. For each character the following table gives numeric values in octal, decimal and hexadecimal, followed by the conventional symbolic name (2 or 3 upper-case letters) or graphic representation of the character.

Convention

000	000	00	NUL	040	032	20	SP	100	064	40	@	140	096	60	
001	001	01	SOH	041	033	21	!	101	065	41	A	141	097	61	a
002	002	02	STX	042	034	22	"	102	066	42	B	142	098	62	b
003	003	03	ETX	043	035	23	#	103	067	43	C	143	099	63	c
004	004	04	EOT	044	036	24	$	104	068	44	D	144	100	64	d
005	005	05	ENQ	045	037	25	%	105	069	45	E	145	101	65	e
006	006	06	ACK	046	038	26	&	106	070	46	F	146	102	66	f
007	007	07	BEL	047	039	27	'	107	071	47	G	147	103	67	g
010	008	08	BS	050	040	28	(	110	072	48	H	150	104	68	h
011	009	09	HT	051	041	29	)	111	073	49	I	151	105	69	i
012	010	0A	LF	052	042	2A	*	112	074	4A	J	152	106	6A	j
013	011	0B	VT	053	043	2B	+	113	075	4B	K	153	107	6B	k
014	012	0C	FF	054	044	2C	,	114	076	4C	L	154	108	6C	l
015	013	0D	CR	055	045	2D	-	115	077	4D	M	155	109	6D	m
016	014	0E	SO	056	046	2E	.	116	078	4E	N	156	110	6E	n
017	015	0F	SI	057	047	2F	/	117	079	4F	O	157	111	6F	o
020	016	10	DLE	060	048	30	0	120	080	50	P	160	112	70	p
021	017	11	DC1	061	049	31	1	121	081	51	Q	161	113	71	q
022	018	12	DC2	062	050	32	2	122	082	52	R	162	114	72	r
023	019	13	DC3	063	051	33	3	123	083	53	S	163	115	73	s
024	020	14	DC4	064	052	34	4	124	084	54	T	164	116	74	t
025	021	15	NAK	065	053	35	5	125	085	55	U	165	117	75	u
026	022	16	SYN	066	054	36	6	126	086	56	V	166	118	76	v
027	023	17	ETB	067	055	37	7	127	087	57	W	167	119	77	w
030	024	18	CAN	070	056	38	8	130	088	58	X	170	120	78	x
031	025	19	EM	071	057	39	9	131	089	59	Y	171	121	79	y
032	026	1A	SUB	072	058	3A	:	132	090	5A	Z	172	122	7A	z
033	027	1B	ESC	073	059	3B	;	133	091	5B	[	173	123	7B	{
034	028	1C	FS	074	060	3C	<	134	092	5C	\	174	124	7C	|
035	029	1D	GS	075	061	3D	=	135	093	5D	]	175	125	7D	}
036	030	1E	RS	076	062	3E	>	136	094	5E		176	126	7E	
037	031	1F	US	077	063	3F	?	137	095	5F	_	177	127	7F	DEL

FILES
/usr/pub/ascii

libc Library

NAME
assert – check assertion at runtime

USAGE
#include <assert.h>

assert(*condition***)**

DESCRIPTION
assert checks the validity of the given *condition*, as a debugging aid. If the *condition* is false (0), **assert** prints an error message and the program exits. **assert** is usually used to detect situations in a program that should "never happen".

The **–DNDEBUG** argument to **cc** disables all checking of assertions.

Assertions may also be made at compile time, during the preprocessing phase, by the **#assert** *condition* directive. If *condition* (an expression involving constants of the form acceptable to **#if**) is false (0), **cc** prints a diagnostic during compilation.

FILES
<assert.h>

SEE ALSO
exit
COHERENT Command Manual: **cc**

DIAGNOSTICS
assert prints

file: *line*: `assert(`*condition*`) failed`

when *condition* is not true at line number *line* of source file *file*. Because **assert** is a macro using **printf**, it expands into an illegal C statement if the *condition* includes double quotes ('"').

libc Library

NAME
atof, atoi, atol – convert strings to numbers

USAGE
double
atof(*string***)**
char ****string*;**

int
atoi(*string***)**
char ****string*;**

long
atol(*string***)**
char ****string*;**

DESCRIPTION
These routines convert the argument *string* to binary representations of numbers of various types. In all cases, leading blanks and tabs are ignored. Each stops scanning when it encounters any inappropriate character and returns the resulting number.

atoi and **atol** each read an integer number and return a result of type **int** or **long**, respectively. The *string* may contain an optional leading sign and any number of decimal digits.

atof reads a floating point number and returns a result of type **double**. The *string* may contain an optional leading sign, any number of decimal digits, and possibly one decimal point. It may be terminated by an optional exponent given by an **'e'** or **'E'**, an optional sign, and any number of decimal digits.

SEE ALSO
printf, scanf

NOTES
No overflow checks are performed.

Maintenance

NAME
boottime – time of last system boot

DESCRIPTION
/etc/boottime is an empty file maintained by the **init** process and the **date** command. The modification time of **boottime** is the time of the last system boot. The modification time of a file is accessible with the command '**ls –l**' or the **stat** or **fstat** system call.

FILES
/etc/boottime

SEE ALSO
init
COHERENT Command Manual: **date, mount**

DIAGNOSTICS
Commands which depend on **/etc/boottime** may malfunction if the date is not set correctly. For instance, the **mount** command depends on the relative modification times of **/etc/boottime** and **/etc/mtab** to detect whether the mount table has been invalidated by a system boot. If the date is set sufficiently far into the past, the mount table may appear to be valid when in fact it is not.

System Call

NAME
brk, sbrk – change size of data area

USAGE
brk(*addr***)**
char ****addr*;**

char *
sbrk(*incr***)**
int *incr***;**

DESCRIPTION
The *break* is the lowest address above the data area of a process. **brk** sets the break to the given *addr*, possibly rounding up by some machine-dependent factor. **brk** returns 0 on success, −1 on failure.

sbrk changes the break by *incr* bytes (possibly rounding up) and returns the start of the new data area. **sbrk** returns **NULL** in case of error.

SEE ALSO
end, exec, malloc

DIAGNOSTICS
sbrk returns −1 if the request fails. Both routines set **errno** to **ENOMEM** if the request fails.

File Format

NAME
canon.h – portable layout of binary data

USAGE
#include <canon.h>
#include <types.h>

canshort(*s*)
short *s*;

canint(*i*)
int *i*;

canlong(*l*)
long *l*;

canvaddr(*v*)
vaddr_t *v*;

cansize(*s*)
size_t *s*;

candaddr(*d*)
daddr_t *d*;

cantime(*t*)
time_t *t*;

candev(*d*)
dev_t *d*;

canino(*i*)
ino_t *i*;

DESCRIPTION
The layout of binary data varies among machines. For example, the byte order of a 16-bit word on the PDP-11 is lo-byte.hi-byte, while the byte order on the Z8000 is hi-byte.lo-byte.

File Format

To ensure portability of file systems across machines with differing byte order, the COHERENT system uses a canonical layout of binary data. Data not in primary memory (e.g. on disk or tape or communications line) must conform to this canonical layout. To insulate programs from the details of the difference between 'natural' and canonical layout, the COHERENT system provides procedures to convert between layouts.

Each procedure takes a single *l-value* of the indicated type and converts it in place; there is no return value. The argument should not have side-effects. Each procedure is its own inverse. Several procedures are targeted specifically for elements of file systems.

The file formats currently containing canonical binary data and the commands (see the *COHERENT Command Manual*) dealing with them are:

format	*commands*
ar.h	**ar, ld, ranlib**
dir.h	**ls, tar**
l.out.h	**as, cc, db, ld, nm, size, strip**

Any program which manipulates binary data on files should perform canonical conversion immediately upon input and immediately before output. The following fragment of code from **df** should be instructive.

File Format

```
#include <stdio.h>
#include <canon.h>
#include <filsys.h>
char    superb[BSIZE];
        .
        .
        .
df(fs)
char *fs;
{
        register struct filsys *sbp = superb;
        FILE *fp;
        daddr_t nfree;

        if ((fp = fopen(fs, "r")) == NULL) {
                perror(fs);
                return (1);
        }
        fseek(fp, (long)BSIZE, 0);
        if (fread(superb, sizeof superb, 1, fp) != 1) {
                fprintf(stderr, "%s: read error\n", fs);
                return (1);
        }
        candaddr(sbp->s_tfree);
        candaddr(sbp->s_fsize);
        canshort(sbp->s_isize);
        nfree = sbp->s_tfree;
        if (nfree > sbp->s_fsize-sbp->s_isize  ||  nfree < 0) {
                fprintf(stderr, "%s: bad free count\n", fs);
                return (1);
        }
        printf("%s: %D\n", fs, nfree);
        fclose(fp);
        return (0);
}
```

FILES
<canon.h>

SEE ALSO
ar.h, dir.h, l.out.h

System Call

NAME
chdir – change working directory

USAGE
chdir(*directory***)**
char ****directory*;**

DESCRIPTION
The *working directory* (or *current directory*) is the directory from which file name searching begins if a pathname does not begin with '/'. By convention, the working directory has the name '.'. **chdir** changes the working directory to the *directory* specified.

SEE ALSO
chroot
COHERENT Command Manual: **cd**

DIAGNOSTICS
chdir returns 0 for successful calls. It returns −1 on errors, such as *directory* nonexistent, not a directory, or not searchable.

System Call

NAME
chmod – change file protection modes

USAGE
#include <sys/stat.h>

chmod(*file, mode***)**
char **file*;
int *mode*;

DESCRIPTION
chmod sets the mode bits of the given *file*. The mode bits include protection bits, the set user id bit, the set group id bit and the sticky bit.

The *mode* argument is constructed from the logical OR of the following, which are defined symbolically in the **stat.h** header file:

04000	set user id
02000	set group id
01000	save text (sticky bit)
00400	read permission for owner
00200	write permission for owner
00100	execute permission for owner
00040	read permission for members of owner's group
00020	write permission for members of owner's group
00010	execute permission for members of owner's group
00004	read permission for other users
00002	write permission for other users
00001	execute permission for other users

For directories, some protection bits have a different meaning: write permission means files may be created and removed, while execute permission means the directory may be searched.

The save text bit, or sticky bit, is a flag to the system when it executes a shared form of a load module. After the system runs the program, it leaves shared segments on the swap storage device to facilitate faster subsequent reinvocation of the program. The setting of this bit is restricted to the superuser (to control depletion of swap space which might result from overuse).

Only the owner of a file or the superuser may change its mode.

FILES
<sys/stat.h>

SEE ALSO
creat, stat
COHERENT Command Manual: **chmod**

DIAGNOSTICS
chmod returns −1 for errors, such as *file* being nonexistent or the invoker being neither the owner nor superuser.

System Call

NAME
chown – change ownership of a file

USAGE
chown(*file, uid, gid***)**
char **file*;
short *uid, gid*;

DESCRIPTION
chown changes the owner of *file* to user id *uid* and group id *gid*.

To change only the user id without changing the group id, **stat** should be used to determine the value of *gid* to pass to **chown**.

Because granting the ordinary user the ability to change the ownership of files might circumvent file space quotas or accounting based upon file ownership, **chown** is restricted to the superuser.

SEE ALSO
chmod, passwd, stat
COHERENT Command Manual: **chown**

DIAGNOSTICS
chown returns −1 for errors, such as nonexistent *file* or the caller not being the superuser.

System Call

NAME
chroot – change process's root directory

USAGE
chroot(*directory***)**
char **directory*;

DESCRIPTION
The *root directory* is the directory from which file name searches commence when a pathname begins with '/'. **chroot** changes the root directory to *directory* for the requesting process and all of its children.

Because of security problems, **chroot** is restricted to the superuser. It is sometimes useful for a system administrator; for example, to test a new system environment which resides on a mounted file system.

SEE ALSO
chdir, fork

DIAGNOSTICS
chroot returns 0 for a successful call. It returns −1 on errors, such as the caller not being the superuser or the *directory* being nonexistent or not a directory.

System Call

NAME
close – close a file

USAGE
close(*fd***)**
int *fd***;**

DESCRIPTION
close closes a file identified by the file descriptor *fd,* as returned by **creat, dup, open,** or **pipe**. **close** frees the associated file descriptor. Since each process has a limited number of open files, programs processing many files should **close** files when possible. Closing a file may have side effects, especially with device files.

The system closes all open files automatically when a process exits.

SEE ALSO
creat, dup, exit, fclose, fflush, open, pipe

DIAGNOSTICS
close returns −1 if an error occurs, such as a bad file descriptor. Otherwise, it returns 0.

File Format

NAME
core – core dump file format

USAGE
#include <sys/uproc.h>

DESCRIPTION
When a process terminates abnormally because of a process fault or because it receives an asynchronous signal from another process, the system tries to write a memory dump of the process into a file called **core**. This file contains an image of the process code and data segments and the system description segment for the process. The following list gives the segment types and the symbolic names of their locations in the file:

SIUSERP	user process description segment
SISTACK	user stack segment
SISTEXT	shared text segment
SIPTEXT	private text segment
SISDATA	shared data segment
SIPDATA	private data segment

Dumps do not necessarily contain all of the above segments. Neither shared text nor shared data segments are dumped. They are write protected in memory and the load module running when the dump occurred contains shared segment data.

The best way for a program (such as a debugger) to read the **core** file is to first read the user process description segment, which is always at the front and has a fixed size. It should be read into an area of **UPASIZE** bytes, but referenced with structured type **UPROC** (somewhat smaller than **UPASIZE** because of the system stack, which contains the user registers and other information in fixed places).

The **sr_segl** member of the **UPROC** structure is a list of segment reference descriptors which contain the virtual address and length of each segment, corresponding exactly to its size in the dump. **NUSEG** segments are possible; flag **SRFDUMP** in the **sr_flag** field indicates a segment was dumped. Using the above method, the entire file may be used to reference program data and code at the time of the dump.

Other information found in the user process structure may be pertinent; the header file **sys/uproc.h** contains more information.

FILES
core
<sys/uproc.h>

SEE ALSO
l.out.h, signal
COHERENT Command Manual: **db, kill, wait**

DIAGNOSTICS
COHERENT will not write **core** if it already exists as a non-ordinary file or if there is more than one link to it. The 0200 bit in the status returned to the parent process by **wait** indicates a successful dump.

System Call

NAME
creat – create/truncate a file

USAGE
#include <sys/stat.h>

creat(*file, mode*)
**char ****file*;
int *mode*;

DESCRIPTION
creat creates a new *file* or truncates an existing *file*. It returns a file descriptor which identifies *file* for subsequent system calls.

If *file* existed previously, its previous contents are lost. In this case, **creat** ignores the specified *mode*; the mode of the *file* remains unchanged.

If *file* did not exist previously, **creat** uses the *mode* argument to determine the mode of the new *file*. For a full definition of file modes, see **chmod** or the header file **stat.h. creat** masks the *mode* argument with the current **umask**, so it is common practice to create files with the maximal mode desirable.

FILES
<sys/stat.h>

SEE ALSO
chmod, mknod, open, stat, umask
COHERENT Command Manual: **sh**

DIAGNOSTICS
If the call is successful, **creat** returns a file descriptor. It returns −1 if it could not create the file, typically for reasons such as protection violations or insufficient file system resources.

NAME
cron – execute commands periodically

USAGE
/etc/cron&

DESCRIPTION
cron is a daemon which executes commands at preset times. The commands and their scheduled execution times are kept in the file **/usr/lib/crontab**.

Once each minute **cron** searches through **crontab**. For each command stored there, **cron** compares the current time with the scheduled execution time and executes the command if the times match. When it finishes the search, **cron** sleeps until the next minute. Since it never exits, **cron** should be executed only once (customarily by **/etc/rc**).

crontab consists of lines separated by newlines. Each line consists of fields separated by white space (tabs or blanks). The first five fields describe the scheduled execution time of the command. In order, they represent:

minute (0 – 59),
hour (0 – 23),
day of the month (1 – 31),
month of the year (1 – 12), and
day of the week (0 – 6, 0 meaning Sunday).

Each field may contain a single integer in the appropriate range, a pair of integers separated by a '–' (meaning all integers between the two, inclusive), an asterisk '*' (meaning all legal values), or a comma-separated list of the above forms. The remainder of the line gives the command to be executed at the given time.

cron recognizes three special characters and escape sequences in **crontab**. If a command contains the percent character '%', **cron** executes only the portion up to the first '%' as a command and passes the remainder to the command as its standard input. **cron** translates any percent characters in the remainder to newlines. The special interpretation of '%' can be prevented by preceding it with a backslash, '\%'. Finally, **cron** removes the sequence '**\newline**' from the text before passing it to the shell **sh**; this can be used to make an entry in **crontab** more readable.

Maintenance

cron is designed for commands that need to be executed regularly. One-shot commands should be handled by the **at** command.

FILES
/usr/lib/crontab for stored commands

SEE ALSO
init
COHERENT Command Manual: **at**
COHERENT Administrator's Guide

libc Library

NAME
crypt, encrypt, setkey – encryption/decryption using DES algorithm

USAGE
char *
crypt(*key*, *extra***)**
**char ****key*, ******extra*;

encrypt(*bits*, *flag***)**
**char ****bits*;
int *flag*;

setkey(*key***)**
**char ****key*;

DESCRIPTION
These routines implement a variant on the encoding scheme defined by the National Bureau of Standards Data Encryption Standard (DES). **crypt** produces encrypted passwords which are verified by comparing the encrypted clear text against an original encryption. To prevent brute force methods from violating security, some internal tables are modified so that current hardware implementations of DES will not be compatible with the result of **crypt**.

The *key* argument to **crypt** is an ASCII string containing the user's password. The *extra* argument is a string of two additional characters, stored in the password file with the encrypted password. The *extra* characters are used to modify the DES algorithm, as described above. Each character must come from an alphabet of 64 symbols, consisting of upper-case and lower-case letters, digits, period '.' and slash '/'. **crypt** returns a string using the same 64 character alphabet as the *extra* argument. Its first two characters are the *extra* argument and the rest is the encrypted password.

setkey and **encrypt** provide a more direct access to DES encryption software for other uses. **setkey** takes an array of 64 characters. These characters are binary values (0 or 1) which define a 64-bit key. As in the DES standard, only the first 7 of every 8 bits are significant to the key. Once the key has been defined, **encrypt** encodes or decodes a string of 8 bytes, or 64 bits, at a time. The *bits* argument is an array of characters each containing one bit, as for **setkey**. **encrypt** transforms the array of bits into the encrypted or decrypted result. The *flag* argument specifies the direction; 0 means encryption, anything else means decryption.

SEE ALSO
getpass
COHERENT Command Manual: **login, passwd**

NOTES
The encrypted result returned by **crypt** is static and is overwritten by each invocation.

Device Driver

NAME
ct – controlling terminal driver

DESCRIPTION
For each process, the controlling terminal driver **/dev/tty** is an interface to the appropriate terminal driver. COHERENT passes any input-output call (e.g. **close, ioctl, open,** or **write)** on this special file directly to the controlling terminal device for the requesting process.

Normally, the controlling terminal is the default standard input, output, and error device. This is not the case for daemon processes started by the initial process.

FILES
/dev/tty

SEE ALSO
init

DIAGNOSTICS
When a call finds no valid controlling terminal for a process, it returns a value of –1 and sets **errno** to **ENXIO.**

libc Library

NAME

asctime, ctime, gmtime, localtime, settz, timezone, tzname – time and date conversion

USAGE

#include <time.h>
#include <types.h>

char *
asctime(*tmp*)
**struct tm ****tmp*;

char *
ctime(*timep*)
**time_t ****timep*;

struct tm *
gmtime(*timep*)
**time_t ****timep*;

struct tm *
localtime(*timep*)
**time_t ****timep*;

void
settz( )

long
timezone

char *
tzname[2][16]

DESCRIPTION

The internal form of COHERENT time is a long integer containing the number of seconds since Midnight, January 1, 1970. These routines convert this format to more accessible forms.

ctime takes a pointer to the internal time **time_t** (as defined in **<types.h>**) and returns a fixed-length string in the form

"Thu Mar 14 11:12:14 1982\n"

localtime and **gmtime** convert the time to a more intermediate form. Each returns a pointer to a **tm** structure, as defined in **time.h**:

```
struct tm {
        int     tm_sec;     /* Second (0-59) */
        int     tm_min;     /* Minute (0-59) */
        int     tm_hour;    /* Hour (0-23) */
        int     tm_mday;    /* Day of month (1-31) */
        int     tm_mon;     /* Month of year (0-11) */
        int     tm_year;    /* Year-1900 */
        int     tm_wday;    /* Weekday (Sunday=0) */
        int     tm_yday;    /* Day of year (0-365) */
        int     tm_isdst;   /* Daylight Saving Time */
};
```

gmtime returns Greenwich Mean Time (GMT), while **localtime** returns the local time (possibly including daylight saving time conversion), as indicated by **ftime.** The daylight saving time flag indicates whether daylight saving time is in effect, not simply whether it will be in effect during some part of the year.

asctime returns an ASCII string containing the time and date for the structure referenced by *tmp*. In fact, **ctime** is implemented as a call to **localtime** followed by a call to **asctime.**

settz searches for the environmental parameter **TIMEZONE** which specifies local time zone information in the format specified below. If **TIMEZONE** is set, **settz** initializes **timezone** and **tzname** accordingly.

timezone is an external variable containing the number of seconds to be subtracted from GMT to obtain local standard time. **tzname[0]** and **tzname[1]** are external character arrays which contain the names of the local standard time zone and the local daylight saving time zone, respectively. If **TIMEZONE** is not set, **timezone** defaults to 0, **tzname[0]** defaults to **"GMT"**, and **tzname[1]** defaults to the empty string.

The environmental parameter **TIMEZONE** contains colon-separated fields which specify the local time zone. It must contain at least two fields, giving the name of the local standard time zone and its offset from GMT in minutes. Offsets are positive for time zones west of Greenwich and negative for time zones east of Greenwich.

If a third field appears in **TIMEZONE**, it gives the name of the local daylight saving time zone. The absence of this field indicates that no daylight saving time correction should be made. If **TIMEZONE** contains no additional fields, the changes between standard time and daylight saving time occur at the times currently legislated in the United States: 2AM standard time on the last Sunday in April and 2AM daylight saving time on the last Sunday in October.

If a fourth and fifth field are present in **TIMEZONE**, they specify the dates on which daylight saving time begins and ends. Each consists of three numbers separated by periods. The third number specifies a month of the year, numbering January as 1. The second number specifies a day of the week, numbering Sunday as 1. The first number specifies which occurence of the weekday in the month marks the change, counting positive occurences from the beginning of the month and negative occurences from the the end of the month.

If a sixth and seventh field are present in **TIMEZONE**, they specify the hour of the day at which daylight saving time begins and ends and the number of minutes of adjustment.

For example, possible **TIMEZONE** settings for Central Standard Time are:

```
TIMEZONE=CST:360
TIMEZONE=CST:360:CDT
TIMEZONE=CST:360:CDT:-1.1.4:-1.1.10
TIMEZONE=CST:360:CDT:-1.1.4:-1.1.10:2:60
```

The first setting provides conversions to standard time only, a convention used by many farmers. The last three settings provide conversions to daylight time and specify the default conversion rules in increasing detail.

FILES
<time.h>
<types.h> for definition of **time_t**

SEE ALSO
ftime
COHERENT Command Manual: **date**

NOTES
The return values of most of these routines are pointers to statically allocated data areas and are overwritten by successive calls.

libc Library

NAME
isalnum, isalpha, isascii, iscntrl, isdigit, islower, isprint, ispunct, isspace, isupper, tolower, toupper – character type checks and conversions

USAGE
#include <ctype.h>

isalnum(*c*)

. . .

DESCRIPTION
The **ctype** type checking macros return a logical value indicating whether the argument is of a particular character type. The macro **isascii** determines whether its integer argument falls within the range of ASCII codes. The other macros assume that **isascii(*c*)** is true or that *c* is **EOF**.

isalnum(*c*)	*c* is alphanumeric (0-9, A-Z, or a-z)
isalpha(*c*)	*c* is a letter (A-Z or a-z)
isascii(*c*)	*c* is an ASCII character (0 <= *c* < 0200)
iscntrl(*c*)	*c* is a control character or delete
isdigit(*c*)	*c* is a digit (0-9)
islower(*c*)	*c* is a lower-case letter (a-z)
isprint(*c*)	*c* is a printable character (neither delete nor control)
ispunct(*c*)	*c* is a punctuation character (neither alphanumeric nor control)
isspace(*c*)	*c* is a space, tab, newline, carriage return, or formfeed
isupper(*c*)	*c* is an upper-case letter (A-Z)

The **ctype** conversion macros perform case conversion. Each assumes that **isalpha(*c*)** is true.

tolower(*c*)	returns *c* converted to a lower-case letter
toupper(*c*)	returns *c* converted to an upper-case letter

FILES
<ctype.h>

SEE ALSO
ascii

File Format

NAME
dir.h – directory format

USAGE
#include <dir.h>

DESCRIPTION
A COHERENT directory is exactly like an ordinary file, except that a user process may write on it only through system calls such as **creat**, **link**, **mknod**, or **unlink**. The system distinguishes directories from other types of files by the mode word **S_IFDIR** in the i-node (see **stat**).

Every directory is an array of entries of the following structure, as defined in the **dir.h** header file:

```
#define DIRSIZ  14

struct  direct  {
        ino_t   d_ino;                  /* i-number */
        char    d_name[DIRSIZ];         /* name */
};
```

Any entry in which **d_ino** has value 0 is unused.

The **mkdir** command creates a directory, with the convention that its first two entries are '**.**' and '**..**'. The name '**.**' is self-referential—a link to the directory itself. The name '**..**' is a link to the parent directory. Since the root directory has no parent, its '**..**' is a link to itself.

The **d_ino** entry of the directory structure is stored in the file system in canonical form, as described in **canon.h**.

FILES
<dir.h>

SEE ALSO
canon.h, stat
COHERENT Command Manual: **mkdir**

System Call

NAME
dup, dup2 – duplicate a file descriptor

USAGE
dup(*fd***)**
int *fd***;**

dup2(*fd***,** *newfd***)**
int *fd***,** *newfd***;**

DESCRIPTION
dup creates a duplicate copy of an existing file descriptor *fd* and returns the duplicate descriptor. The returned value is the smallest file descriptor not already in use by the calling process. The new file descriptor is just a reference to the old one, so seek position and other attributes remain common between them.

dup2 allows the requesting process to specify a new file descriptor *newfd*, rather than having the system pick one. If *newfd* is already open, the system closes it before assigning it to the new file. **dup2** returns the duplicate descriptor.

dup2 is implemented by turning on the 0100 bit of the *fd* argument to the **dup** system call.

SEE ALSO
creat, fopen, fork, open, pipe
COHERENT Command Manual: **sh**

DIAGNOSTICS
Both calls return −1 when an error occurs, such as a bad old file descriptor or no file descriptor available.

libc Library

NAME
ecvt, fcvt, gcvt – convert floating point numbers to strings

USAGE
char *
ecvt(*d, w, dp, signp***)**
double *d***;**
int *w, *dp, *signp***;**

char *
fcvt(*d, w, dp, signp***)**
double *d***;**
int *w, *dp, *signp***;**

char *
gcvt(*d, w, buffer***)**
double *d***;**
int *w***;**
char **buffer***;**

DESCRIPTION
These routines convert floating point numbers to ASCII strings.

ecvt converts *d* into a null-terminated string of decimal digits *w* characters wide, rounding the last digit, and returns a pointer to the result. On return, *dp* points to an integer indicating the location of the decimal point relative to the beginning of the string; to the right if positive, to the left if negative. *signp* points to an integer indicating the sign of *d*; zero if positive, nonzero if negative.

The arguments to **fcvt** have the same meaning, but it converts to FORTRAN F-format.

ecvt and **fcvt** perform conversions into static string buffers which are overwritten by each execution. **gcvt** uses the given *buffer* instead; it should be large enough to hold the result. If possible, **gcvt** mimics **fcvt**; otherwise, it mimics **ecvt. gcvt** returns *buffer.*

SEE ALSO
frexp, printf

libc Library

NAME
edata, end, etext – loader-defined symbols

USAGE
extern int edata;
extern int end;
extern int etext;

DESCRIPTION
The loader **ld** defines the values of **edata, end** and **etext** when it binds a program for execution. **end** is the location after the uninitialized data segment. **edata** is the location after the shared and private data segments. **etext** is the location after the shared and private text (code) segments. The values are just addresses; the locations to which they point contain no known value, and may be illegal memory locations for the program.

The storage allocator **malloc** uses these addresses to determine where the free memory arena should start. The values do not change while the program is running. When a program begins execution, **sbrk** returns the same value as **end.**

SEE ALSO
brk, malloc
COHERENT Command Manual: **ld**

Convention

NAME
environ – process environment

USAGE
extern char **environ;

DESCRIPTION
environ is an array of strings, called the *environment* of a process. By convention, each string has the form

name = value

Normally, each process inherits the environment of its parent process. The shell **sh** and various forms of **exec** can change the environment. The shell adds the name and value of each shell variable marked for **export** to the environment of subsequent commands. The shell adds assignments given on the same line as a command to the environment of the command, without affecting subsequent commands.

SEE ALSO
exec, getenv
COHERENT Command Manual: **sh**

NAME
errno – system call error returns

USAGE
#include <errno.h>
extern int errno;

DESCRIPTION
When an error occurs in a system call, the call normally returns −1 to signify failure. In addition, the call sets the external variable **errno** to a value which identifies the error. Successful calls do not reset **errno** to zero.

The following table gives symbolic error names from the header file **errno.h**, error numbers, and error messages printed by **perror**.

0
The value of **errno** is 0 before any error has occured.

EPERM 1 not the owner or superuser
Someone other than the owner of a file or the superuser attempted to modify it (with **chmod**, for example). This error also occurs when a user other than the superuser requests a privileged system facility, such as setting the time with **stime**.

ENOENT 2 no such file or directory
A referenced file does not exist, or part of the directory structure leading to the file does not exist or is inaccessible.

ESRCH 3 no such process
A process id specified to **kill** or **ptrace** does not exist, perhaps because it has already terminated or has been killed.

EINTR 4 interrupted system call
A signal was caught while a system call was suspended, awaiting the completion of an external event (such as a read from a typewriter device).

EIO 5 I/O error
A physical I/O error occurred on a device driver. This could be a tape error, a CRC error on a disk, or a framing error on a synchronous HDLC link.

System Call

ENXIO 6 no such device or address
A specified minor device is invalid or the unit is powered off. This error might also indicate that a block number given to a minor device is out of range. **suload** returns this error code if the driver was not loaded.

E2BIG 7 argument list too long
The number of bytes of arguments passed in an **exec** is too large.

ENOEXEC 8 exec format error
The file given to **exec** or **load** is not a valid load module (probably because it does not have the magic number at the beginning), even though its mode indicates that it is executable.

EBADF 9 bad file descriptor
A file descriptor passed to a system call is not open or is inappropriate to the call. For example, a file descriptor opened only for reading may not be accessed for writing.

ECHILD 10 no children
A process issued a **wait** call when it had no outstanding children.

EAGAIN 11 no more processes
The system cannot create any more processes, either because it is out of table space or because the invoking process has reached its process quota.

ENOMEM 12 not enough memory
The system cannot accomodate the memory size requested (by **exec** or **brk**, for example).

EACCES 13 permission denied
The user is denied access to a file.

EFAULT 14 bad address
An address in a system call does not lie in the user's address space. Normally, this generates a **SIGSYS** signal, which terminates the process.

ENOTBLK 15 block device required
The **mount** and **umount** calls require block devices as arguments.

System Call

EBUSY 16 mount device busy
The special file passed to **mount** is already mounted, or the file system given to **umount** has open files or active working directories.

EEXIST 17 file exists
An attempt was made to **link** to a file that already exists.

EXDEV 18 cross-device link
A link to a file must be on the same logical device as the file.

ENODEV 19 no such device
An unsuitable I/O call was made to a device. **ioctl** requests to the wrong type of device or attempts to read a line printer are examples.

ENOTDIR 20 not a directory
A component in a pathname exists but is not a directory, or a **chdir** or **chroot** argument is not a directory.

EISDIR 21 is a directory
Directories cannot be opened for writing.

EINVAL 22 invalid argument
An argument to a system call is out of range, e.g. a bad signal number to **kill** or **umount** of a device that is not mounted.

ENFILE 23 file table overflow
A table inside the COHERENT system has run out of space, preventing further **open** calls and related requests.

EMFILE 24 too many open files
A process is limited to 20 open files at any time.

ENOTTY 25 not a tty
An **ioctl** call was made to a file which is not a terminal device.

ETXTBSY 26 text file busy
The text segment of a shared load module is unwritable. Therefore, an attempt to execute it while it is being written or an attempt to open it for writing while it is being executed will fail.

EFBIG 27 file too large
The block mapping algorithm for files fails above 1082201088 bytes.

ENOSPC 28 no space left on device
Indicates an attempt to write on a file when no free blocks remain available on the associated device. This error may also indicate that a device is out of i-nodes, so a file cannot be created.

ESPIPE 29 illegal seek
It is illegal to **lseek** on a pipe.

EROFS 30 read-only file system
Indicates an attempt to write on a file system mounted read-only (e.g. with **creat** or **unlink**).

EMLINK 31 too many links
A new link to a file cannot be created, because the link count would exceed 32767.

EPIPE 32 broken pipe
A write occurred on a pipe for which there are no readers. This condition is accompanied by the signal **SIGPIPE**, so the error will only be seen if the signal is ignored or caught.

EDOM 33 math library domain error
An argument to a mathematical routine falls outside the domain of the function.

ERANGE 34 math library result too large
The result of a mathematical function is too large to be represented.

EKSPACE 35 out of kernel space
No more space is available for tables inside the COHERENT system. Table space is dynamically allocated from a fixed area of memory; it may be possible to increase the size of the area by reconfiguring the system.

ENOLOAD 36 driver not loaded
Not used.

EBADFMT 37 bad exec format
An attempt was made to **exec** a file on the wrong type of processor.

EDATTN 38 device needs attention
The device being referenced needs operator attention. For example, a tape might need a write ring, or a line printer might need paper.

EDBUSY 39 device busy
The indicated device is busy. For **load,** this implies that the given major device number is already in use.

FILES
<errno.h>

SEE ALSO
perror, signal

System Call

NAME
exec – execute a load module

USAGE
execl(*file, arg1, ..., argn,* **NULL)**
char ****file,* ******arg1, ...,* ******argn*;**

execle(*file, arg1, ..., argn,* **NULL,** *env***)**
char ****file,* ******arg1, ...,* ******argn,* ******env*[];**

execlp(*file, arg1, ..., argn,* **NULL)**
char ****file,* ******arg1, ...,* ******argn*;**

execv(*file, argv***)**
char ****file,* ******argv*[];**

execve(*file, argv, env***)**
char ****file,* ******argv*[],** ******env***[];**

execvp(*file, argv***)**
char ****file,* ******argv*[];**

extern char **environ;

DESCRIPTION
The various forms of **exec** allow a process to execute another executable *file* (load module, as described in **l.out.h**). The code, data and stack of *file* replace those of the requesting process. The new stack contains the command arguments and its environment, in the format given below. Execution starts at the entry point of *file*.

During a successful **exec**, the system deactivates profiling and resets any caught signals to **SIG_DFL**.

Every process has a real user id, an effective user id, a real group id, and an effective group id, as described in **getuid**. For most load modules, **exec** does not change any of these. However, if the *file* is marked with the set user id or set group id bit (see **stat**), **exec** sets the effective user id (effective group id) of the process to the user id (group id) of the *file* owner. In effect, this changes the file access privilege level from that of the real id to that of the effective id.

The *file* owner should be careful to limit its abilities, to avoid compromising file security.

exec initializes the new stack of the process to contain a list of strings which are command arguments. **execl, execle** and **execlp** specify arguments individually, as a **NULL**-terminated list of *arg* parameters. **execv, execve** and **execvp** specify arguments as a single **NULL**-terminated array *argv* of parameters.

The **main** routine of a C program is invoked in the following way:

```
main(argc, argv, env)
int argc;
char *argv[], *env[];
```

argc is the number of command arguments passed through **exec**, and *argv* is an array of the actual argument strings. *env* is an array of strings which constitute the process environment. By convention, these strings are of the form *variable=value*, as described in **environ**. Typically, each *variable* is an **export**ed shell variable with the given *value*.

execl and **execv** simply pass the old environment, referenced by the external pointer **environ**. **execle** and **execve** pass a new environment *env* explicitly. **execlp** and **execvp** search for *file* in each of the directories indicated by the shell variable **$PATH**, in the same way that the shell searches for a command. These calls will execute a shell command *file*.

FILES
/bin/sh to execute command files

SEE ALSO
environ, fork, ioctl, signal, stat
COHERENT Command Manual: **sh**

DIAGNOSTICS
None of the routines returns if successful. Each returns −1 for errors, such as *file* nonexistent, not accessible with execute permission, having a bad format, or too large to fit in memory.

System Call, **libc** Library

NAME
exit – terminate a process

USAGE
libc Library:

```
void
exit(status)
int status;
```

System Call:

```
void
_exit(status)
int status;
```

DESCRIPTION
exit is the normal method of terminating the execution of a process. The given *status* information is passed to the parent process. By convention, an exit status of 0 indicates success. If the parent process issued a **wait** call, it is notified of the termination and is passed the least significant 8 bits of *status*. As **exit** never returns, it is always successful.

The routine **exit** from the standard C library **libc** does extra termination cleanup, such as flushing buffered files and closing open files. If this is undesirable, the routine **_exit,** which is simply the system call, may be used instead. The system call exits directly, without performing cleanup.

SEE ALSO
close, wait

NAME
fclose – close stream

USAGE
#include <stdio.h>

fclose(*fp*)
**FILE ****fp*;

DESCRIPTION
fclose closes the stream *fp*. It calls **fflush** on the given *fp,* releases any allocated buffer and calls **close** to complete the closing of the stream.

exit calls **fclose** for each open stream.

FILES
<stdio.h>

SEE ALSO
close, exit, fflush, fopen

DIAGNOSTICS
fclose returns **EOF** on error.

NAME
clearerr, feof, ferror – stream status

USAGE
#include <stdio.h>

clearerr(*fp***)**
FILE ****fp*;**

feof(*fp***)**
FILE ****fp*;**

ferror(*fp***)**
FILE ****fp*;**

DESCRIPTION
Each of these calls is a macro which tests or resets the status of the argument stream *fp*.

ferror returns nonzero if an error has occurred on the argument stream *fp*. For buffered writes, **fflush** should be called before **ferror**, in case an error occurs on the last block written. Any error condition persists until the stream is closed, unless **clearerr** is called to clear it.

feof returns nonzero when an input stream reaches end of file and 0 otherwise. One use is to distinguish a value of −1 returned by **getw** from an **EOF**.

FILES
<stdio.h>

SEE ALSO
errno, getw

STDIO Library

NAME
fflush – flush stream output buffer

USAGE
#include <stdio.h>

fflush(*fp***)**
FILE ****fp*;**

DESCRIPTION
fflush writes out any buffered output data associated with the given stream *fp*. **fclose** calls **fflush**; there is no need for the user program to call it directly under ordinary conditions.

FILES
<stdio.h>

SEE ALSO
fclose, setbuf, write

DIAGNOSTICS
fflush returns **EOF** if the write fails.

STDIO Library

NAME
fileno – get file descriptor

USAGE
#include <stdio.h>

fileno(*fp*)
**FILE ****fp*;

DESCRIPTION
fileno returns the file descriptor associated with the stream *fp*. This file descriptor is the integer returned by the **open** or **creat** call, which some routine such as **fopen** used in creating the stream.

FILES
<stdio.h>

SEE ALSO
fopen, open

NAME
ceil, fabs, floor – ceiling, absolute value, and floor functions

USAGE
#include <math.h>

double
ceil(*z*)
double *z*;

double
fabs(*z*)
double *z*;

double
floor(*z*)
double *z*;

DESCRIPTION
ceil returns the smallest integer greater than or equal to its argument *z*.

fabs implements the absolute value function, returning *z* if *z* is zero or positive and −*z* if *z* is negative.

floor returns the largest integer less than or equal to its argument *z*.

The **math.h** header file contains declarations for each of these routines.

FILES
<math.h>

SEE ALSO
abs

NAME
fdopen, fopen, freopen – open stream for standard I/O

USAGE
#include <stdio.h>

FILE *
fdopen(*fd, type***)**
int *fd*;
**char ****type*;

FILE *
fopen(*name, type***)**
**char ****name, *type*;

FILE *
freopen(*name, type, fp***)**
**char ****name, *type*;
**FILE ****fp*;

DESCRIPTION
fopen allocates and initializes a FILE structure, or *stream*. It calls **open** or **creat** with *name* and returns a pointer to the structure for subsequent use by other STDIO library routines. *type* is a string containing a subset of the characters **rwab** (for **r**ead, **w**rite, **a**ppend, **b**inary) to indicate the desired mode:

r	read; error if file inaccessible
w	write; truncate if found, create if not
a	append; no truncation, create if not found
rw	read and write; error if file inaccessible
wr	write and read; truncate if found, create if not
ar	append and read; no truncation, create if not found

In addition, the *type* strings **r+**, **w+** and **a+** are synonymous with **rw**, **wr** and **ar**, respectively.

Appending the character **b** to the string indicates that the file contains binary data rather than text. This has no effect under COHERENT, but provides compatibility with other operating systems.

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** or **stdout**.

fdopen allocates and returns a file structure for the file descriptor *fd*, as obtained from **open**, **creat**, **dup** or **pipe**. The *type* is ignored.

FILES
<stdio.h>

SEE ALSO
creat, dup, fclose, open, pipe

DIAGNOSTICS
Each function returns **NULL** if it cannot allocate a **FILE** structure, if the *type* is nonsense, or if the **open** or **creat** fails. The current limit is 20 allocated **FILE** structures, including **stdin, stdout** and **stderr**.

System Call

NAME
fork – create a new process

USAGE
fork()

DESCRIPTION
In the COHERENT system, many processes may be active simultaneously. **fork** creates a new process, which is just a duplicate of the requesting process. In practice the new process often issues an **exec** call to invoke a new program.

The process issuing a **fork** call is called the *parent* process, and the new process is called the *child* process. **fork** returns the process id of the newly-created child to the parent process, and returns 0 to the child process. The parent may issue a **wait** to suspend execution until the child terminates.

Parts of the environment of a process exactly duplicated by a **fork** call are: open files and their seek positions; current working and root directories; the file creation mask; the values of all signals; the alarm clock setting; and 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 copying the code segment is avoided. It is write protected, and therefore may be shared.

SEE ALSO
alarm, exec, exit, umask, wait
COHERENT Command Manual: **sh**

DIAGNOSTICS
fork returns −1 on failure, which usually involves insufficient system resources. On successful calls, **fork** returns 0 to the child and the process id of the child to the parent.

STDIO Library

NAME
fread – read from stream

USAGE
#include <stdio.h>

fread(*buffer, size, n, fp***)**
char ****buffer*;**
unsigned *size, n***;**
FILE ****fp*;**

DESCRIPTION
fread reads *n* items of *size* bytes each from the stream *fp* into the memory location *buffer*. **fread** returns the actual number of items read.

FILES
<stdio.h>

SEE ALSO
getc, gets, getw, read, scanf

DIAGNOSTICS
fread returns 0 on end of file or error.

libc Library

NAME

frexp, ldexp, modf – separate mantissa and fraction

USAGE

double
frexp(*real, ep***)**
double *real*;
int **ep*;

double
ldexp(*m, e***)**
double *m*;
int *e*;

double
modf(*real, ip***)**
double *real,* **ip*;

DESCRIPTION

These routines break floating point **(double)** numbers into mantissa and exponent parts.

frexp returns the mantissa *m* of its *real* argument, such that 0 <= *m* < 1, and stores the binary exponent *e* in location *ep*. These numbers satisfy the equation *real* = *m* * 2^*e*.

Conversely, **ldexp** combines the given mantissa *m* with the binary exponent *e* to return a floating point value *real*, which also satisfies the above equation.

modf is the floating point modulus function. It returns the fractional part of its *real* argument, which is a value *f* in the range 0 <= *f* < 1. It also stores the integral part in the **double** location referenced by *ip*. These numbers satisfy the equation *real* = *f* + **ip*.

SEE ALSO

atof, ecvt

STDIO Library

NAME
fseek, ftell, rewind – seek on stream

USAGE
#include <stdio.h>

fseek(*fp, where, how***)**
FILE **fp*;
long *where*;
int *how*;

long
ftell(*fp***)**
FILE **fp*;

rewind(*fp***)**
FILE **fp*;

DESCRIPTION
fseek changes the location where the next read or write operation will occur on stream *fp*. It is the STDIO analogue of **lseek**. **fseek** handles any effects of the seek on the internal buffering strategies of the system.

The *where* and *how* arguments specify the desired seek position. *where* indicates the new seek position in the file, measured from the start of the file if *how* is 0, from the current seek position if *how* is 1, or from the end of the file if *how* is 2.

Sparse files may be created by seeking beyond the current size of the file and writing. Any resultant holes occupy no disk space.

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

ftell returns the current position of the seek pointer. Like **fseek,** it takes into account any buffering associated with the stream *fp*. The return value of **ftell** may be used directly as the input parameter *where* to **fseek**.

FILES
<stdio.h>

SEE ALSO
lseek

DIAGNOSTICS
For any error, such as seeking on a pipe, **fseek** returns -1; otherwise it returns 0.

System Call

NAME
ftime, time – get the current time

USAGE
#include <sys/timeb.h>

ftime(*tbp*)
struct timeb **tbp*;

time_t
time(*tp*)
time_t **tp*;

DESCRIPTION
ftime fills the structure pointed to by argument *tbp* with the internal COHERENT representation of the current time. The structure **timeb** is defined in the **sys/timeb.h** header file, as follows:

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

The member **time** is the number of seconds since midnight GMT of January 1, 1970. The member **millitm** is a count of milliseconds. The members **timezone** and **dstflag** are obsolete.

time is a simpler version. If its pointer argument *tp* is **NULL,** it returns the value of the **time** member of **timeb,** which gives the current time. If *tp* is not **NULL, time** also places the time in the **time_t** variable to which *tp* points.

FILES
<sys/timeb.h>

SEE ALSO
ctime, stime
COHERENT Command Manual: **date**

NOTES
Earlier releases of COHERENT used **timeb** members **timezone** and **dstflag** for time zone and daylight savings time information. **ctime** describes how this information is handled now.

NAME
fwrite – write to stream

USAGE
#include <stdio.h>

fwrite(*buffer, size, n, fp***)**
char ****buffer*;**
unsigned *size, n***;**
FILE ****fp*;**

DESCRIPTION
fwrite writes *n* items of *size* bytes each from *buffer* to stream *fp*. It returns the number of items written.

FILES
<stdio.h>

SEE ALSO
printf, putc, puts, putw, write

DIAGNOSTICS
fwrite returns the number of items actually written; if an error occurs, the return value will not be the same as *n*.

STDIO Library

NAME
fgetc, getc, getchar – read character from stream

USAGE
#include <stdio.h>

int
fgetc(*fp***)**
**FILE ****fp*;

int
getc(*fp***)**
**FILE ****fp*;

int
getchar()

DESCRIPTION
getc is a macro which reads a character from the stream *fp*. **getchar** is a macro which expands to **getc(stdin)**, so it reads a character from the standard input. **fgetc** is a function with body **getc**, for the truly parsimonious.

FILES
<stdio.h>

SEE ALSO
fread, gets, getw, ungetc

DIAGNOSTICS
fgetc, **getc** and **getchar** each return **EOF** at end of file or on read error.

NOTES
Since **getc** is a macro, arguments with side effects will probably not work as expected.

libc Library

NAME
getenv – get environmental variable

USAGE
char *
getenv(*variable*)
**char ****variable*;

DESCRIPTION
The shell **sh** and other programs may set or read certain variables in the process *environment*. This provides a method for conveying user-specific information to commands. The conventional variables stored in the environment are listed in **environ**.

The environment consists of an array of strings, each having the form *variable=value*. When called with the string *variable*, **getenv** returns the string *value*. When *variable* is not found, it returns **NULL**.

SEE ALSO
environ, exec
COHERENT Command Manual: **sh**

libc Library

NAME

endgrent, getgrent, getgrgid, getgrnam, setgrent – get group file information

USAGE

#include <grp.h>

endgrent()

struct group *
getgrent();

struct group *
getgrgid(*gid***)**
int *gid***;**

struct group *
getgrnam(*gname***)**
char ****gname*;**

setgrent()

DESCRIPTION

These routines search the file **/etc/group**, which contains information about the name and members of valid user groups. The returned structure **group**, which is defined in the **grp.h** header file, is as follows:

```
struct  group {
        char    *gr_name;          /* Group name */
        char    *gr_passwd;        /* Group password */
        int     gr_gid;            /* Numeric group id */
        char    **gr_mem;          /* Group members */
};
```

For detailed descriptions of the above fields, consult **group**.

Each **getgrent** call returns the next entry from **/etc/group**. The **getgrgid** call attempts to find the first entry with a numerical group id of *gid*. Similarly, **getgrnam** looks for a group with name *gname*.

setgrent rewinds the group file, which allows restarting the search at the beginning for repeated group searches. A call to **endgrent** closes the group file.

The routines in **getpwent** provide similar information for the password file **/etc/passwd**.

FILES
/etc/group
<grp.h>

SEE ALSO
getlogin, getpwent, group

DIAGNOSTICS
The routines return **NULL** for any error or at end of file.

NOTES
All structures and information returned are in static areas internal to these routines. Therefore, information from a previous call is overwritten by each subsequent call.

libc Library

NAME
getlogin – get login name

USAGE
char *
getlogin()

DESCRIPTION
The name corresponding to the current user id is not always the same as the name under which a user logged into the COHERENT system. For example, the user may have issued a **su** command, or there may be several login names associated with a user id. **getlogin** returns the login name found in the file **/etc/utmp**.

In cases where **getlogin** fails to produce a result, **getpwuid** (described in **getpwent**) is normally used to determine the user name for a process.

FILES
/etc/utmp login names

SEE ALSO
getpwent, getuid, ttyname, utmp.h
COHERENT Command Manual: **su, who**

DIAGNOSTICS
getlogin returns **NULL** if the login name cannot be determined.

NOTES
getlogin stores the returned name in a static area which is destroyed by subsequent calls.

libc Library

NAME
getpass – get password with prompting

USAGE
char *
getpass(*prompt***)**
char ****prompt*;**

DESCRIPTION
getpass first prints the *prompt*. Then it disables echoing of input characters on the terminal device (either the file **/dev/tty** or the standard input), reads a password from it, and restores echoing on the terminal. It returns the given password.

FILES
/dev/tty

SEE ALSO
crypt
COHERENT Command Manual: **login, passwd, su**

NOTES
The password is stored in a static location which is overwritten by successive calls.

System Call

NAME
getpid – get process id

USAGE
getpid( )

DESCRIPTION
Every process has a unique number, called its *process id*. **fork** returns the process id of a created child process to the parent process.

getpid returns the process id of the requesting process. Typically a process uses **getpid** to pass its process id to another process which wants to send it a signal, or to generate a unique temporary file name.

SEE ALSO
fork, kill, mktemp

libc Library

NAME
getpw – search password file

USAGE
getpw(*uid, line***)**
short *uid***;**
char ****line*;**

DESCRIPTION
getpw searches the password file **/etc/passwd** for the first entry with numerical user id *uid*. If found, *line* receives the corresponding line from the password file.

FILES
/etc/passwd

SEE ALSO
getpwent, getuid, passwd

DIAGNOSTICS
getpw returns a nonzero value on error.

NAME
endpwent, getpwent, getpwnam, getpwuid, setpwent – get password file information

USAGE
#include <pwd.h>

endpwent()

struct passwd *
getpwent()

struct passwd *
getpwnam(*uname***)**
char ****uname*;**

struct passwd *
getpwuid(*uid***)**
int *uid***;**

setpwent()

DESCRIPTION
These routines search the file **/etc/passwd**, which contains information about every user of the system. The returned structure **passwd**, which is defined in the **pwd.h** header file, is as follows:

```
struct  passwd {
        char    *pw_name;          /* login user name */
        char    *pw_passwd;        /* login password */
        int     pw_uid;            /* login user id */
        int     pw_gid;            /* login group id */
        int     pw_quota;          /* file quota (unused) */
        char    *pw_comment;       /* comments (unused) */
        char    *pw_gecos;         /* (unused) */
        char    *pw_dir;           /* working directory */
        char    *pw_shell;         /* initial program */
};
```

For detailed descriptions of the above fields, consult **passwd**.

libc Library

Each **getpwent** call returns the next entry from **/etc/passwd**. The **getpwuid** call attempts to find the first entry with a numerical user id of *uid*. Similarly, **getpwnam** looks for an entry with user name *uname*.

setpwent rewinds the password file, which allows restarting the search at the beginning for repeated searches. A call to **endpwent** closes the password file.

The routines in **getgrent** provide similar information for the group file **/etc/group**.

FILES
/etc/passwd
<pwd.h>

SEE ALSO
getgrent, getlogin, passwd

DIAGNOSTICS
The routines return **NULL** for any error or on end of file.

NOTES
All structures and information returned are in static areas internal to these routines. Therefore, information from a previous call is overwritten by each subsequent call.

STDIO Library

NAME
fgets, gets – read line from stream

USAGE
#include <stdio.h>

char *
fgets(*s, n, fp***)**
char ****s*;**
int *n***;**
FILE ****fp*;**

char *
gets(*s***)**
char ****s*;**

DESCRIPTION
gets reads characters from the standard input into string *s* up to the next newline or end of file. It discards the newline (if any) and appends a trailing null character. **gets** returns the argument *s*.

fgets reads characters from the stream *fp* into string *s* until $n-1$ characters have been read or up to the next newline or end of file. **fgets** retains the newline (if any) and appends a trailing null character. **fgets** returns the argument *s*.

FILES
<stdio.h>

SEE ALSO
getc, puts

DIAGNOSTICS
Both functions return **NULL** if end of file occurs before any characters are read or if an error occurs.

NOTES
gets deletes newlines, in the name of backward compatibility.

NAME
getty – terminal initialization

USAGE
/etc/getty *type*

DESCRIPTION
The initialization process **init** invokes **getty** for each terminal indicated in the file **/etc/ttys**. **getty** tries to read a user name from the terminal which is the standard input, adapting its mode settings accordingly. Then **getty** invokes **login** with the name read. This process may set delays, upper to lower case mapping, speed, and whether the terminal normally uses carriage return or linefeed to terminate input.

If the terminal baud rate is wrong, the login message printed by **getty** will appear garbled. If the specified *type* indicates variable speeds, as described below, hitting BREAK will try the next speed.

init passes the second character in a line of the **/etc/ttys** file as the *type* argument to **getty**. *type* conveys information about the terminal port. An upper-case letter in the range **A** to **S** specifies a hard-wired baud rate, as indicated in **<sgtty.h>**. Other characters specify a range of speeds suitable to a dial-in modem. The following characters are recognized:

0	Cycles through speeds 300, 1200, 150, and 110 baud, in that order; a good default for dial-in ports.
–	Teletype model 33, fixed at 110 baud.
1	Teletype model 37, fixed at 150 baud.
2	9600 baud with delays (e.g. Tektronix 4104).
3	Cycles between 1200 and 300 baud; used with 212 modems.
4	DECwriter (LA36) with delays.
5	Like **3**, but starts at 300 baud.
A	50 baud.
B	75 baud.
C	110 baud.
D	134 baud.
E	150 baud.
F	200 baud.
G	300 baud.

H	600 baud.
I	1200 baud.
J	1800 baud.
K	2000 baud.
L	2400 baud.
M	3600 baud.
N	4800 baud.
O	7200 baud.
P	9600 baud.
Q	19200 baud.
R	EXTA
S	EXTB

FILES
/etc/ttys
<sgtty.h>

SEE ALSO
init, ioctl, ttys
COHERENT Command Manual: **login, stty**

System Call

NAME
getegid, geteuid, getgid, getuid – get user and group id

USAGE
getegid()

geteuid()

getgid()

getuid()

DESCRIPTION
Every process has two different versions of its *user id*, called the *real* user id and the *effective* user id. The user ids determine eligibility to access files or employ system privileges. Normally these two ids are identical. However, for a *set user id* load module (see **exec**), the real user id is that of the user, while the effective user id is that of the load module owner. This distinction allows system programs to use files which are protected from the user who invokes the program.

getuid returns the real user id, while **geteuid** returns the effective user id. **setuid** sets the effective user id to the real user id.

getgid and **getegid** are analogous calls for the *group id*, returning the real and effective group ids, respectively.

SEE ALSO
access, exec, setuid
COHERENT Command Manual: **login**

STDIO Library

NAME
fgetw, getw – read integer from stream

USAGE
#include <stdio.h>

fgetw(*fp*)
FILE **fp*;

getw(*fp*)
FILE **fp*;

DESCRIPTION
The macro **getw** reads a word (an **int**) from the stream *fp*. **fgetw** has the same effect, but is a function rather than a macro.

FILES
<stdio.h>

SEE ALSO
ferror, fread, getc, gets, ungetc

DIAGNOSTICS
Both routines return the value **EOF** on errors. A call to **feof** or **ferror** may be necessary to distinguish this value from a valid data item.

NOTES
Because **getw** is a macro, arguments with side effects will probably not work as expected.

libc Library

NAME
getwd – get current working directory name

USAGE
char *
getwd()

DESCRIPTION
The *current working directory* is the directory from which file name searches commence when a pathname does not begin with '/'. **getwd** returns the name of the current working directory. It is useful for processes which need to generate full pathnames for files, such as spoolers and daemons.

If the invoker does not have permission to search all levels of directory hierarchy above the current directory, **getwd** will not be able to obtain the directory name.

SEE ALSO
chdir
COHERENT Command Manual: **pwd**

DIAGNOSTICS
getwd returns **NULL** if the current directory cannot be found.

NOTES
The return value points at a static area which is limited in size to 400 characters. **getwd** will fail if the current directory name is longer.

There is some chance that the working directory will not be restored to its initial value if **getwd** fails.

File Format

NAME
group – group file format

DESCRIPTION
The group file **/etc/group** describes user groups for file access purposes. The file contains the information to map any ASCII group name to the corresponding numerical group id, and vice versa. It also contains the ASCII names of the members of each group. The **newgrp** command uses this information.

Each group entry consists of a single line. Each line consists of four colon-separated ASCII fields:

group_name:*password*:*group_number*:*member*[,*member*]...

Passwords are encrypted with **crypt**, so the group file is generally readable.

FILES
/etc/group

SEE ALSO
crypt, getgrent, passwd
COHERENT Command Manual: **chgrp, newgrp, passwd**

NOTES
At present the group password field cannot be set directly (no command similar to **passwd** exists for groups). One alternative is to set the password in the **/etc/passwd** file for a user with the **passwd** command, and then transcribe the password into the group file manually.

libm Library

NAME
cabs, hypot – complex absolute value function

USAGE
#include <math.h>

double
cabs(*z***)**
struct { double *r, i*; **}** *z*;

double
hypot(*r, i***)**
double *r, i*;

DESCRIPTION
cabs computes the absolute value (or modulus) of its complex argument *z*. The absolute value of a complex number is the length of the hypotenuse of a right triangle with sides given by the real part *r* and the imaginary part *i*. The result is the square root of the sum of the squares of the parts.

hypot computes the same value, but with *r* and *i* passed as separate parameters.

FILES
<math.h>

SEE ALSO
abs, floor, log

DIAGNOSTICS
The functions return a very large number and set **errno** to **ERANGE** when the correct result would overflow.

Maintenance

NAME
init, rc – system initialization

USAGE
/etc/init

/etc/rc

DESCRIPTION
The COHERENT boot procedure executes **init** as process 1 to perform initialization. **init** opens the console terminal **/dev/console** and invokes a shell **sh** on it with **HOME** set to **/etc**. The shell executes **/etc/profile** and **/etc/.profile** if present. The system then runs in single user mode and accepts commands from the console.

When the console terminates the shell, normally by typing **<ctrl-D>**, **init** brings up the system in multiuser mode. It executes the shell command file **/etc/rc**, which performs standard bookkeeping and maintenance chores. Typically it mounts standard file systems, removes temporary files, and invokes **cron** and **update**. If desired, it may load device drivers, enable swapping with **swap**, and enable process accounting with **accton**.

Next **init** opens terminals as specified in the file **/etc/ttys**. It invokes **getty** to read a user name and perform a **login** on each terminal.

When a user shell terminates, **init** updates the system accounting information in **/etc/utmp** and **/usr/adm/wtmp**. Then it reopens the appropriate terminal and invokes **getty**, as above.

init rescans the **/etc/ttys** file for terminal changes if it receives the **SIGQUIT** signal. The command "**kill –3 1**" sends **SIGQUIT** to the **init** process. **init** then invokes **getty** as necessary.

init returns the system to single user mode if it receives the **SIGHUP** signal. The command "**kill –1 1**" sends **SIGHUP** to the **init** process.

FILES

/dev/console	console terminal
/dev/tty_??_	terminal devices
/etc/rc	initialization command file
/etc/ttys	active terminals

/etc/utmp	logged in users
/usr/adm/wtmp	login accounting data

SEE ALSO
getty, ttys
COHERENT Command Manual: **kill, login, sh**

NAME
ioctl, gtty, stty – device-dependent control

USAGE
#include <sgtty.h>

ioctl(*fd, command, info***)**
int *fd, command***;**
char **info***;**

gtty(*fd, sgp***)**
int *fd***;**
struct sgttyb **sgp***;**

stty(*fd, sgp***)**
int *fd***;**
struct sgttyb **sgp***;**

DESCRIPTION
ioctl provides direct interaction with a device driver. Possible uses include setting or retrieving parameters for devices (line printers, communications lines, terminals) and non-standard spacing operations for tape drives.

ioctl acts upon a block special file or a character special file associated with an already open file descriptor *fd*. The *command* argument gives the specific request. A system header file defines symbolic command parameters for each device type. For example, **sgtty.h** defines commands for terminals and **mtioctl.h** defines commands for magnetic tape drives. Using the symbolic command definitions from the header files promotes device independence within each device type. Section **Device Drivers** at the beginning of this manual lists other sections which give details about specific devices.

The *info* argument passes a buffer of information (defined by structures in the appropriate header files) to the driver. For any *command* not needing additional information, this argument should be **NULL**.

stty and **gtty** are shorthand notations for **ioctl** calls with a *command* argument of **TIOCSETP** and **TIOCGETP**, respectively. These routines set and get attributes of a terminal.

FILES
<sgtty.h>
<mtioctl.h>

SEE ALSO
exec, open, read, write
COHERENT Command Manual: **stty**

DIAGNOSTICS
ioctl returns −1 on errors, such as a bad file descriptor. Since the call is device dependent, almost any other error could be returned.

NOTES
The type of the *info* argument to **ioctl** is declared as **char *** mainly for portability reasons. In most cases, the actual argument type will be something like **struct sgttyb ***, depending on the particular device and command. The actual argument should be cast to type **char *** to ensure cross-machine portability.

libm Library

NAME
j0, j1, jn – Bessel functions of first kind

USAGE
#include <math.h>

double
j0(*z*)
double *z*;

double
j1(*z*)
double *z*;

double
jn(*n, z*)
int *n*;
double *z*;

DESCRIPTION
j0, j1 and **jn** take an argument *z* and compute the Bessel function of the first kind for order 0, order 1, and arbitrary order, respectively. For **jn**, the argument *n* is the integral order of the function.

FILES
<math.h>

NOTES
Bessel functions of the second kind **y0, y1** and **yn** are not yet implemented. Hankel functions **h0, h1** and **hn** (Bessel functions of the third kind) might also be useful but are not implemented.

System Call

NAME
kill – send a signal to a process

USAGE
#include <signal.h>

kill(*pid, signal***)**
int *pid, signal*;

DESCRIPTION
kill provides a method of asynchronously signalling a process with one of a pre-defined set of signals, as described in **signal**. **kill** sends the given *signal* to the process with the specified *pid*.

There are two special cases for specifying which processes will receive the signal. If *pid* is 0, **kill** signals all other processes in the same process group (see **tty**) as the invoker. Usually this group is all processes affiliated with a particular terminal. If *pid* is −1, **kill** signals all other processes except process 1 (the initialization process). Use of this case is restricted to the superuser.

The caller may send any signal to processes which have the same effective user id. The caller may send **SIGHUP**, **SIGINT**, **SIGQUIT** or **SIGTERM** to processes which have the same real user id. A process running as the superuser may send any signal to any process. A process sending a signal to itself (suicide) is allowed.

The signal has been sent when the call returns. However, it may be some time before the recipient sees it, owing to scheduling delays.

FILES
<signal.h>

SEE ALSO
ptrace, signal, wait
COHERENT Command Manual: **kill**

DIAGNOSTICS
kill is successful and returns 0 if the conditions above have been met: whether the recipient ignores the signal is irrelevant.

NAME
l.out.h – object file format

USAGE
#include <l.out.h>

DESCRIPTION
This section describes the format for the output of compilers, assemblers, and the loader. Assembler output is an unbound object; it must be bound with any required libraries (leaving no unresolved symbols) to produce an executable object file, or *load module*. An **exec** call can then execute the load module directly.

The load module begins with a header, which gives global information and size information about each segment. Segments of the size indicated follow the header in a fixed order. The **l.out.h** header file defines the header structure for the Z8000 and M68000:

```
struct  ldheader {
        short   l_magic;
        short   l_flag;
        short   l_machine;
        short   l_tbase;
        size_t  l_ssize[NLSEG];
        long    l_entry;
};
```

for the 8086 and 8088 processors and PDP-11:

```
struct  ldheader {
        int     l_magic;
        int     l_flag;
        int     l_machine;
        vaddr_t l_entry;
        size_t  l_ssize[NLSEG];
};
```

l_magic is the magic number which identifies a load module; it always contains **L_MAGIC**. **l_flag** contains flags indicating the type of the object. **l_machine** is the processor identifier, as defined in the **mtype.h** header file. **l_tbase** is the start of the text segment. **l_entry** contains the machine address where execution of the module commences. **l_ssize** gives the size of each segment.

FILES

l.out	default load module name
<l.out.h>	
<mtype.h>	machine identifiers

SEE ALSO
core, exec, mtype
COHERENT Command Manual: **as, cc, ld, nm**

NOTES
In the early releases of COHERENT the header structure was defined only as it shown above for 8086; it was changed to handle 32-bit addresses. In the future, the header structure defined above for Z8000 and M68000 machines will be implemented on 8086 and 8088 systems as well.

NAME
l3tol, ltol3 – convert long integer to/from file system block number

USAGE
l3tol(*lp, l3p, n***)**
long **lp***;**
char **l3p***;**
unsigned *n***;**

ltol3(*l3p, lp, n***)**
char **l3p***;**
long **lp***;**
unsigned *n***;**

DESCRIPTION
To conserve space inside i-nodes in COHERENT file systems, the system stores block addresses in three bytes. Programs which reference or maintain file systems use these routines to convert between the three byte representation and **long** numbers.

l3tol converts *n* 3-byte block addresses at location *l3p* to an array of **long** integers at location *lp*. **ltol3** converts *n* **long** integers at *lp* to the more compact form at *l3p*.

SEE ALSO
canon.h

System Call

NAME

link – create a link

USAGE

link(*old, new***)**
char ****old*, ******new*;**

DESCRIPTION

A *link* to a file is another name for the file. All attributes of the file appear identical among all links.

link creates a link called *new* to an existing file named *old*.

For administrative reasons, it is an error for users other than the superuser to create a link to a directory. Such links can make the file system no longer tree structured unless carefully controlled, posing problems for commands such as **find**.

SEE ALSO

unlink
COHERENT Command Manual: **find, ln**

DIAGNOSTICS

link returns 0 when successful. It returns –1 on errors, such as: *old* does not exist, *new* already exists, attempt to link across file systems, no permission to create *new* in the target directory.

NOTES

Because each mounted file system is a completely separate and self-contained entity, links between different mounted file systems fail.

System Call

NAME
lock – prevent process from swapping

USAGE
lock(*flag***)**
int *flag***;**

DESCRIPTION
Normally a process may be swapped in and out of memory by the COHERENT system. However, locking a process in memory is sometimes required to guarantee real-time response.

With a nonzero *flag*, **lock** prevents the calling process from swapping (unless swapping is required to increase its memory size). With a zero *flag*, **lock** unlocks the process.

This call is restricted to the superuser. Processes doing raw I/O are automatically locked into memory for the duration of the I/O operation.

SEE ALSO
swap
COHERENT Command Manual: **ps**

DIAGNOSTICS
lock returns 0 if it performs the indicated action and −1 otherwise. An error occurs if the caller is not the superuser.

NOTES
The existence of several locked processes may cause memory fragmentation. Therefore, all locked processes should be created just after the system is booted, if possible. In general, **lock** should be avoided if any alternative exists.

libm Library

NAME
exp, log, log10, pow, sqrt – logarithmic and exponential functicns

USAGE
#include <math.h>

double
exp(*z***)**
double *z***;**

double
log(*z***)**
double *z***;**

double
log10(*z***)**
double *z***;**

double
pow(*z***,** *x***)**
double *z***,** *x***;**

double
sqrt(*z***)**
double *z***;**

DESCRIPTION
exp returns the exponential of *z*, or e^*z*.

log returns the natural (base e or Napierian) logarithm of *z*. **log10** returns the common (base 10) logarithm of *z*.

pow returns *z* raised to the power *x*, or *z*^*x*.

sqrt returns the square root of *z*.

FILES
<math.h>

SEE ALSO
hypot, sin, sinh

DIAGNOSTICS

exp and **pow** indicate overflow by an **errno** of **ERANGE** and a huge returned value. A domain error in **log** (*z* less than or equal to 0), in **pow** (*x* negative and not an integer, or both *z* and *x* 0), or in **sqrt** (*z* negative) sets **errno** to **EDOM** and returns 0.

NAME
lpd – line printer spooler daemon

USAGE
/usr/lib/lpd

DESCRIPTION
lpd is a daemon program that runs in the background and prints listings queued by the **lpr** command. It is run automatically by **lpr**. If there is no printing to do, or if another daemon is already running (indicated by the **dpid** file), **lpd** exits immediately. Otherwise, it searches the spool directory for control files of listings to print. These control files contain the names of files to print, the user name, banners, and files to be removed upon completion.

lpd does not print listings in any particular order. There is no prioritization of printing, neither by size nor by requester.

The **lpskip** command terminates or restarts the current line printer listing.

FILES

/dev/lp	printer
/usr/spool/lpd	spool directory
/usr/spool/lpd/cf*	control files
/usr/spool/lpd/df*	data files
/usr/spool/lpd/dpid	lock and process id

SEE ALSO
init
COHERENT Command Manual: **lpr, lpskip**

System Call

NAME
lseek – set read/write position

USAGE
long
lseek(*fd, where, how***)**
int *fd, how***;**
long *where***;**

DESCRIPTION
lseek changes the location where the next read or write operation will occur on the file identified by file descriptor *fd*. A read or write will happen at the current seek position, and will advance the seek position by the number of bytes successfully transferred.

The *where* and *how* arguments specify the desired seek position. *where* indicates the new seek position in the file, measured from the beginning of the file if *how* is 0, from the current seek position if *how* is 1, or from the end of the file if *how* is 2. A successful **lseek** call returns the new seek position.

Sparse files may be created by seeking beyond the current size of the file and writing. Any resultant holes occupy no disk space.

SEE ALSO
fseek, open, read, write

DIAGNOSTICS
lseek returns (long) – 1 on error, such as seeking on a pipe or seeking to a negative position.

NOTES
lseek is permitted on character special files, but drivers do not generally implement it. As a result, seeking a terminal will not generate an error but will have no discernible effect.

libc Library

NAME
calloc, free, malloc, realloc – memory allocator

USAGE
char *
calloc(*count, size***)**
unsigned *count, size*;

free(*ptr***)**
**char ****ptr*;

char *
malloc(*size***)**
unsigned *size*;

char *
realloc(*ptr, size***)**
**char ****ptr*;
unsigned *size*;

DESCRIPTION
These routines manage an *arena*, an area of memory divided into used and unused blocks. **malloc** selects an unused block of at least *size* bytes using a circular first fit algorithm, marks as much of it as needed as used, and returns a pointer to it. **malloc** extends the arena when necessary (if possible).

free marks the block indicated by *ptr* as unused and coalesces it with contiguous free blocks. It issues a diagnostic and calls **abort** if *ptr* points to a bad block (not obtained from **malloc**, or overwritten beyond its boundaries).

calloc calls **malloc** to obtain a block large enough to contain *count* items of *size* bytes each. **calloc** initializes the block to zeroes and returns a pointer to it.

realloc returns a block with the new *size* and with the same contents as the old block indicated by *ptr* up to the smaller of the old and new sizes. **realloc** tries to return the same block (truncated or extended); if *size* is smaller than the size of the old block, **realloc** will always return the same *ptr*.

SEE ALSO
abort

DIAGNOSTICS
malloc, calloc and **realloc** all return **NULL** if insufficient memory is available to satisfy the request. **malloc** prints a message and calls **abort** if it discovers that the arena has been corrupted (usually by storing past the bounds of an allocated block).

NOTES
The *ptr* argument to **realloc** must have been obtained from **malloc**. If **realloc** fails and returns **NULL**, the old block will have been freed.

Convention

NAME
man – manual macro package

USAGE
nroff –man *file*...

DESCRIPTION
The **nroff** macro package **man** formats manual pages in the style of the *COHERENT Command Manual* and the *COHERENT System Manual*. It includes the following macros:

.B	Boldface font
.BI	Bold/italic alternating fonts
.BR	Bold/Roman alternating fonts
.CO	COHERENT
.DE	Display end
.DS	Display start
.DT	Default tabs
.HE	Help end
.HP	Hanging paragraph
.HS	Help start
.I	Italic font
.IB	Italic/bold alternating fonts
.IP	Indented paragraph
.IR	Italic/Roman alternating fonts
.LP	Paragraph
.PD	Paragraph distance
.PP	Paragraph
.RB	Roman/bold alternating fonts
.RE	Relative indent end
.RI	Roman/italic alternating fonts
.RS	Relative indent start
.SH	Subheader
.SM	Smaller size
.TH	Header
.TP	Tagged paragraph

FILES

/usr/lib/tmac.an	macro package
/usr/man/*/*	manual files

SEE ALSO
ms
COHERENT Command Manual: **help, man, nroff**
nroff Text Processor Tutorial

Device Driver

NAME
mem – physical memory file

DESCRIPTION
The special file **/dev/mem** allows the physical memory of the host computer to be read and written just like an ordinary file. The location where I/O will occur can be positioned to any valid byte address by the **lseek** call.

Commands may examine or change addresses in physical memory. Addresses to use when changing the system itself normally are obtained from the system load module (**/coherent**) name list, so that they always reflect the currently running version of the system.

FILES
/dev/mem

SEE ALSO
core, lseek
COHERENT Command Manual: **ps**

DIAGNOSTICS
On an error, such as nonexistent memory location, –1 is returned.

System Call

NAME
mknod – create a special file

USAGE
#include <sys/stat.h>

mknod(*file, mode, dev***)**
char ****file*;**
int *mode***;**
dev_t *dev***;**

DESCRIPTION
mknod creates a *file*, which may be an ordinary file, a directory or a special file. The *mode* argument works in the same way as for **creat**, except that all bits of the mode are significant.

The *dev* argument specifies the first block address of the i-node for the new file. When creating a directory, *dev* should be 0. When creating a block or character special file, *dev* should hold the major and minor device numbers. The **makedev(***major, minor***)** macro from the **stat.h** header file is available to construct the *dev* argument.

mknod is restricted to the superuser.

FILES
<sys/stat.h>

SEE ALSO
creat, stat, umask
COHERENT Command Manual: **mkdir, mknod**

DIAGNOSTICS
mknod returns 0 if successful. It returns –1 on errors, such as *file* already existing or the caller not being the superuser.

libc Library

NAME
mktemp – generate temporary file name

USAGE
char *
mktemp(*pattern*)
**char ****pattern*;

DESCRIPTION
mktemp facilitates the generation of a file name which no other process will use, for purposes such as intermediate data files or files in a spool directory. The *pattern* argument should consist of a string with six **X**'s at the end. **mktemp** replaces these **X**'s by the five digit process id of the requesting process and a letter that is changed for each subsequent call. **mktemp** returns *pattern*.

As an example, the call:

```
mktemp("/tmp/sortXXXXXX");
```

might return the name "**/tmp/sort01234a**". It is normal practice to place temporary files in the directory **/tmp**. The start of the file name identifies the originator of the file.

SEE ALSO
getpid

System Call

NAME

mount, umount – mount/unmount file system

USAGE

mount(*filesystem, directory, flag***)**
char ****filesystem,* ******directory*;**
short *flag***;**

umount(*filesystem***)**
char ****filesystem*;**

DESCRIPTION

The existing file system hierarchy may be augmented by grafting another file system onto it with **mount**. It may be pruned with **umount**.

mount grafts the file system living on the device *filesystem* onto the hierarchy at location *directory*. This *directory* loses its identity for the duration of the **mount**, becoming instead the root directory of the newly-mounted file system. *filesystem* must be the pathname of a block special file which holds a COHERENT file system structure.

If the *flag* argument is nonzero, the file system is mounted read-only. No write operations will be allowed. The system will not update information such as access times. This mode is useful for eliminating write operations where too expensive or inappropriate (e.g. magnetic tape), or where special care is needed (e.g. a precious backup).

umount undoes a previous **mount**. The pathname *filesystem* must be a block special file containing an already mounted file system.

FILES

/dev/*

SEE ALSO

init, sload, sync
COHERENT Command Manual: **load, mount, uload, umount**

DIAGNOSTICS

The routines return zero if successful. A return value of −1 indicates an error. **mount** errors can occur if *filesystem* does not exist or is not marked executable, if *directory* does not exist, or if the system has insufficient resources to mount an additional file system. **umount** will fail if *filesystem* is not currently mounted.

libmp Library

NAME
mp – multiple precision arithmetic library

USAGE
#include <mprec.h>

```
void    gcd(a, b, c)
int     ispos(a)
mint *  itom(n)
void    madd(a, b, c)
int     mcmp(a, b)
void    mcopy(a, b)
void    mdiv(a, b, q, r)
void    min(a)
void    minit(a)
void    mintfr(a)
void    mitom(n, a)
void    mneg(a, b)
void    mout(a)
void    msqrt(a, b, r)
void    msub(a, b, c)
int     mtoi(a)
char *  mtos(a)
void    mult(a, b, c)
void    mvfree(a)
void    pow(a, b, m, c)
void    rpow(a, b, c)
void    sdiv(a, n, q, ip)
void    spow(a, n, b)
void    smult(a, n, c)
void    xgcd(a, b, r, s, g)
int     zerop(a)

int     n, *ip;
mint    *a, *b, *c, *g, *m, *q, *r, *s;

extern int ibase, obase;
extern mint *mzero, *mone, *mminint, *mmaxint;
```

libmp Library

DESCRIPTION
The functions in the **libmp** library enable the user to perform multiple precision arithmetic. The data structure they manipulate is called a **mint**, for multiple precision integer, defined in **mprec.h**:

```
typedef struct {
        unsigned len;
        char *val;
} mint;
```

However, users should not depend on the details of this structure, since on some machines a different representation may be more efficient. Using the listed functions is always safe.

In all cases except **xgcd** below, none of the **mint** arguments need be distinct. The description uses a slight notational abuse: frequently a pointer to a **mint**, *a* for example, is used to denote the value of the **mint** pointed to by *a*. The meaning should be clear from the context.

itom creates a new **mint**, initializes it to the signed integer value *n*, and returns a pointer to it. Storage used by a **mint** created with **itom** may be reclaimed using **mintfr**.

A **mint** that already exists may be reinitialized by **mitom**, which sets *a* to the value *n*. If the **mint** was declared as a global or automatic variable, it must be conditioned before first use by **minit**, which prevents garbage values in the **mint** structure from causing chaos. A **mint** conditioned by **minit** has no value; however, it may be used to receive the result of an operation. For **mint**s automatic to a function, **mvfree** should be used before the function is exited to free the storage used by the **val** field of the **mint** structure. Otherwise, this storage will never be reclaimed.

madd, **msub** and **mult** set *c* to the sum, difference or product of *a* and *b*. **mdiv** divides *a* by *b* and places the quotient and remainder in *q* and *r*. *b* must not be zero. The results of the operation are defined by the conditions

1. $a = q * b + r$,
2. the sign of $r =$ the sign of q,
3. the absolute value of $r <$ the absolute value of b.

smult is like **mult**, except the second argument is an integer in the range 0 < = *n* < = 127. **sdiv** is like **mdiv**, except the second argument is an integer in the range 1 < = *n* < = 128, and the remainder argument points to an **int** instead of a **mint**.

pow sets *c* to *a* raised to the *b* power reduced modulo *m*. **rpow** sets *c* to *a* raised to the *b* power. **spow** is like **rpow**, except the exponent is an integer. In no case may the exponent be negative.

mcopy sets *b* equal to *a*. **mneg** sets *b* equal to negative *a*.

msqrt sets *b* to the integral portion of the positive square root of *a*; *r* is set to the remainder. *a* must not be negative. The result of the operation is defined by the condition

$$a = b * b + r.$$

gcd sets *c* to the greatest common divisor of *a* and *b*. **xgcd** is an extended gcd routine that sets *g* to the greatest common divisor of *a* and *b*, and sets *r* and *s* so the relation

$$g = a * r + b * s$$

holds. For **xgcd**, *r*, *s* and *g* must all be distinct.

mints may be compared with **mcmp**, which returns a signed integer less than, equal to, or greater than zero according to whether *a* is less than, equal to, or greater than *b*. **ispos** returns true (nonzero) if *a* is not negative, false (zero) if *a* is negative. **zerop** returns true if *a* is zero, false otherwise.

mtoi returns an integer equal to the value of *a*. *a* should be in the allowable range for a signed integer.

The external integers **ibase** and **obase** govern the I/O and ASCII conversion routines. Allowable bases run from 2 to 16. Permissible digits are 0–9 and A–F (lower case a–f are not allowed). **min** reads a **mint** in base **ibase** from the standard input and sets *a* to that value. Leading blanks and an optional leading minus sign are allowed; the number is terminated by the first non-legal digit. **mout** outputs *a* on the standard output in base **obase**. **mtos** performs the same conversion as **mout**, but the result is placed in a character string instead of being output; a pointer to the string is returned. The string is actually allocated by **malloc**, and may be freed by **free**.

mzero and **mone** point to **mint**s with values 0 and 1. **mminint** and **mmaxint** point to **mint**s containing the minimum and maximum values that will fit in a signed integer. These constants should never be used as the result of an operation.

All the necessary declarations for these constants and for the library functions are contained in the header file **mprec.h**. They need not be repeated.

To link **mp** modules with an executable object, use the argument **−lmp** with the **cc** or **ld** command.

FILES
<mprec.h>
/usr/lib/libmp.a

SEE ALSO
libm Library, malloc
COHERENT Command Manual: **bc, dc**

DIAGNOSTICS
On any error, such as division by zero, running out of space or taking the square root of a negative number, an appropriate message is printed on the standard error stream and the program exits with a nonzero status.

NAME
ms – manuscript macro package

USAGE
nroff –ms *file* ...

DESCRIPTION
The **nroff** macro package **ms** formats manuscripts. The *nroff Text Processor Tutorial* describes the **ms** macros in detail. **ms** includes the following macros:

.AB	Abstract begin
.AE	Abstract end
.AI	Author's institution
.AU	Author
.B	Boldface font
.BD	Block-centered display
.BT	Bottom title
.BX	Draw a box
.CD	Centered display
.CO	COHERENT
.DA	Date
.DE	Display end
.DM	Display monospace
.DS	Display start
.FE	Footnote end
.FS	Footnote start
.I	Italic font
.ID	Indented display
.II	Index invisible
.IP	Indented paragraph
.IV	Index visible
.KE	Keep end
.KF	Start floating keep
.KS	Keep start
.LD	Left display
.LG	Larger size
.LP	Left paragraph
.ND	New (or no) date
.NH	Numbered heading
.NL	Normal size

Convention

.PP	Paragraph
.PT	Page title
.QE	Quoted paragraph end
.QP	Quoted paragraph
.QS	Quoted paragraph start
.R	Roman font
.RE	Relative indent end
.RS	Relative indent start
.SH	Subheading
.SM	Smaller size
.TA	Set tabs in ens
.TL	Title
.UL	Underline

FILES
/usr/lib/tmac.s

SEE ALSO
man
COHERENT Command Manual: **nroff**
nroff Text Processor Tutorial

File Format

NAME
mtab.h – currently mounted file systems

USAGE
#include <mtab.h>

DESCRIPTION
The file **/etc/mtab** contains an entry for each file system mounted by the **mount** command. This does not include the root file system, which is already mounted when the system boots.

Both the **mount** and **umount** commands use the following structure, defined in the **mtab.h** header file. It contains the name of each special file mounted, the directory upon which it is mounted, and any flags passed to **mount** (such as read only).

```
#define MNAMSIZ 32
struct  mtab {
        char    mt_name[MNAMSIZ];
        char    mt_special[MNAMSIZ];
        int     mt_flag;
};
```

FILES
/etc/mtab
<mtab.h>

SEE ALSO
COHERENT Command Manual: **mount, umount**

NAME
mtype – return symbolic machine type

USAGE
#include <mtype.h>

char *
mtype(*type*)
int *type*;

DESCRIPTION
mtype takes an integer machine *type* and returns an ASCII string containing the symbolic name of the machine. The header file **mtype.h** defines the possible machine types. For example,

```
mtype(M_PDP11)
```

returns the string "**PDP-11**".

FILES
<mtype.h>

SEE ALSO
l.out.h
COHERENT Command Manual: **ld**

DIAGNOSTICS
mtype returns **NULL** to indicate a bad machine type.

NAME
nlist – symbol table lookup

USAGE
#include <l.out.h>

nlist(*file, nlp*)
char ****file*;**
struct nlist ****nlp*;**

DESCRIPTION
nlist searches the name list (symbol table) of the load module specified by *file* for each symbol in the array to which *nlp* points. For example, the **ps** command uses this routine on the system load module (**/coherent**) to obtain the addresses of system tables in memory (**/dev/mem**).

The *nlp* argument points to an array of **nlist** structures, terminated by a structure with the null string "" as its **n_name** member. The **l.out.h** header file defines **nlist** as follows:

```
#define NCPLN   16

struct  nlist {
        char    n_name[NCPLN];
        int     n_type;
        unsigned n_value;
};
```

The caller should set the **n_name** entry; **nlist** will fill in the other entries. **nlist** sets both **n_type** and **n_value** to zero if the symbol is not found.

FILES
<l.out.h>

SEE ALSO
l.out.h
COHERENT Command Manual: **nm, strip**

DIAGNOSTICS
If *file* is not a load module or has had its symbol table stripped, all returned **n_type** and **n_value** entries will be 0.

Device Driver

NAME
null – discard data

DESCRIPTION
All data written to the special file **/dev/null** is thrown away (sent to the ''bit bucket''). This is useful, for example, to test a program's side effects while ignoring its output.

A read from file **/dev/null** returns end of file (0 bytes of data). The shell **sh** uses **/dev/null** as input to background processes.

FILES
/dev/null

SEE ALSO
COHERENT Command Manual: **sh**

System Call

NAME
open – open a file

USAGE
open(*file, type***)**
char **file*;
int *type*;

DESCRIPTION
open prepares a *file* for I/O. When successful, **open** returns a file descriptor (a small positive integer), which identifies the open *file* to subsequent **read, write** or **close** calls.

The *type* argument can be **0** for reading, **1** for writing, or **2** for both reading and writing. After a *file* is opened, I/O will occur at the start, or byte 0.

SEE ALSO
close, creat, fopen, read, write

DIAGNOSTICS
open returns −1 if the file is nonexistent, if the caller lacks permission, or if some system resource is exhausted.

NAME
passwd – password file format

DESCRIPTION
The password file **/etc/passwd** describes the user name, password, user id, group id, initial working directory and initial program for each user of the COHERENT system. The **login** and **passwd** commands use this information.

Each entry consists of a single line. Each line consists of seven colon-separated ASCII fields:

user_name:*password*:*uid*:*gid*::*dir*:*sh*

The *user_name* gives the login name of the user. The optional *password* gives the encrypted password of the user. The *uid* and *gid* fields give the numerical user id and group id of the user. The fifth field is unused. The *dir* gives the initial **$HOME** directory for the user. The *sh* gives the pathname of the initial program for the user; if omitted, the command line interpreter **/bin/sh** is assumed.

Passwords are encrypted with **crypt**, so the **passwd** file is generally readable.

FILES
/etc/passwd

SEE ALSO
crypt, getpwent, group
COHERENT Command Manual: **chgrp, login, newgrp, passwd**

System Call

NAME
pause – wait for signal

USAGE
pause( )

DESCRIPTION
pause suspends execution until the process receives a signal. Signals could come from **kill**, from **alarm**, or from the controlling terminal.

SEE ALSO
alarm, kill, signal, sleep

libc Library

NAME
perror – system call error messages

USAGE
#include <errno.h>

perror(*string***)**
char ****string*;**

extern int *sys_nerr*;
**extern char ****sys_errlist[]*;

DESCRIPTION
perror prints an error message on the standard error output (file descriptor 2). The message consists of the argument *string*, followed by a brief description of the last system call which failed. The external variable **errno** contains the last error number. Normally, *string* is the name of the command that failed or a file name.

The external array **sys_errlist** gives the list of messages used by **perror**. The external **sys_nerr** gives the number of messages in the list.

FILES
<errno.h>

SEE ALSO
errno

System Call

NAME
pipe – create a pipe

USAGE
pipe(*fd***)**
int *fd***[2];**

DESCRIPTION
A *pipe* is an interprocess communication mechanism. **pipe** creates a pipe, typically to construct pipelines in the shell **sh**. **pipe** fills in *fd***[0]** and *fd***[1]** with *read* and *write* file descriptors, respectively.

The file descriptors allow the transfer of data from one or more writers to one or more readers. Pipes are buffered to 5120 bytes. If more than 5120 bytes are written into the pipe, the **write** call will not return until the reader has removed sufficient data for the **write** to complete. If a **read** occurs on an empty pipe, its completion awaits the writing of data.

When all writing processes close their write file descriptors, the reader receives an end of file indication. A write on a pipe with no remaining readers generates a **SIGPIPE** signal to the caller.

pipe is generally called just before **fork**. Once the parent and child processes are created, the unused file descriptors should be closed in each process.

SEE ALSO
close, read, signal, write
COHERENT Command Manual: **sh**

DIAGNOSTICS
pipe returns 0 on successful calls, or –1 if it could not create the pipe.

NAME
pnmatch – string pattern matching

USAGE
int
pnmatch(*string, pattern, flag***)**
char **string,* **pattern*;
int *flag*;

DESCRIPTION
pnmatch matches *patterns*, which are a simplified form of regular expressions. The shell **sh** uses patterns for file name expansion and **case** statement expressions.

pnmatch returns 1 if the given *pattern* matches the given *string*. It returns 0 if the *pattern* does not match the *string*.

Each character in *pattern* must exactly match a character in *string*, except for metacharacters, which have a special meaning in the *pattern*. The metacharacter '?' matches any one character. The metacharacter '*' matches a string containing any number of any characters, including the null string (containing no characters). A set of characters enclosed between '[' and ']' matches any one character of the set. Sets of characters may include ranges, such as '[a–z]' for lower-case letters. A backslash ('\') before a metacharacter removes its special meaning.

The *flag* argument must be either 0 or 1. When *flag* is 0, the *pattern* must match the *string* exactly. When *flag* is 1, the *pattern* can match any part of the *string*. In this case, additional metacharacters '^' and '$' match the beginning and end of the *string*, respectively.

SEE ALSO
COHERENT Command Manual: **grep, learn, sh**

NOTES
flag must be 0 or 1 for predictable results.

NAME
pclose, popen – establish stream between processes

USAGE
#include <stdio.h>

pclose(*fp*)
FILE *_fp_;

FILE *
popen(*command, how*)
char *_command_, *_how_;

DESCRIPTION
popen is similar to **fopen**, except that the opened object is a command line to the shell **sh** rather than a file. **popen** creates a pipe. The caller can read the standard output of the *command* when *how* is "**r**", or write to the standard input of the *command* when *how* is "**w**". **popen** returns a stream which may be read or written.

pclose closes a stream opened by **popen**, awaiting the completion of the child process and performing other cleanup. It returns the exit status of the *command*.

FILES
<stdio.h>

SEE ALSO
fclose, fopen, pipe, system, wait
COHERENT Command Manual: **sh**

DIAGNOSTICS
popen returns **NULL** if the link to the *command* could not be established.

pclose returns −1 if *fp* was not created by a previous **popen** call. Otherwise, **pclose** returns the exit status of the *command*, in the format described in **wait**: exit status in the high byte, signal information in the low byte.

NAME
fprintf, printf, sprintf – formatted output

USAGE
#include <stdio.h>

fprintf(*fp, format [, arg] ...***)**
FILE ****fp*;**
char ****format*;**

printf(*format [, arg] ...***)**
char ****format*;**

sprintf(*string, format [, arg] ...***)**
char ****string, *format*;**

DESCRIPTION
Each of these routines uses the *format* string to specify a format for output conversion of each remaining *arg*. **fprintf** writes characters to the given stream *fp*, while **printf** writes to the standard output. **sprintf** puts its output into the given *string* and appends a null character ('\0').

Each routine reads characters one at a time from the *format* string. Each copies any character other than a conversion specification to the output directly. The '%' character identifies the start of a conversion specification. Each conversion uses one or more of the remaining *arg* arguments. It is essential for users to ensure type matching between the arguments and the conversion specifications.

Output modifiers and the desired conversion type may follow the '%' character. The following modifiers, in this order, may precede the conversion type:

1) An optional minus sign '–' indicates left justification (rather than the default right justification) of the output field.

2) An optional string of digits gives the *width* of the output field. Normally, the field is padded with spaces to the field width, on the left unless the above minus sign is specified. If the field width begins with **'0'**, padding is with **'0'** characters rather than spaces. If the width specification is an

asterisk '*', the routine uses the next *arg* as an integer giving the width.

3) An optional period '.' followed by a string of digits indicates the *precision*. For floating point (**e**, **f** and **g**) conversions, the precision is the number of digits printed after the decimal point. For string (**s**) conversions, the precision is the maximum number of characters used from the string. If the precision specification is an asterisk '*', the routine uses the next *arg* as an integer giving the precision.

4) The letter 'l' before any integer conversion (**d**, **o**, **x**, or **u**) indicates that the argument is a **long** rather than an **int**. Capitalizing the conversion type has the same effect.

The routines recognize the following conversion types:

%	Output a '%' character. No arguments are processed.
c	Convert the **int** argument to a character.
d	Convert the **int** argument to signed decimal.
D	Convert the **long** argument to signed decimal.
e	Convert the **float** or **double** argument to exponential form. The format is *d.dddddd*e*sdd*, where there is always one digit before the decimal point and as many as the *precision* after it (default: six). The exponent sign *s* may be either '+' or '−'.
f	Convert the **float** or **double** argument to a representation with an optional leading minus sign '−', at least one decimal digit, a decimal point ('.'), and optional decimal digits after the decimal point. The number of digits after the decimal point is the *precision* (default: six).
g	Convert the **float** or **double** argument to whichever of the formats **d**, **e**, or **f** loses no significant precision and takes the least space.
o	Convert the **int** argument to unsigned octal.
O	Convert the **long** argument to unsigned octal.
r	The next argument points to an array of new arguments that may be used recursively. The first argument of the list

is a **char** * containing a new format string. When the list is exhausted, the routine continues from where it left off in the original format string.

s Output the string to which the **char** * argument points. Reaching either the end of the string, indicated by a null character, or the specified *precision* will terminate output. If no *precision* is given, only the end of the string will terminate.

u Convert the **int** argument to unsigned decimal.

U Convert the **long** argument to unsigned decimal.

x Convert the **int** argument to unsigned hexadecimal.

X Convert the **long** argument to unsigned hexadecimal.

Examples
The following examples show the use of **printf** for integer, string, and floating point conversions, respectively:

```
printf("%d %u %06x\n", 0123456, 0123456, 0123456);
printf("%-7.5s %7.5s %.5s\n", "random", "random", "random");
printf("%g %7.2g %e %f\n", 23.546, 23.546, 23.546, 23.546);
```

In the output from these examples, an underscore character '_' replaces each space character, for increased clarity:

```
-22738_42798_00A72E
rando_____rando_rando
23.546000___23.55_2.354600e+01_23.546000
```

FILES
<stdio.h>

SEE ALSO
ecvt, putc, puts, scanf

NOTES
The output *string* passed to **sprintf** must be large enough to hold all output characters.

System Call

NAME
profil – profile process execution

USAGE
profil(*buffer, size, base, scale*)
short **buffer*;
int *size, base*;
unsigned *scale*;

DESCRIPTION
profil causes the execution of a program to produce a histogram of program counter (**pc**) locations, as sampled by the system clock handler (up to **HZ** times per second). The process can profile activity in one area of the process code segment, starting at location *base*. **profil** sets up *size*/**sizeof(short)** counters starting at location *buffer*, each associated with a segment of memory called a *bin*. The system increments a counter when it finds the **pc** in the corresponding bin. The *scale* determines bin size; it is the reciprocal of the number of bytes per bin, represented as a fixed-point number with assumed binary point to the left of bit 16. The counter incremented is

c = (**pc** – *base*)*(*scale*/2) / 2^16

If **pc** < *base* or c > = *size*/**sizeof(short)**, the system increments no counter.

profil turns off profiling if *scale* is 0. If *scale* is 2, it profiles the entire code segment and increments the single counter each clock tick. If *scale* is 0177777 ((**unsigned**)65535), there is essentially one counter for each **pc** location.

SEE ALSO
COHERENT Command Manual: **prof, time**

DIAGNOSTICS
profil returns –1 and disables profiling if arguments are invalid.

System Call

NAME

ptrace – trace process execution

USAGE

#include <signal.h>

ptrace(*command, pid, location, value***)**
int *command, pid***;**
int ****location*;**
int *value***;**

DESCRIPTION

ptrace provides a parent process with primitives to monitor and alter the execution of a child process. These primitives typically are used by a debugger such as **db**, which needs to examine and change memory, plant breakpoints, and single-step the child process being debugged.

Once a child process indicates it wishes to be traced, its parent issues various *command*s to control the child. *pid* identifies the affected process. The parent may issue a command only when the child process is in a stopped state, which occurs when the child encounters a signal. A special return value of 0177 from **wait** informs the parent that the child has entered the stopped state. The parent may then examine or change the child process memory space or restart the process at any point.

When the child process issues an **exec**, the child stops with signal **SIGTRAP** to enable the parent to plant breakpoints. The set user id and set group id modes are ineffective when a traced process performs an **exec**.

The following list describes each available *command*. A *command* ignores any arguments not mentioned.

0 This is the only *command* the child process may issue. It tells the system that the child wishes to be traced. Parent and child must agree that tracing should occur to achieve the desired effect. Only the *command* argument is significant.

1,2 The **int** at *location* is the return value. Command 1 signifies that *location* is in the instruction space, while command 2 signifies *data* space. Often these two spaces are equivalent.

3 The return value is the **int** of the process description, as defined in **sys/uproc.h**. This call may be used to obtain values such as hardware register contents and segment allocation information.

4,5 Modify the child process's memory by changing the **int** at *location* to *value*. Command 4 means instruction space and command 5 means data space. Shared segments may be written only if no other executing process is using them.

6 Modify the **int** at *location* in the process description area, as with command 3. The permissible values for *location* are restricted to such things as hardware registers and bits of machine status registers which the user may safely change.

7 This command restarts the stopped child process after it encounters a signal. The process resumes execution at *location*, or from where the process was stopped if *location* is **(int *)1**. *value* gives a signal number that the process receives as it restarts. This is normally the number of the signal which caused the process to stop, fetched from the process description area by a 3 command. If *value* is 0, the effect of the signal is ignored.

8 Force the child process to exit.

9 Like command 7, except that the child stops again with signal **SIGTRAP** as soon as practicable after the execution of at least one instruction. The actual hardware method used to implement this command varies from machine to machine, explaining the imprecise nature of its definition. This call may provide part of the basis for breakpoints.

FILES
<signal.h>
<sys/uproc.h>

SEE ALSO
exec, signal, wait
COHERENT Command Manual: **db**

DIAGNOSTICS
ptrace returns −1 if *pid* is not the process id of an eligible child process or if some other argument is invalid or out of bounds. Some commands may return an arbitrary data value, in which case

errno should be checked to distinguish a return value of −1 from an error return.

NOTES
There is no way to specify which signals should not stop the process.

STDIO Library

NAME
fputc, putc, putchar – write character to stream

USAGE
#include <stdio.h>

int
fputc(*c, fp***)**
char *c***;**
FILE ****fp*;**

int
putc(*c, fp***)**
char *c***;**
FILE ****fp*;**

int
putchar(*c***)**
char *c***;**

DESCRIPTION
putc is a macro that writes a single character *c* onto file stream *fp*, returning that character upon success. **putchar** is a macro expanding to **putc(***c***, stdout)**, so it writes a single character onto the standard output.

fputc is a genuine function, whose body is **putc**.

FILES
<stdio.h>

SEE ALSO
fwrite, getc, printf, puts, putw

DIAGNOSTICS
These functions return **EOF** on write errors.

NOTES
Because **putc** and **putchar** are macros, side effects in arguments may not work as expected.

NAME
fputs, puts – write string to stream

USAGE
#include <stdio.h>

fputs(*string, fp***)**
char ****string*;**
FILE ****fp*;**

puts(*string***)**
char ****string*;**

DESCRIPTION
puts appends a newline character to its *string* argument and writes the result on the standard output.

fputs writes *string* on the stream given by *fp*. Unlike **puts,** it does not append a newline character.

FILES
<stdio.h>

SEE ALSO
fwrite, gets, printf, putc

NOTES
For historical reasons, **fputs** outputs the string unchanged, while **puts** appends a newline.

NAME
fputw, putw – write integer to stream

USAGE
#include <stdio.h>

fputw(*word, fp***)**
int *word***;**
FILE **fp***;**

putw(*word, fp***)**
int *word***;**
FILE **fp***;**

DESCRIPTION
The macro **putw** writes *word* (an **int**) to the stream *fp*. It returns the value written. **fputw** has the same effect, but is a function rather than a macro.

FILES
<stdio.h>

SEE ALSO
ferror, fwrite, putc, puts

DIAGNOSTICS
Both routines return the value **EOF** on errors. A call to **ferror** may be necessary to distinguish this value from a valid data item.

NOTES
Because **putw** is a macro, side effects in arguments may not work as expected.

NAME
qsort, shellsort – in-memory sorting

USAGE
qsort(*data, nitems, size, comp***)**
char ****data*;**
int *nitems, size***;**
int (**comp***)();**

shellsort(*data, nitems, size, comp***)**
char ****data*;**
int *nitems, size***;**
int (**comp***)();**

DESCRIPTION
qsort and **shellsort** are generalized algorithms for sorting arrays of data in memory. The **sort** command sorts arrays of data too large to fit into memory. **qsort** is C. A. R. Hoare's Quicksort algorithm, and is preferable for most applications. **shellsort** is Shell's method, and has an identical calling sequence.

Each routine sorts a sequential array of memory called *data*, divided up into *nitems* parts of *size* bytes each. Each routine compares pairs of items and exchanges them as required.

The user-supplied routine to which *comp* points performs the comparison. It is called repeatedly, as follows:

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

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

SEE ALSO
COHERENT Command Manual: **sort**
Donald Knuth, *The Art of Computer Programming, Vol. 3: Sorting and Searching*, Addison-Wesley, 1971, pp. 84 ff., 114 ff.

NAME
rand, srand – random number generator

USAGE
rand()

srand(*seed*)
int *seed*;

DESCRIPTION
rand is a linear congruential pseudo-random number generator. It returns integers in the range 0 to 2^15 – 1, and purportedly has a period of 2^32. **srand** initializes ("seeds") the sequence of pseudo-random numbers. Unequal values of *seed* initialize different sequences.

SEE ALSO
Donald Knuth, *The Art of Computer Programming, Vol. 2: Seminumerical Algorithms*, 2nd ed., Addison-Wesley, 1981.

System Call

NAME
read – read from a file

USAGE
read(*fd, bufp, nb***)**
int *fd***;**
char **bufp***;**
int *nb***;**

DESCRIPTION
read tries to read up to *nb* bytes of data from the file given by file descriptor *fd* into the data segment at address *bufp*. The amount of data actually read may be smaller than requested if the read pointer hits end of file, or if the file is a device with certain properties (e.g. a terminal).

For block devices and regular files, the **read** occurs at the current seek position in the file, which was set by the last **read** or **lseek** call. **read** advances the seek pointer by the number of characters actually read.

SEE ALSO
lseek, open, write

DIAGNOSTICS
read returns the number of bytes read for successful calls; thus, zero bytes signals end of file. It returns –1 if an error occurs; bad file descriptor, bad *bufp* address, and physical read error are among the possibilities.

NAME
fscanf, scanf, sscanf – formatted input

USAGE
#include <stdio.h>

fscanf(*fp, format [, arg] ...***)**
FILE ****fp*;**
char ****format*;**

scanf(*format [, arg] ...***)**
char ****format*;**

sscanf(*string, format [, arg] ...***)**
char ****string*;**
char ****format*;**

DESCRIPTION
Each of these routines uses the *format* string to specify a format for input conversion. Each converted input item is assigned through the next specified *arg*, which must be a pointer. **fscanf** reads input from the stream *fp*, while **scanf** reads from the standard input. **sscanf** reads from the given *string*.

Each routine takes characters one at a time from the *format* string. White space characters in the *format* are ignored. Other characters except '%' match non-white space characters in the input. The '%' character identifies the start of a conversion specification. Each conversion may use one or more of the remaining *arg* arguments. It is essential for users to ensure type matching between the arguments and the conversion specifications.

Each routine terminates when it encounters the end of the *format* string or when the input does not match a specification. Each returns the number of successful assignments.

After the '%' character, there may be characters indicating the width of the input field and the conversion type. A field is delimited by white space (space, tab, newline) or by the given field width, if any. Newlines are white space, so the input can include more than one line. The following modifiers, in this order, may precede the conversion type:

1) An optional '*', indicating that the next input field should be skipped (rather than assigned to the next *arg*).

2) An optional string of decimal digits, specifying a maximum field width.

3) An **l**, specifying that the next input item is a **long** object rather than an **int** object. Capitalizing the conversion character has the same effect.

The routines recognize the following conversion types:

c	Assign the next input character to the next *arg*, which should be of type **char ***. This conversion does not skip white space characters in the input.
d	Assign the decimal integer from the next input field to the next *arg*, which should be of type **int ***.
D	Assign the decimal integer from the next input field to the next *arg*, which should be of type **long ***.
e	Assign the floating point number from the next input field to the next *arg*, which should be of type **float ***.
E	Assign the floating point number from the next input field to the next *arg*, which should be of type **double ***.
f	Same as **e** or **double ***.
F	Same as **E**.
o	Assign the octal integer from the next input field to the next *arg*, which should be of type **int ***.
O	Assign the octal integer from the next input field to the next *arg*, which should be of type **long ***.
s	Assign the string from the next input field to the next *arg*, which should be of type **char ***.
x	Assign the hexadecimal integer from the next input field to the next *arg*, which should be of type **int ***.
X	Assign the hexadecimal integer from the next input field to the next *arg*, which should be of type **long ***.
[*string*]	Assign characters from the input until the first character not in the given *string* to the next *arg*, which should be of

type **char ***. This conversion does not skip white space characters in the input.

[^*string*] Assign characters from the input until the first character in the given *string* to the next *arg*, which should be of type **char ***. This conversion does not skip white space characters in the input.

FILES
<stdio.h>

SEE ALSO
getc, printf

STDIO Library

NAME
setbuf – enable/disable stream buffering

USAGE
#include <stdio.h>

setbuf(*fp, buffer***)**
FILE ****fp*;**
char ****buffer*;**

DESCRIPTION
The standard I/O library STDIO automatically buffers all reading and writing on streams, except the standard error stream. The system uses **malloc** to allocate the buffer, a **char** array of **BUFSIZ** characters in length. Because it may be desirable to use a different allocation strategy or to buffer an output stream unconditionally, the system provides the **setbuf** routine.

The arguments to **setbuf** are a stream pointer *fp* and a *buffer* to be associated with the stream. The call should be issued after the stream has been opened but before any input or output request has been issued.

Passing a *buffer* pointer of **NULL** disables implicit buffering on a stream.

FILES
<stdio.h>

SEE ALSO
fopen, malloc, ttyname

NAME
longjmp, setjmp – non-local goto

USAGE
#include <setjmp.h>

longjmp(*env, rval***)**
jmp_buf *env***;**
int *rval***;**

setjmp(*env***)**
jmp_buf *env***;**

DESCRIPTION
The function call is the only mechanism the C language provides for transferring control between separately compiled modules. This mechanism is inadequate for some purposes, such as handling unexpected errors or interrupts at lower levels of a program. The COHERENT system therefore includes the **setjmp** and **longjmp** routines to implement a non-local *goto* facility.

setjmp saves its environment in *env* and returns value 0. **longjmp** restores the environment saved by a previous **setjmp** and returns value *rval* to the caller of **setjmp**, as if the **setjmp** call had just returned (again). **longjmp** must not restore the environment of a routine which has already returned.

The type declaration for **jmp_buf** is in the **setjmp.h** header file. The environment saved includes the program counter, stack pointer and stack frame. The routines do not affect any other variables of a program.

FILES
<setjmp.h>

SEE ALSO
signal

System Call

NAME
setgid, setuid – set group id and user id

USAGE
setgid(*id***)**

setuid(*id***)**

DESCRIPTION
setuid sets the real user id and the effective user id of the calling process to the given *id*. Similarly, **setgid** sets the group id. These calls can be used to turn off set user id and set group id privileges (see **exec**).

The call is allowed if the real id of the calling process matches *id* or is the superuser.

SEE ALSO
exec, getuid, login

DIAGNOSTICS
Each call returns 0 on success, or –1 on failure.

System Call

NAME

signal – specify disposition of a signal

USAGE

#include <signal.h>
#include <msig.h>

int (*signal(*signum, action***))()**
int *signum***;**
int (**action***)();**

DESCRIPTION

A process can receive a *signal*, or interrupt, from a hardware exception, from terminal input, or from a **kill** call made by a process. A hardware exception might be an illegal instruction code or a bad machine address, caught by the segmentation hardware. A terminal interrupt character, described in detail in **tty**, generates a process interrupt (and in one case a core dump file for debugging purposes).

When a process receives a signal, it performs an appropriate *action*. The default action **SIG_DFL** causes the process to terminate. **signal** specifies a new *action* for signal number *signum*, and returns a pointer to the previous action. *action* points to a function which will handle the signal, in the manner of a hardware interrupt handler. The action **SIG_IGN** causes a signal to be ignored. **SIGKILL** can be neither caught nor ignored.

With the exception of **SIGILL** and **SIGTRAP**, caught signals are reset to the default action **SIG_DFL**. To catch a signal again, the specified *action* must reissue the **signal** call.

The following list gives machine independent signals by symbolic name (defined in the **signal.h** or **msig.h** header file), numeric value, and description. Signals marked by '*' produce a core dump if the action is **SIG_DFL**.

SIGHUP	1	hangup
SIGINT	2	interrupt
SIGQUIT	3*	quit
SIGALRM	4	alarm clock
SIGTERM	5	termination
SIGREST	6	restart indication

SIGSYS	7*	bad system call argument
SIGPIPE	8	write on closed pipe
SIGKILL	9	kill
SIGTRAP	10*	breakpoint
SIGSEGV	11*	segmentation violation

The following signals are specific to the PDP-11 version of the system:

SIGILL	12*	illegal instruction
SIGIOT	13*	IOT instruction
SIGEMT	14*	EMT instruction
SIGFPE	15*	floating point exception
SIGBUS	16*	bus error

The following signals are specific to the Zilog Z8002 version of the system:

SIGUNI	12*	unimplemented instruction
SIGPRV	13*	privileged instruction
SIGNVI	14*	non-vectored interrupt
SIGPAR	15*	parity error

The following signals are specific to the Zilog Z8001 version of the system:

SIGEPA	12*	extended processor trap
SIGPRV	13*	privileged instruction
SIGNVI	14*	non-vectored interrupt
SIGNMI	15*	non-maskable interrupt (not in all versions)

The following signals are specific to the Intel 8086 or 8088 version of the system:

SIGDIVE	12*	divide error
SIGOVFL	13*	overflow

A signal may be caught during a system call which has not yet returned. In this case, the system call appears to fail, with **errno** set to **EINTR**. If desired, such an interrupted system call may be reissued. System calls which may be interrupted in this way include **pause**, **read** on a device such as a terminal, **write** on a pipe, and **wait**.

FILES
<msig.h>
<signal.h>

SEE ALSO
kill, ptrace
COHERENT Command Manual: **sh**

DIAGNOSTICS
signal returns a pointer to the previous action on success. It returns **(int)** – 1 for invalid *signum*.

libc Library

NAME
signame – signal meanings

USAGE
#include <signal.h>

extern char *signame[NSIG + 1];

DESCRIPTION
When a program terminates abnormally, its parent process receives a byte of termination information from the **wait** call. This byte contains a signal number, as defined in the **signal.h** header file. For example, **SIGINT** indicates an interrupt from the terminal.

The **signame** array, indexed by signal number, contains strings which give the meaning of each signal. Thus, **signame[SIGINT]** is the string ''**interrupt**''. For portability reasons, all programs which wait on child processes (such as the shell **sh**) should use **signame**.

FILES
<signal.h>

SEE ALSO
signal, wait
COHERENT Command Manual: **sh**

libm Library

NAME
acos, asin, atan, atan2, cos, sin, tan – trigonometric functions

USAGE
#include <math.h>

double
acos(*arg*)
double *arg*;

double
asin(*arg*)
double *arg*;

double
atan(*arg*)
double *arg*;

double
atan2(*num, den*)
double *num, den*;

double
cos(*radian*)
double *radian*;

double
sin(*radian*)
double *radian*;

double
tan(*radian*)
double *radian*;

DESCRIPTION
The trigonometric functions are **sin**, **cos**, and **tan**. The argument *radian* should be in radian measure.

The inverse trigonometric functions are **asin**, **acos**, and **atan**. The argument of **asin** or **acos** should be in the range [−1., 1.], while the argument of **atan** may be any real number. The result is in the range [− pi/2, pi/2] for **asin**, in the range [0, pi] for **acos**, and in the range [− pi/2, pi/2] for **atan**.

The **atan2** function returns **atan** of the quotient of its arguments, *num/den*, with the result in the range [− pi, pi]. The sine of the result will have the same sign as *num*, and the cosine of the result will have the same sign as *den*.

FILES
<math.h>

SEE ALSO
sinh

DIAGNOSTICS
Out of range arguments set **errno** to **EDOM** and return 0. **tan** returns a very large number where it is singular and sets **errno** to **ERANGE**.

libm Library

NAME
cosh, sinh, tanh – hyperbolic functions

USAGE
#include <math.h>

double
cosh(*z*)
double *z*;

double
sinh(*z*)
double *z*;

double
tanh(*z*)
double *z*;

DESCRIPTION
sinh, **cosh**, and **tanh** compute the hyperbolic sine, hyperbolic cosine, and hyperbolic tangent, respectively. In each case, the argument *z* is in radian measure.

FILES
<math.h>

SEE ALSO
log, sin

DIAGNOSTICS
Both **sinh** and **cosh** set **errno** to **ERANGE** and return a huge value with the same sign as the actual result when overflow occurs.

libc Library

NAME
sleep – suspend execution

USAGE
sleep(*seconds***)**
unsigned *seconds***;**

DESCRIPTION
sleep suspends execution of the calling process for at least the number of *seconds* specified. The system resumes processing at the next whole-second interval after the specified time has elapsed.

sleep uses the **alarm** and **pause** system calls. Any alarm pending during the **sleep** interval is executed properly, after which the **sleep** continues for the remaining time.

SEE ALSO
alarm, pause, signal

System Call

NAME
sload, suload – load/unload device driver

USAGE
#include <con.h>

sload(*major, file, conp***)**
int *major***;**
char ****file*;**
CON ****conp*;**

suload(*major***)**
int *major***;**

DESCRIPTION
The COHERENT system accesses all devices through drivers residing in the system. Except for the root device, drivers must be explicitly loaded before use; this operation does not involve re-booting.

sload loads the driver given by *file* as device number *major*. This number uniquely identifies the driver to the system. The *conp* argument is a reference to a **CON** structure, as defined in the **con.h** header file. It describes standard entry points and gives other information on the driver. Normally the *major* and *conp* parameters are obtained from the driver load module; this is the method used by the **load** command.

file must be in the correct format. Usually, it is created using the **−k** option to the **ld** command.

suload unloads the driver identified by *major*, which was previously loaded by a **sload** call.

Both calls are restricted to the superuser.

FILES
<con.h>
/drv/*

SEE ALSO
init, l.out.h
COHERENT Command Manual: **ld, load, uload**

DIAGNOSTICS
The routines return 0 upon successful loading or unloading of the appropriate driver, or −1 on errors. **sload** errors include *file* nonexistent, parameter (such as *major*) out of range, driver already loaded for *major*, or *file* not a file containing a proper driver. **suload** fails if the driver *major* is not loaded.

NOTES
Because of hardware restrictions, the COHERENT system does not support loadable device drivers on systems based on the 8086 or 8088 processors (such as the IBM Personal Computer). The /**drv** directory and the **sload** and **suload** system calls do not exist on such systems.

System Call

NAME
fstat, stat – find file attributes

USAGE
#include <sys/stat.h>

fstat(*fd, statp***)**
int *fd***;**
struct stat ****statp*;**

stat(*file, statp***)**
char ****file*;**
struct stat ****statp*;**

DESCRIPTION
stat and **fstat** each return a structure containing attributes of a file, including protection information, file type, and file size. **stat** takes a pathname given by *file*, while **fstat** takes a file descriptor *fd*.

stat and **fstat** also take a pointer *statp* to a **stat** structure, as described in the **stat.h** header file. The following summarizes the structure and defines the permission and file type bits.

```
struct  stat {
        dev_t   st_dev;
        int_t   st_ino;
        unsigned short  st_mode;
        short   st_nlink;
        short   st_uid;
        short   st_gid;
        dev_t   st_rdev;
        size_t  st_size;
        time_t  st_atime;
        time_t  st_mtime;
        time_t  st_ctime;
};
```

```
#define S_IFMT  0170000       /* file types */
#define S_IFREG 0100000       /* ordinary file */
#define S_IFDIR 0040000       /* directory */
#define S_IFCHR 0020000       /* character special */
#define S_IFBLK 0060000       /* block special */
#define S_ISUID 0004000       /* set user id */
#define S_ISGID 0002000       /* set group id */
#define S_ISVTX 0001000       /* save text bit */
#define S_IREAD 0000400       /* owner read permission */
#define S_IWRITE 000200       /* owner write permission */
#define S_IEXEC 0000100       /* owner execute permission */
```

The **st_dev** and **st_ino** entries together form a unique description of the file. The former is the device on which the file and its i-node reside, while the latter is the index number of the file. The **st_mode** entry gives the permission bits, as outlined above. The **st_nlink** entry gives the number of links to the file. The user id and group id of the owner are **st_uid** and **st_gid**, respectively. The **st_rdev** entry, valid only for special files, holds the major and minor numbers for the file.

The **st_size** entry gives the size of the file in bytes. For a pipe, the size is the number of bytes waiting to be read from the pipe.

Three entries for each file give the last occurrences of various events in the file's history. The **st_atime** entry gives the last access (read or write). The **st_mtime** entry gives the last modification (write for files, create or delete entry for directories). The **st_ctime** entry gives the last change to the attributes (not including times and size).

FILES
<sys/stat.h>

SEE ALSO
chmod, chown, open
COHERENT Command Manual: **ls**

DIAGNOSTICS
The routines return −1 if the file is not found or if the *statp* pointer is invalid.

System Call

NAME
stime – set the time

USAGE
#include <sys/types.h>

stime(*timep***)**
time_t **timep*;

DESCRIPTION
stime sets the system time. The *timep* argument is a pointer to a **time_t** (actually a **long**) which contains the number of seconds since midnight GMT of January 1, 1970.

stime is restricted to the superuser.

FILES
<sys/types.h>

SEE ALSO
ctime, ftime, stat, utime
COHERENT Command Manual: **date**

DIAGNOSTICS
stime returns –1 on error, 0 otherwise.

libc Library

NAME
index, rindex, strcat, strcmp, strcpy, strlen, strncat, strncmp, strncpy – string manipulation

USAGE
char *
index(*string, c***)**
char ****string*;**
char *c***;**

char *
rindex(*string, c***)**
char ****string*;**
char *c***;**

char *
strcat(*string1, string2***)**
char ****string1*, ****string2***;**

strcmp(*string1, string2***)**
char ****string1*, ****string2***;**

char *
strcpy(*string1, string2***)**
char ****string1*, ****string2***;**

strlen(*string***)**
char ****string*;**

char *
strncat(*string1, string2, n***)**
char ****string1*, ****string2***;**
unsigned *n***;**

strncmp(*string1, string2, n***)**
char ****string1*, ****string2***;**
unsigned *n***;**

char *
strncpy(*string1, string2, n***)**
char ****string1*, ****string2***;**
unsigned *n***;**

libc Library

DESCRIPTION
These routines act on *strings*, which are arrays of characters, usually terminated by a null character ('\0'). Their implementations may exploit special machine features, so use of these routines is encouraged.

strcmp compares *string1* and *string2* lexicographically. It returns 0 if the strings are the same, −1 if *string1* is less than *string2*, and 1 otherwise. This routine is compatible with the ordering routine desired by **qsort**. **strncmp** operates exactly like **strcmp**, except that a maximum of *n* characters are significant in the comparison. Comparison always ends at a null character.

strcpy copies the contents of *string2*, up to a null byte, to *string1*. The order of the arguments is reminiscent of an assignment statement. **strncpy** acts similarly, but copies exactly *n* characters to *string1*. If *string2* is shorter than *n* characters in length, it pads *string1* with null bytes. If *string2* is longer than *n* characters, the result may not be null-terminated. Both routines return the result *string1*.

strcat copies all characters in *string2* to the end of *string1*. It returns *string1*, which is the concatenation of the argument strings. **strncat** copies up to *n* characters from *string2* to the end of *string1*, stopping if it encounters a null character.

strlen returns the length of *string* in bytes, not including the null terminator. This may be useful in determining how much storage to allocate for a string.

index scans the given *string* for the first occurrence of character *c*. If it is found, **index** returns a pointer to it. If it is not found, **index** returns **NULL ((char *)0)**. **rindex** works similarly, but searches for the last occurrence of *c* in *string*.

NAME
swab – swap byte pairs

USAGE
swab(*src, dest, nb***)**
char **src*, **dest***;**
unsigned *nb***;**

DESCRIPTION
The ordering of bytes within a word is different on various machines. This may cause portability problems when moving binary data between machines.

swab interchanges each pair of bytes in array *src* of length *nb* and places the result into the array *dest*. The length *nb* should be an even number.

While **swab** is not a general solution to the portability problem, it may be useful for certain applications. In general COHERENT solves the portability problem by using canonical form, as described in **canon.h**.

SEE ALSO
canon.h
COHERENT Command Manual: **dd**

Maintenance

NAME
swap – enable swapping

USAGE
/etc/swap&

DESCRIPTION
The swapping code for COHERENT is a kernel process rather than being a fixed part of the operating system. This enables the system administrator to enable swapping only if and when desired.

The swapping code determines which processes or segments should be moved from memory to disk and which should be returned from disk to memory. Reasonable requests (either explicit or implicit) to COHERENT for more memory might fail when swapping is not enabled.

The swapper should be invoked from the shell **sh** as a background process. The file **/etc/rc** should invoke the swapper if swapping is always desired.

FILES
/etc/swap swapper load module

SEE ALSO
exec, init, lock

System Call

NAME
sync – flush system buffers

USAGE
sync()

DESCRIPTION
To improve system performance, the COHERENT system often changes a copy of part of a file system in a buffer in memory, rather than performing the time-consuming disk access required. **sync** writes all file system data which is in memory to the disk. This data includes cached write blocks, changed i-nodes, and super-blocks for each mounted file system. This writes the current time on each mounted file system, as the time is contained in the super-block.

Programs which need to examine a mounted file system issue a **sync** call to ensure that the file system is current and consistent. **sync** should also be issued immediately before rebooting the machine, to assure the integrity of mounted file systems.

SEE ALSO
update
COHERENT Command Manual: **sync**

NAME
sysgen – system generation

DESCRIPTION
This procedure generates a bootable file system on an installation's root device, typically a disk. The distribution medium is industry-standard half inch 9-track magnetic tape, at 800 or 1600 BPI. A source distribution tape contains two files, each terminated by a tape mark. The first file consists of 512-byte records, and the second is a dump of the source file system. On a binary distribution tape the second file is absent.

The first step is to boot the tape: read record 0 into memory location 0 and execute it. This produces the prompt '>'. A sample session of prompts and responses follows; responses are in bold face and must be typed in lower case.

```
>coherent
Coherent(222K, 2222)
# /etc/load /drv/rl
# /etc/mkfs /dev/rrl0 /gen/rlproto
# restor rf /gen/rldump /dev/rrl0
```

If there are any messages, something is probably wrong. Otherwise, the disk is ready for use. The following is a table of root devices currently supported:

RL01	/dev/rl	/dev/rrl0	/gen/rlproto	/gen/rldump
RL02	/dev/rl	/dev/rrl0	/gen/rlproto	/gen/rldump
RK05	/drv/rk	/dev/rrk0	/gen/rkproto	/gen/rkdump
RM02	/drv/rm	/dev/rrm00	/gen/rmproto	/gen/rmdump

To boot the disk, read sector 0 to memory location 0 and execute it. Respond to the '>' prompt with **coherent**. The '#' prompt should appear to indicate the single-user mode of COHERENT under the command interpreter (or shell) **sh**.

The following is an octal bootstrap for a PDP-11 using a TM11 controller at 800 BPI. This code must be keyed in above location 01000. For 1600 BPI distributions, change 060003 to 000003.

```
012700  172526              mov     $TMCMA, r0
005010                      clr     (r0)
012740  177000              mov     $-01000, -(r0)
012740  060003              mov     $060003, -(r0)
105710          1:          tstb    (r0)
100376                      bpl     1b
005007                      clr     pc
```

SEE ALSO
tape
COHERENT Command Manual: **load, mkfs, restor**
COHERENT Administrator's Guide

NOTES
The DEC bootstrap ROM cannot be used to boot the tape.

libc Library

NAME
system – pass a command line to the shell

USAGE
system(*command***)**
char ****command*;**

DESCRIPTION
system duplicates the action of the shell **sh** for a one line *command*. The *command* string is passed directly to the shell. The requesting process waits for the completion of the *command* and returns its exit status.

system may be used by commands such as **ed**, which can pass commands to the COHERENT shell in addition to processing normal interactive requests.

FILES
/bin/sh

SEE ALSO
exec, fork, popen, wait
COHERENT Command Manual: **sh**

DIAGNOSTICS
system returns the exit status of the child process, in the format described in **wait**: exit status in the high byte, signal information in the low byte. 0 normally means success, while nonzero normally means failure. This, however, depends on the *command*. If the shell is not executable, **system** returns a special code of 0177.

Device Driver

NAME

tape – magnetic tape devices

DESCRIPTION

This section gives a general explanation of COHERENT's use of industry-standard half inch 9-track magnetic tape. Exceptions or additional information may be found in sections of this manual describing particular devices.

A tape volume contains files, each consisting of one or more records and terminated by a tape mark. Two tape marks terminate the last file. Tape records may vary in length, but cannot exceed 2^16 bytes (2^15 is more practical).

Like other block-oriented devices, tape units may be accessed through the system's *cooked* interface or through the *raw* interface. On a cooked device, seeking to any byte offset and reading in any number of bytes is possible. It is not possible to read beyond the tape mark at the end of the current file. All records in the file must be 512 bytes in length, except the last. Write requests must be made in increments of 512 bytes, except the last. A cooked tape may be mounted like a disk, but only as a read-only file system.

A raw device bypasses the buffer cache, so I/O occurs directly to or from the user's buffer. One write request generates one tape record, and one read request returns exactly one record. The number of bytes read may be less than expected. If the tape mark is read, a count of 0 is returned, but the system positions the tape at the start of the next tape file. Seeking on a raw device is ignored, and mounting is not allowed.

A unit cannot be opened if it is off-line or already in use. If the write ring is absent, the unit cannot be opened for writing. Closing the device has varying effects, depending on the minor device opened and whether the device was opened for reading or writing. In the case of reading, the tape is rewound; if the no-rewind option was specified, the tape advances to the next file. In the case of writing, two tape marks are written at the current position and the tape is rewound; if the no-rewind option was specified, two tape marks are written and the tape is positioned between them. Note that when a device opened for writing is closed, the tape volume ends at the current position; data beyond this point is undefined.

Device Driver

The following device options exist, selected by prefixes to the device name:

h Read or write data at high density. The exact density depends on the drive model, but 1600 BPI (high) and 800 BPI (low) are typical.

n Do not rewind on close.

r The device is raw.

Hard errors may occur during tape operation. They include: detection of the end-of-tape (EOT) reflector, reading an unexpectedly long record, or seeking a cooked tape into a tape mark. After an error, no further operations may be performed on the unit until the program closes the device and the operator rewinds the tape. Soft parity errors may arise due to dirt, bad tape or misaligned heads. On writes, the driver attempts to place the record further along the tape. On reads, the driver simply rescans the record. After several failures, the driver announces a hard error.

Most utilities use generic device names, which are links to the actual device files appropriate for the site.

FILES

/dev/mt	generic cooked tape device
/dev/rmt	generic raw tape device
/dev/rnhtm0	raw no-rewind 1600 BPI TM11, unit 0

DIAGNOSTICS
Drivers may report errors to the console.

System Call

NAME
times – obtain process execution times

USAGE
#include <sys/times.h>
#include <sys/const.h>

times(*tbp***)**
struct tbuffer ****tbp*;**

DESCRIPTION
times fills in the structure addressed by its *tbp* argument with CPU time information about the current process and its children. The **tbuffer** structure may be obtained from the **sys/times.h** header file.

```
struct  tbuffer {
        long    tb_utime;       /* process user time */
        long    tb_stime;       /* process system time */
        long    tb_cutime;      /* childrens' user times */
        long    tb_cstime;      /* childrens' system times */
};
```

All of the times are measured in basic machine cycles, or **HZ**, which may be obtained from the **sys/const.h** header file. On a PDP-11 in North America, **HZ** is 60.

The childrens' times include the sum of the times of all terminated child processes of the current process and of all of their children. The *user* time represents execution time of user code, while *system* time represents system overhead such as executing system calls, processing signals, and other monitoring functions.

FILES
<sys/times.h>
<sys/const.h>

SEE ALSO
acct, ftime
COHERENT Command Manual: **time**

libc Library

NAME
isatty, ttyname, ttyslot – terminal identification

USAGE
isatty(*fd***)**
int *fd***;**

char *
ttyname(*fd***)**
int *fd***;**

ttyslot()

DESCRIPTION
Given a file descriptor *fd* attached to a terminal, **ttyname** returns the complete pathname of the special file (normally found in the directory **/dev**).

ttyslot returns the number of the line in the file **/etc/ttys** which describes the controlling terminal (see **ttys**).

isatty returns 1 if the file descriptor *fd* is attached to a terminal, and 0 otherwise.

FILES

/dev/*	terminal special files
/etc/ttys	login terminals

SEE ALSO
ioctl
COHERENT Command Manual: **tty**

DIAGNOSTICS
ttyname returns **NULL** if it cannot find a special file corresponding to *fd*.

A return value of 0 from **ttyslot** indicates an error.

NOTES
The string returned by **ttyname** is contained in a static area, and is overwritten by each subsequent call.

File Format

NAME
ttys – active terminal ports

DESCRIPTION
The file **/etc/ttys** describes the terminals in the COHERENT system and which should have a login process. The **init** process reads this file when bringing up the system in multi-user mode.

/etc/ttys contains one line for each terminal. If the first character of this line is '1', logins are enabled; '0' indicates an inactive port. The next character is 'l' if the port is a local line, and 'r' if it is a remote terminal. The third character is passed to **/etc/getty** to give speed and other information about the terminal port. The remainder of the line is the name of the special file for the terminal, normally found in the directory **/dev**.

FILES
/etc/ttys

SEE ALSO
getty, init
COHERENT Command Manual: **login**

System Call

NAME
umask – set file creation mask

USAGE
umask(*mask***)**
int *mask***;**

DESCRIPTION
umask allows a process to restrict the mode of files it creates. Commands that create files should specify the maximum reasonable mode. A parent (e.g. the shell **sh**) usually calls **umask** to restrict access to files created by subsequent commands.

The *mask* argument should be constructed from any of the permission bits found in **chmod** (the low-order 9 bits). When a file is created with **creat** or **mknod**, the bits specified by *mask* are zeroed in the *mode* argument; thus bits set in *mask* specify permissions which will be denied.

umask returns the old value of the file creation mask.

SEE ALSO
creat, mknod
COHERENT Command Manual: **sh, umask**

STDIO Library

NAME
ungetc – return character to input stream

USAGE
#include <stdio.h>

ungetc(*c, fp***)**
int *c*;
FILE **fp*;

DESCRIPTION
ungetc returns the character *c* to the stream *fp*. This character can then be read by a subsequent **getc, getw,** or **fread** call. Exactly one character at a time may be pushed back on any stream. A call to **fseek** will nullify the effects of **ungetc.**

FILES
<stdio.h>

SEE ALSO
fseek, getc, setbuf

DIAGNOSTICS
ungetc normally returns *c*; it returns **EOF** if the character cannot be pushed back.

System Call

NAME
unlink – remove a file

USAGE
unlink(*file***)**
char ****file*;**

DESCRIPTION
unlink removes the directory entry for the given *file*. If *file* is the last link, **unlink** frees the i-node and data blocks. Deallocation is delayed if the file is open. Other links to the file remain intact.

SEE ALSO
link
COHERENT Command Manual: **ln, rm, rmdir**

DIAGNOSTICS
unlink returns 0 on successful calls. It returns –1 if *file* does not exist, if the user does not have write and search permission in the directory containing *file*, or if *file* is a directory and the invoker is not superuser.

NAME
update – update file systems periodically

USAGE
/etc/update&

DESCRIPTION
update periodically performs **sync** to write all file system data which is in memory to the disk. It never exits.

The initialization command file **/etc/rc** normally executes **update**. It should not be executed directly.

SEE ALSO
init, sync
COHERENT Command Manual: **sync**

System Call

NAME
utime – change file access and modification times

USAGE
#include <sys/types.h>

utime(*file, times***)**
char **file***;**
time_t *times***[2];**

DESCRIPTION
utime sets the access and modification times associated with the given *file* to times obtained from *times***[0]** and *times***[1]**, respectively. The time of last change to the attributes is set to the time of the **utime** call.

This call must be made by the owner of *file* or by the superuser.

FILES
<sys/types.h>

SEE ALSO
stat
COHERENT Command Manual: **restor**

DIAGNOSTICS
utime returns −1 on errors, such as *file* nonexistent or invoker not the owner.

NAME
utmp.h – login accounting information

USAGE
#include <utmp.h>

DESCRIPTION
The file **/etc/utmp** contains a **utmp** entry for every user currently logged into the COHERENT system. The **utmp** structure is defined in the **utmp.h** header file.

```
#define DIRSIZ  14

struct utmp {
        char    ut_line[8];     /* terminal name */
        char    ut_name[DIRSIZ];/* user name */
        time_t  ut_time;        /* time of login */
};
```

If either the user name or terminal name is cleared, the entry is unused. The **ut_line** entry is the name of the special file for the user's terminal, normally in the directory **/dev**. The **ut_time** entry gives the date and time the user logged into COHERENT.

The file **/usr/adm/wtmp** maintains a record of all logins and logouts, and may be summarized by the **ac** command. **login** and **init** write entries into the **wtmp** file; neither creates the file, so login accounting is disabled unless **/usr/adm/wtmp** exists.

Entries in the **wtmp** file are identical to those in the **utmp** file. A null string in the **ut_name** field indicates a logout. Three special terminal names may be found in the **wtmp** file. When the system is booted, the **init** process writes a **ut_line** entry of '~'. When the time is changed with the **date** command, **date** writes an entry giving the old date ('|') and an entry giving the new date ('}'). This allows **ac** to adjust connect times appropriately.

FILES
<utmp.h>
/etc/utmp
/usr/adm/wtmp

SEE ALSO
init
COHERENT Command Manual: **ac, date, login, who**

System Call

NAME
wait – await completion of child process

USAGE
wait(*statp***)**
int ****statp*;**

DESCRIPTION
wait suspends execution of the invoking process until a child process (created with **fork**) terminates. If there are no outstanding child processes, **wait** returns an error indication.

The return value of a successful **wait** is the process id of the terminated child process. In addition, **wait** fills in the integer pointed to by the *statp* argument with exit status information about the completed process. If *statp* is **NULL**, **wait** discards the exit status information.

wait fills in the low byte of the status information word with the termination status of the child process. Termination may be because of a signal, because of an exit call, or because of stopped execution during **ptrace**. Termination with **exit**, which is normal completion, gives status 0. Other terminations give signal values as status, as defined in **signal**. The **0200** bit of the status code indicates that a core dump was produced. A status of **0177** indicates that the process is waiting for further **ptrace** actions.

The high byte of the returned status is the low byte of the argument to the **exit** system call.

If a parent process does not remain in existence long enough to **wait** on a child process, the child process is adopted by process 1 (the initialization process).

SEE ALSO
exit, fork, ptrace, signal
COHERENT Command Manual: **sh**

DIAGNOSTICS
wait returns the process id of the terminating child. If there are no children or if an interrupt occurs, it returns −1.

System Call

NAME
write – write to a file

USAGE
write(*fd, bufp, nb***)**
int *fd***;**
char **bufp***;**
int *nb***;**

DESCRIPTION
write writes *nb* bytes of data starting from address *bufp* to the file associated with file descriptor *fd*.

For block devices and regular files, data is written at the current write position, set either by the last **write** or by an **lseek** call. **write** advances the position by the number of characters written.

SEE ALSO
open, read

DIAGNOSTICS
write returns a value of -1 if an error occurred before the **write** operation commenced, such as a bad file descriptor *fd* or invalid *bufp* pointer. Otherwise, it returns the number of bytes actually written. It should be considered an error if this number is not the same as *nb*.

User Reaction Report

To keep this manual and COHERENT free of bugs and facilitate future improvements, we would appreciate receiving your reactions. Please fill in the appropriate sections below and mail to us. Thank you.

Mark Williams Company
1430 W. Wrightwood Avenue
Chicago, IL 60614

Name: ____________________

Company: ____________________

Address: ____________________

____

Phone: ____________________ Date: ____________________

Version and hardware used: ____________________

Did you find any errors in the manual? ____________________

____

____

Can you suggest any improvements to the manual? ____________________

____

____

Did you find any bugs in the software? ____________________

____

____

Can you suggest improvements or enhancements to the software?

____

____

____

Additional comments: (Please use other side.)