l — Command

List directory's contents in long format
l [*file ...***]**

l is a link to the command **ls -l**. It prints the contents of *file* in long format, that is, showing its length, its owner, the date and time it was last modified, and other useful information. If a *file* is a directory, **l** lists its contents. If no *file* is named, **l** lists the contents of the current directory by default.

See Also

commands, lc, lf, lr, ls, lx

l.out.h — Header File

Format for COHERENT 286 objects
#include <l.out.h>

The header file **l.out.h** describes the **l.out** object format, which is produced by the compiler, assembler, and the linker under COHERENT 286.

The assembler outputs an unlinked object module, which must be bound with any required libraries (leaving no unresolved symbols) to produce an executable file, or *load module*. A call to one of the **exec** routines can then execute the load module directly.

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

```
struct     ldheader {
           short l_magic;
           short l_flag;
           short l_machine;
           unsigned short l_entry;
           fsize_t l_ssize[NLSEG];
};
```

l_magic is the magic number that identifies a link 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 header file **mtype.h**. **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> — Define format of COHERENT 286 objects
<mtype.h> — Machine identifiers

See Also

as, cc, core, exec, ld, libc, mtype, nm

Notes

COHERENT 386 uses the common object file format (COFF) for its executables. See the Lexicon entry for **coff.h** for information on this format.

l3tol() — General Function (libc)

Convert file system block number to long integer
l3tol(*lp, l3p, n***)**
long **lp***;*
char **l3p***;*
unsigned *n***;**

To conserve space inside i-nodes in COHERENT file systems, the system stores block addresses in three bytes. Programs that reference or maintain file systems use the functions **l3tol()** and **ltol3()** routines to convert between the three-byte representation and **long** numbers.

l3tol() converts *n* three-byte block addresses at location *l3p* to an array of **long** integers at location *lp.*

See Also

libc

LASTERROR — Environmental Variable

Program that last generated an error
LASTERROR=*program name*

The environmental variable **LASTERROR** names the last program to have returned an error to the shell. For example, if you had used the command **set** with an incorrect number of arguments, it would have failed to run and would have exited with an error condition, and **LASTERROR** would read **LASTERROR=set**.

The command **help** reads **LASTERROR** to determine what the last program was for which you needed help. Thus, if you type **help** without an argument, it will return information about the program named in **LASTERROR**.

See Also

environmental variables

.lastlogin — System Administration

Record of last login
$HOME/.lastlogin

The command **login** records in file **$HOME/.lastlogin** the date and time you last logged in. **login** displays the information the next time you log in.

If this file does not exist **login** assumes that you are a new user, and by default executes the file **/etc/default/welcome**. This provides a "friendly" environment for users who are using COHERENT for the first time.

See Also

Administering COHERENT, login, Using COHERENT

Latin 1 — Definition

Standard 8859 of the International Standards Organization (ISO) defines a set of tables of eight-bit codes for the printable characters used in various languages.

The lower seven bits of each table (i.e., values 0x20 through 0x7E) are the same as those defined in ISO Standard 646; which, in turn, are the same as those in the character set called "U.S. ASCII", which encodes the Latin alphabet, plus numerals, punctuation marks, and additional characters commonly used in the United States. Values 0xA0 through 0xFF of each table in ISO 8859 either adds additional characters used in family of languages, or maps the characters in the Latin alphabet to the characters in another alphabet. For example, the table ISO 8859.2 (also called ISO Latin 2) adds to U.S. ASCII the characters used in the languages Albanian, Czech, Slovak, Hungarian, Polish, Romanian, and Croatian; whereas the table ISO 8859.6 maps the Latin alphabet to the Arabic alphabet.

Table ISO 8859.1 is also called *ISO Latin 1*. It adds the national characters used most Romance and Germanic languages (i.e., English, German, Dutch, Flemish, Norwegian, Icelandic, Swedish, Danish, Spanish, French, Italian, and Portuguese), plus Anglo-Saxon, Irish, and Finnish.

Please note that unlike U.S. ASCII, the characters in an ISO 8859 table are *not* in their proper order for lexical sorting. The character tables for locale-specific sorting are kept elsewhere.

As mentioned above, the printable characters 0x20 (space) through 0x7E (tilde) in ISO Latin 1 are the same as they are in U.S. ASCII. The following table gives the additional printable characters that ISO Latin 1 defines, from 0xA1 through 0xFF. The table gives each character's value in octal, decimal, and hexadecimal, and its description. Note that our printer does cannot print every character, so in some cases the description must suffice.

0241	161	0xA1	¡	Inverted exclamation mark
0242	162	0xA2	¢	Cent sign
0243	163	0xA3	£	Pound sign
0244	164	0xA4	¤	Currency sign
0245	165	0xA5	¥	Yen sign
0246	166	0xA6		Broken bar
0247	167	0xA7	§	Section symbol
0250	168	0xA8	¨	Diaeresis
0251	169	0xA9	©	Copyright sign
0252	170	0xAA	ª	Feminine ordinal indicator
0253	171	0xAB	«	Left angle quotation mark
0254	172	0xAC	–	Not sign
0255	173	0xAD		Soft hyphen
0256	174	0xAE	®	Registered trade mark sign
0257	175	0xAF	¯	Macron
0260	176	0xB0	°	Ring above or degree sign
0261	177	0xB1	±	Plus-minus sign
0262	178	0xB2		Superscript two
0263	179	0xB3		Superscript three
0264	180	0xB4		Acute accent
0265	181	0xB5	µ	Micro sign (mu)
0266	182	0xB6	¶	Pilcrow (paragraph) sign
0267	183	0xB7		Middle dot
0270	184	0xB8	¸	Cedilla
0271	185	0xB9		Superscript one
0272	186	0xBA	º	Masculine ordinal indicator
0273	187	0xBB	»	Right angle quotation mark
0274	188	0xBC		Vulgar fraction one quarter
0275	189	0xBD		Vulgar fraction one half
0276	190	0xBE		Vulgar fraction three quarters
0277	191	0xBF	¿	Inverted question mark
0300	192	0xC0		A with grave accent
0301	193	0xC1		A with acute accent
0302	194	0xC2		A with circumflex accent
0303	195	0xC3		A with tilde
0304	196	0xC4		A with diaeresis
0305	197	0xC5		A with ring above
0306	198	0xC6	Æ	Diphthong A with E
0307	199	0xC7	Ç	C with cedilla
0310	200	0xC8		E with grave accent
0311	201	0xC9		E with acute accent
0312	202	0xCA		E with circumflex accent
0313	203	0xCB		E with diaeresis
0310	204	0xCC		I with grave accent
0311	205	0xCD		I with acute accent
0312	206	0xCE		I with circumflex accent
0313	207	0xCF		I with diaeresis
0320	208	0xD0		Capital eth
0321	209	0xD1		N with tilde
0322	210	0xD2		O with grave accent
0323	211	0xD3		O with acute accent
0324	212	0xD4		O with circumflex accent
0325	213	0xD5		O with tilde
0326	214	0xD6		O with diaeresis
0327	215	0xD7		Multiplication sign
0330	216	0xD8	Ø	O with oblique stroke

0331	217	0xD9		U with grave accent
0332	218	0xDA		U with acute accent
0333	219	0xDB		U with circumflex accent
0334	220	0xDC		U with diaeresis
0335	221	0xDD		Y with acute accent
0336	222	0xDE		Thorn
0337	223	0xDF	ß	Sharp s
0340	224	0xE0	à	a with grave accent
0341	225	0xE1	á	a with acute accent
0342	226	0xE2	â	a with circumflex accent
0343	227	0xE3	ã	a with tilde
0344	228	0xE4	ä	a with diaeresis
0345	229	0xE5	å	a with ring above
0346	230	0xE6	æ	Diphthong a with e
0347	231	0xE7	ç	c with cedilla
0350	232	0xE8	è	e with grave accent
0351	233	0xE9	é	e with acute accent
0352	234	0xEA	ê	e with circumflex accent
0353	235	0xEB	ë	e with diaeresis
0354	236	0xEC	ì	i with grave accent
0355	237	0xED	í	i with acute accent
0356	238	0xEE	î	i with circumflex accent
0357	239	0xEF	ï	i with diaresis
0360	240	0xF0		Small eth
0361	241	0xF1	ñ	n with tilde
0362	242	0xF2	ò	o with grave accent
0363	243	0xF3	ó	o with acute accent
0364	244	0xF4	ô	o with circumflex accent
0365	245	0xF5	õ	o with tilde
0366	246	0xF6	ö	o with diaeresis
0367	247	0xF7		Division sign
0370	248	0xF8	ø	o with oblique stroke
0371	249	0xF9	ù	u with grave accent
0372	250	0xFA	ú	u with acute accent
0373	251	0xFB	û	u with circumflex accent
0374	252	0xFC	ü	u with diaresis
0375	253	0xFD	ý	y with acute accent
0376	254	0xFE		Small thorn
0377	255	0xFF	ÿ	y with diaeresis

See Also

ASCII, Programming COHERENT

lc — Command

List directory's contents in columnar format
lc [-1abcdfp] [*directory ...***]**

lc lists the names of the files in each *directory*, or the current directory if no *directory* is named. The files are categorized by type (files, directories, and so on) and listed in columns within each category.

The following options modify the output.

-1 List only one file name per line (do not print in columns). Please note that this is the numeral one, not a lower-case el.

-a List all file names, including '.' and '..'.

-b List block-special files only.

-c List character-special files only.

-d List directories only.

-f List regular files only.

-p List pipe files only.

See Also

commands, ls

Notes

lc -lf is useful for producing a list of regular files. For example

```
cp `lc -lf` mydir
```

copies all regular files to directory **mydir**.

lcasep — Command

Convert text to lower case
lcasep [-f *inputfile***] [-o** *outputfile***]**

The command **lcasep** converts characters in its input stream to lower case. If you do not name an *inputfile* on the command line, **lcasep** reads the standard input. If you do not name an *outputfile*, it writes it output to the standard output.

See Also

commands, mail

Notes

The command **smail** uses **lcasep** to convert headers on mail messages to convert addresses to lower case. Normally, users do not run it directly.

lcong48() — Random-Number Function (libc)

Initialize values from which 48-bit random numbers are computed
long lcong48(*param***)**
unsigned short *param***[7];**

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

Function **lcong48()** initializes the variables used to compute 48-bit pseudo-random numbers. *param* is an array of seven unsigned short integers that hold the initializers: *param***[0]** through *param***[2]** hold the "seed"; *param***[3]** through *param***[5]** hold the multiplier; and *param***[6]** holds the addend.

lcong48() returns nothing.

See Also

libc, srand48()

ld — Command

Link relocatable object modules
ld *[option …] file …*

A compiler translates a file of source code into a *relocatable object*. This relocatable object cannot be executed by itself, for calls to routines stored in libraries have not yet been resolved. **ld** combines, or *links*, relocatable object files with routines stored in libraries produced by the archiver **ar** to construct an executable file. For this reason, **ld** is sometimes called a *linker*, a *link editor*, or a *loader*.

ld scans its arguments in order and interprets each option as described below. Each non-option argument is either a relocatable object file, produced by **cc**, **as**, or **ld**, or a library archive produced by **ar**. It rejects all other arguments and prints a diagnostic message.

Each relocatable file argument is bound into the output file if its machine type matches the machine type of the first file so bound; if it does not, **ld** prints a diagnostic message. The symbol table of the file is merged into the output symbol table and the list of defined and undefined symbols updated appropriately. If the file redefines a symbol defined in an earlier bound module, the redefinition is reported and the link continues. The last such

redefinition determines the value that the symbol will have in the output file, which may be acceptable but is probably an error.

Each library archive argument is searched only to resolve undefined references, i.e., if there are no undefined symbols, the linker goes to the next argument immediately. The library is searched from first module to last and any module that resolves one or more undefined symbols is bound into the output exactly as an explicitly named relocatable file is bound. The library is searched repeatedly until an entire scan adds nothing to the executable file.

The order of modules in a library is important in two respects: it will affect the time required to search the library, and, if more than one module resolves an undefined symbol, it can alter the set of library modules bound into the output.

A library will link faster if the undefined symbols in any given library module are resolved by a library module that comes later in the library. Thus, the low-level library modules, those with no undefined symbols, should come at the end of the library, whereas the higher-level modules, those with many undefined symbols, should come at the beginning. The library module **ranlib.sym**, which is maintained by the **ar s** modifier, provides **ld** with a compressed index to the symbols defined in the library. But even with the index, the library will link much faster if the modules occur in top-down rather than bottom-up order.

A library can be constructed to provide a type of "conditional" linking if alternate resolutions of undefined symbols are archived in a carefully thought-out order. For instance, **libc.a** contains the modules

finit.o
exit.o
_finish.o

in precisely the order given, though some other modules may intervene. **finit.o** contains most of the internals of the STDIO library, **exit.o** contains the **exit()** function, and **_finish.o** contains an empty version of **_finish()**, the function that **exit()** calls to close STDIO streams before process termination. If a program uses any STDIO routines, macros, or data, then **finit.o** will be bound into the output with its version of **finish()**. If a program uses no STDIO, then the "dummy" **_finish.o** will be bound into the output because it is the first module that defines **_finish()** that the linker encounters after **exit.o** adds the undefined reference. This saves approximately 3,000 bytes. To set the order of routines within a library, use the archiver **ar**.

COFF Linking

COHERENT uses the Common Object File Format (COFF). This format renders many advantages, but it also places special demands upon the linker. The following discussing some of the complexities that arise for linking into the COFF format.

Under COFF, common variables are kept aligned according to their most strongly aligned contributor. If *name* is linked with another module that also declares *name* but sets it to another length, the linker creates one such variable and gives it the greater length of the two. **ld** deduces the alignment of a common variable by its length: if the length of a common is divisible by four, it is aligned on a four-byte boundary; if it is divisible by two, it is aligned on a two-byte boundary. Otherwise, it is assumed to be unaligned. The linker supports only three classes of alignment: four-byte, two-byte, and unaligned. It then aligns a common variable according to its most strongly aligned contributor.

For example, if one assembly-language module contributes a **.comm** (common) variable named **xyz** whose length is four bytes, and another contributes another **xyz** whose length is five bytes, **ld** gives the resulting **xyz** a length of eight bytes to satisfy the length requirement (at least five) and the alignment requirement (four-byte boundary).

Or in another example, if you declare a C variable **char x;** x is a common variable, with a length of one byte. If another C module declares **long x;** the two **x**'s will share a length of four bytes. However, in the first module **sizeof(x) == 1** and in the second **sizeof(x) == 4**. These will cause warning messages to appear, which you can turn off by using the **-q** option.

After **ld** has made its first pass, it places all common variables at the end of the **.bss** segment: first the four-byte-aligned variables, then the two-byte-aligned, then the unaligned.

Options

ld recognizes the following options:

-e *entry*
Specify the *entry* point of the output module, either as a symbol or as an absolute octal address.

LEXICON

-f (Force) Force link even if there are errors. Results may be undefined.

-G Suppress the common/global warning — that is, tell **ld** not to complain if a global variable is also used as a common variable.

-i This option is obsolete, but is kept for compatibility purposes. If you include it in a **makefile**, **ld** will silently ignore it.

-K Link a kernel segment.

-L*directory*
Search *directory* for libraries and objects before searching the directories named in **LIBPATH**. Note that you can have more than one **-L** option in a **ld** command line. For example, if **LIBPATH** is set to **/lib:/usr/lib**, then the command line

```
ld -L/search/First -L/search/Next a.o -lxyz
```

tells **ld** to search for libraries **libxyz.a** and **libc.a** along the path:

```
/search/First:/search/Next:/lib:/usr/lib
```

The character that separates entries in the path is set by the macro **LISTSEP**. Header file **path.h** defines this to be the ':'.

-l *name* An abbreviation for the library **/lib/lib***name***.a** or **/usr/lib/lib***name***.a** if the first is not found.

-o *file* Write output to *file*. The default is **a.out**.

-q Suppress all warning messages.

-Q Suppress all error messages, not just warnings.

-r Retain relocation information in the output, and issue no diagnostic message for undefined symbols. This option builds a **.o** file that appears as if its pieces had been compiled together.

-s Strip the symbol table from the output. The same effect may be obtained by using the command **strip**. The **-s** and **-r** options are mutually exclusive.

-u *symbol*
Add *symbol* to the symbol table as a global reference, usually to force the linking of a particular library module.

-X Discard local compiler-generated symbols beginning **.L**.

-x Discard all local symbols.

ld reads the environmental variables **LDHEAD** and **LDTAIL** and appends them to, respectively, the beginning and end of its command line. For example, to ensure that **ld** is always executed with the option **-d**, insert the following into your **.profile**:

```
export LDHEAD=-d
```

Likewise, to ensure that **ld** always includes the mathematics library **libm** when it links, insert the following into your **.profile**:

```
export LDTAIL=-lm
```

LIBPATH

Except when used with its **-l** option, **ld** does not know about paths: it links exactly what you tell it to link via the **cc** command line. **cc** looks for libraries by searching the directories named in the environmental variable **LIBPATH**. If you do not define **LIBPATH** in your environment, it searches the default **LIBPATH** as defined in **/usr/include/path.h**. If you define **LIBPATH**, **cc** searches the directories in the order you specify. For example, a typical definition is:

```
export LIBPATH=:/lib:/usr/lib
```

This searches the current directory '.', then **/lib**, then **/usr/lib**.

Linker-defined Symbols

ld defines the following set of symbols within an executable program:

__end_text End of the **.text** segment
__end_data End of the **.data** segment
__end_bss End of the **.bss** segment
__end End of the highest segment

Note that if you have a segment named **.xyz**, then **ld** will allow you to use **__end_xyz**.

Files

a.out — Default output
/coherent for **-k** option
/lib/lib*.a — Libraries
/usr/lib/lib*.a — More libraries

See Also

ar, ar.h, as, cc, cdmp, coff.h, commands, l.out.h, LIBPATH, strip

Diagnostics

The following gives the error messages returned by **ld**. The messages are in alphabetical order; each is marked as to whether it is a *fatal* or *warning* condition. A fatal message usually indicates a condition that caused the compiler to terminate execution. Warning messages point out code that is compilable, but may produce trouble when the program is executed.

archive '*string*' is corrupt (fatal)
This archive makes no sense. You may wish to examine this with the archiver **ar**.

file *string*: module *string*: bad header (warning)
string does not look like a real object module.

can't find '*string*' (fatal)
ld cannot find the requested library. Make sure that the **cc** command line points to the directory that holds the archive.

cannot create '*string*' (fatal)
ld cannot create its output file.

entry point '*string*' not in .text (warning)
error reading '*string*' (fatal)

'*string*' is not a COFF archive (fatal)
All files ending **.a** should be COFF archives. You may need to rebuild this archive.

Library must be created with ar -s option (fatal)
The option **-s** to **ar** gives libraries a symbol table for the use of **ld**.

No work (fatal)
There were no object files loaded.

pass 1, *n* errors (fatal)
At the end of pass 1 there were *n* errors detected. The link stopped here.

symbol '*string*' redefined in file '*string*': module '*string*' (warning)
A symbol is defined in incompatible ways in different files.

symbol '*string*' redefined in file '*string*' (warning)
A symbol is defined in incompatible ways in different files.

file *string*: module *string*: relocation out of range *0xn* (warning)
A relocation record points outside the range of its segment.

symbol '*string*' severe warning symbol defined as a common and a global (warning)
A symbol was defined as a common, e.g.

```
int x;
```

and as a global, e.g.:

```
int x = 5;
```

There is no good way to fix this without reading the code and thinking about the variable usage. The linker turned the global into an external. That is, it turned

LEXICON

```
int x;
```
into
```
extern int x;
```
This matches the UNIX linker.

file *string*: module *string*: unknown r_type *n* in segment *n* record *n* (warning)
Unknown type on COFF relocation record.

unlikely input file name '*string*' (warning)
Input file names must end **.o** for object or **.a** for archive.

symbol '*string*' warning defined with lengths *n* and *n* (warning)
A common was defined with different lengths, while this is legal it is very unusual in C programs. This warning may be turned off with the flag **-c**.

symbol '*string*' warning, redefines builtin symbol (warning)
Some symbols such as **__end** and **__end_text** are special to the linker. In general, symbols beginning '__' are reserved to implementors and should be avoided by users. Your definition has been used.

write error (fatal)
ld cannot write the executable program. Check that you have permission to write into the target directory.

Notes

If you are linking a program by hand (that is, running **ld** independently from the **cc** command), be sure to include the appropriate run-time start-up routine with the **ld** command line; otherwise, the program will not link correctly.

ldexp() — General Function (libc)

Combine fraction and exponent
#include <math.h>
double ldexp(*f***, *e*)**
double *f*; **int** *e*;

ldexp() combines the fraction *f* with the binary exponent *e* to return a floating-point value *real* that satisfies the equation *real*=*m*×2^*e*.

See Also

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

LDHEAD — Environmental Variable

Append options to beginning of ld command line
export LDHEAD=*options*

The COHERENT linker **ld** reads the environmental variables **LDHEAD** and **LDTAIL** before it begins its work. You can set these variables to hold the default options that you want the linker always to use.

ld appends the options in **LDHEAD** to the beginning of its command line.

See Also

environmental variables, ld, LDTAIL

Notes

This environmental variable is included only to support existing code. Its use is deprecated, and it may not be supported in future releases of COHERENT.

ldiv() — General Function (libc)

Perform long integer division
#include <stdlib.h>
ldiv_t ldiv(*numerator***, *denominator*)**
long *numerator*, *denominator*;

ldiv() divides *numerator* by *denominator*. It returns a structure of the type **ldiv_t**, which is structured as follows:

```
typedef struct {
      long quot;
      long rem;
} ldiv_t;
```

ldiv() writes the quotient into **quot** and the remainder into **rem**.

The sign of the quotient is positive if the signs of the arguments are the same; it is negative if the signs of the arguments differ. The sign of the remainder is the same as the sign of the numerator.

If the remainder is non-zero, the magnitude of the quotient is the largest integer less than the magnitude of the algebraic quotient. This is not guaranteed by the operators **/** and **%**, which merely do what the machine implements for divide.

See Also

libc
ANSI Standard, §7.10.6.4

Notes

The ANSI Standard includes this function to permit a useful feature found in most versions of FORTRAN, where the sign of the remainder will be the same as the sign of the numerator. Also, on most machines, division produces a remainder. This allows a quotient and remainder to be returned from one machine-divide operation.

If the result of division cannot be represented (e.g., because *denominator* is set to zero), the behavior of **ldiv()** is undefined. *Caveat utilitor.*

LDTAIL — Environmental Variable

Append options to end of ld command line
export LDTAIL=*options*

The COHERENT linker **ld** reads the environmental variables **LDHEAD** and **LDTAIL** before it begins its work. You can set these variables to hold the default options that you want the linker always to use.

ld appends the options in **LDTAIL** to the end of its command line.

See Also

environmental variables, ld, LDHEAD

Notes

This environmental variable is included only to support existing code. Its use is deprecated, and it may not be supported in future releases of COHERENT.

let — Command

Evaluate an expression
let [*expression***]**

The command **let** is built into the Korn shell **ksh**. It evalutes *expression*; it returns zero if *expression* evaluates to non-zero status, and non-zero if it evaluates to zero status.

See Also

commands, ksh

lex — Command

Lexical analyzer generator
lex [-t][-v][*file***]**
cc lex.yy.c -ll

Many programs, e.g., compilers, process highly structured input according to rules. Two of the most complicated parts of such programs are *lexical analysis* and *parsing* (also called *syntax analysis*). The COHERENT system includes two powerful tools called **lex** and **yacc** to help you construct these parts of a program. **lex** converts a set of lexical rules into a lexical analyzer, and **yacc** converts a set of parsing rules into a parser.

The output of **lex** may be used directly, or may be used by a parser generated by **yacc**.

LEXICON

lex reads a specification from the given *file* (or from the standard input if none), and generates a C function called **yylex()**. **lex** writes the generated function in the file **lex.yy.c**, or on standard output if you use the **-t** option. The **-v** option prints some statistics about the generated tables.

The tutorial on **lex** that appear in this manual describes **lex** in detail. In brief, the generated function **yylex()** matches portions of its input to one pattern (sometimes called a regular expression) from a set of rules, or *context*, and executes associated C commands. Unmatched portions of the input are copied to the output stream. **yylex()** returns EOF when input has been exhausted.

lex uses the following macros that you may replace with the preprocessor directive **#undef** if you wish: **input()** (read the standard input stream), and **output(*c*)** (write the character *c* to the standard output stream). You may also replace the following functions if you wish: **main()** (main function), **error(...)** (print error messages; takes same arguments as **printf**), and **yywrap()** (handle events at the end of a file). If an action is desired on end of file, such as arranging for more input, **yywrap()** should perform it, returning zero to keep going.

A full **lex** specification has the following format:

- Macro definitions, of the form:

```
name  pattern
```

- Start condition declarations:

```
%S    NAME ...
```

- Context declarations:

```
%C    NAME ...
```

- Code to be included in the header section:

```
%{
anything
%}
<tab or space> anything
```

- Rules section delimiter (must always be present):

```
%%
```

- Code to appear at the start of **yylex()**:

```
<tab or space> anything
```

- Rules for initial context, in any of the forms:

```
rule        action;
rule        | (means use next action)
rule        {
<tab or space>    action;
<tab or space>    }
```

- For each additional context:

```
%C    NAME
...rules for this context...
```

- End of rules section delimiter:

```
%%
```

- Code to be copied verbatim, such as user provided **input(), output(), yywrap(),** or other.

lex matches the longest string possible; if two rules match the same length string, the rule specified first takes precedence. **lex** puts the matched string, or *token,* in the **char** array **yytext[]**, and sets the variable **yyleng** to its length.

Actions may use the following:

ECHO . . . . . Output the token
REJECT . . . . . Perform action for lower precedence match
BEGIN *NAME* . . . . . Set start condition to *NAME*

BEGIN 0 . . . . . Clear start condition
yyswitch(*NAME***)** . . . . Switch to context *NAME*, return current
yyswitch(0) . . . . . Switch to initial context
yynext() . . . . . Steal next character from input
yyback(*c***)**. . . . . Put character *c* back into input
yyless(*n***)** . . . . . Reduce token length to *n*, put rest back
yymore() . . . . . Append next token to this one
yylook() . . . . . Returns number of chars in input buffer

lex rules are contiguous strings of the form

[<*NAME,...*>][^] *token* [*/lookahead*][$]

where brackets '[]' indicate optional items.

<*NAME,...*> . . . . . Match only under given start conditions
^. . . . . Match the beginning of a line
$. . . . . Match the end of a line
token . . . . . Pattern that a given token is to match
/lookahead. . . . . Pattern that given trailing text is to match

Pattern elements:

a . . . . . The character **a**
\a . . . . . The character **a**, even if special
. . . . . . Any character except newline
[abx-z] . . . . . Any of **a, b,** or **x** through **z**
[^abx-z]. . . . . Any except **a, b,** or **x** through **z**
abc . . . . . The string **abc**, even if any are special
{*name*} . . . . . The macro definition *name*
(*exp***)**. . . . . The pattern *exp* (grouping operator)

Optional operators on elements:

*e***?** . . . . . Zero or one occurrence of *e*
*e** . . . . . Zero or more consecutive *e*s
*e***+** . . . . . One or more consecutive *e*s
e{*n*} . . . . . *n* (a decimal number) consecutive *e*s
e{*m,n*}. . . . . *m* through *n* consecutive *e*s

Patterns may be of the form:

e1e2. . . . . Matches the sequence *e1 e2*
e1|*e2* . . . . . Matches either *e1* or *e2*

lex recognizes the standard C escapes: **\n**, **\t**, **\r**, **\b**, **\f**, and *****ooo* (octal representation). The special characters

```
\ ( ) < > { } % * + ? [ - ] ^ / $ . |
```

must be prefixed with \ or enclosed within quotation marks (excepting " and \) to be normal. Within classes, only the characters . ^ - \ and] are special.

Files

/usr/lib/libl.a
/usr/src/libl/* — library source code

See Also

commands, yacc
Introduction to lex, the Lexical Analyzer

Lexicon — Technical Information

Format of the COHERENT manual pages

The COHERENT manual pages use a unique format, which we call "the Lexicon." Under this format, each function, each command, and each term has its own entry in the manual. All manual entries are printed together, instead of being divided into segments. The purpose of this format is to make it easy for you to find the manual entry for any given command or function. Anyone who as struggled with the multiple volumes of UNIX documentation can

appreciate this feature of the COHERENT Lexicon.

For more details on the Lexicon format, see the introduction to the Lexicon.

See Also

Administering COHERENT, Programming COHERENT, Using COHERENT

lf — Command

List directory's contents in columnar format
lf [*file ...***]**

lf is a link to the command **ls -CF**. It prints *file* in columnar format, like the command **lc**. **lf**, however, combines files and directories into one listing, with directories being indicated by a slash after the file name and executable being indicated by an asterisk. If a *file* is a directory, **l** lists its contents. If no *file* is named, **lf** lists the contents of the current directory by default.

See Also

commands, l, lc, lr, ls, lx

libc — Library

Standard C library
/lib/libc.a

libc is the library that contains most functions linked into C programs. It contains many general-purpose functions, as well as stubs for COHERENT system calls. The following summarizes these functions.

Binary Data

The following functions manipulate binary data types, that is, integers and floating-point numbers.

abs() . . . . . Return the absolute value of an integer
decvax_d() . . . . . Convert a **double** from IEEE to DECVAX format
decvax_f() . . . . . Convert a **float** from IEEE to DECVAX format
div() . . . . . Perform integer division
frexp() . . . . . Separate fraction and exponent
ieee_d() . . . . . Convert a **double** from DECVAX to IEEE format
ieee_f() . . . . . Convert a **float** from DECVAX to IEEE format
ldexp() . . . . . Combine fraction and exponent
ldiv() . . . . . Perform long integer division
modf() . . . . . Separate integral part and fraction

Binary Data and Strings

The following functions convert binary data forms to strings, or strings to binary forms.

atof() . . . . . Convert ASCII strings to floating point
atoi() . . . . . Convert ASCII strings to integers
atol() . . . . . Convert ASCII strings to long integers
ecvt() . . . . . Convert floating-point numbers to strings
fcvt() . . . . . Convert floating-point numbers to strings
gcvt() . . . . . Convert floating-point numbers to strings
strtod() . . . . . Convert string to floating-point number
strtol() . . . . . Convert string to long integer
strtoul() . . . . . Convert string to unsigned long integer

ctype Functions

The **ctype** functions test a character's *type*. Some can transform some characters into others. "ctype" is an abbreviation for "character type"; all are declared or defined in the header file **<ctype.h>**. They are as follows:

_tolower() . . . . . Convert an upper-case character to lower case
_toupper() . . . . . Convert a lower-case character to upper case
isalnum() . . . . . Test if alphanumeric character
isalpha() . . . . . Test if alphabetic character
isascii() . . . . . Test if ASCII character
iscntrl() . . . . . Test if a control character
isdigit() . . . . . Test if a numeric digit

isgraph() . . . . . Test if a graphics character
islower() . . . . . Test if lower-case character
isprint() . . . . . Test if printable character
ispunct() . . . . . Test if punctuation mark
isspace() . . . . . Test if a tab, space, or return
isupper() . . . . . Test if upper-case character
isxdigit() . . . . . Test if hexadecimal numeral
toascii() . . . . . Convert a character to ASCII
tolower() . . . . . Convert an upper-case character to lower case
toupper() . . . . . Convert a lower-case character to upper case

Files and Directories

The following functions are used to manipulate files and directories, and their names.

_getwd() . . . . . Get current working directory name
closedir() . . . . . Close a directory stream
dup2() . . . . . Duplicate a file descriptor
getcwd() . . . . . Get current working directory
mktemp() . . . . . Generate a temporary file name
opendir() . . . . . Open a directory stream
path() . . . . . Build a path name for a file
readdir() . . . . . Read a directory stream
remove() . . . . . Remove a file
rewinddir() . . . . . Rewind a directory stream
seekdir() . . . . . Reset the position within a directory stream
telldir() . . . . . Return position within a directory stream

Interprocess Communication

The following functions perform interprocess communcation.

ftok() . . . . . Generate keys for interprocess communication
msgctl() . . . . . Control message operation
msgget() . . . . . Get a message queue
msgrcv() . . . . . Receive a message
msgsnd() . . . . . Send a message
semctl() . . . . . Control semaphore operations
semget() . . . . . Get a set of semaphores
semop() . . . . . Perform semaphore operations
shmat() . . . . . Attach a shared-memory segment to a process
shmctl() . . . . . Manipulate shared memory
shmdt() . . . . . Detach a shared-memory segment from a process
shmget() . . . . . Get the shared-memory segment

Memory Management

The following functions help to manage memory.

alloca() . . . . . Dynamically allocate space on the stack
calloc() . . . . . Allocate dynamic memory
free() . . . . . Return dynamic memory to free memory pool
malloc() . . . . . Allocate dynamic memory
realloc() . . . . . Reallocate dynamic memory
sbrk() . . . . . Increase a program's data space

Passwords and Groups

The following functions manipulate the system files **/etc/group**, **/etc/password**, and **/etc/shadow**, and uses the information found therein.

endgrent() . . . . . Close group file
endpwent() . . . . . Close password file
endspent() . . . . . Close the shadow-password file
getgrent() . . . . . Get group file information
getgrgid() . . . . . Get group file information, by group id
getgrnam() . . . . . Get group file information, by group name

LEXICON

getlogin() . . . . . Get login name
getpass() . . . . . Get password with prompting
getpw() . . . . . Search password file
getpwent() . . . . . Get password file information
getpwnam() . . . . . Get password file information, by name
getpwuid() . . . . . Get password file information, by identifier
getspent() . . . . . Get a shadow-password record
getspnam() . . . . . Get a shadow-password record, by user name
initgroups() . . . . . Initialize the supplementary group-access list
setgrent() . . . . . Rewind group file
setpwent() . . . . . Rewind password file
setspent() . . . . . Rewind the shadow-password file

Processes

The following functions execute and terminate. For information on how the **exec()** functions differ, see the Lexicon entry **execution**.

_exit() . . . . . Terminate a process
abort() . . . . . End program immediately
atexit() . . . . . Register a function to be called when the program exits
ctermid() . . . . . Name the terminal device that controls the current process
execl() . . . . . Execute a load module
execle() . . . . . Execute a load module
execlp() . . . . . Execute a load module
execlpe() . . . . . Execute a load module
execv() . . . . . Execute a load module
execvp() . . . . . Execute a load module
execvpe() . . . . . Execute a load module
raise() . . . . . Let a process send a signal to itself
sleep() . . . . . Suspend execution

Random Number

libc contains the following functions for generating pseudo-random numbers:

drand48() . . . . . Return 48-bit pseudo-random number as double
erand48() . . . . . Return 48-bit pseudo-random number as double
jrand48() . . . . . Return 48-bit pseudo-random number as long integer
lcong48() . . . . . Initialize values from which 48-bit random numbers are computed
lrand48() . . . . . Return 48-bit pseudo-random number as non-negative long integer
mrand48() . . . . . Return 48-bit pseudo-random number as long integer
nrand48() . . . . . Return 48-bit pseudo-random number as non-negative long integer
rand() . . . . . Generate pseudo-random numbers
seed48() . . . . . Initialize values from which 48-bit random numbers are computed
srand() . . . . . Seed random number generator
srand48() . . . . . Seed 48-bit pseudo-random number routines

Regular Expressions

The following functions read and interpret UNIX-style regular expressions:

regcomp() . . . . . Compile a regular expression into a structure
regerror() . . . . . Return an error message from a regular-expression function
regexec() . . . . . Compare a string with a regular expression
regsub() . . . . . Use regular expression to build a string

STDIO

STDIO is an abbreviation for *standard input and output*. It refers to a set of standard library functions that accompany all C compilers and that govern input and output with peripheral devices. COHERENT includes the following STDIO routines:

clearerr() . . . . . Present status stream
fclose() . . . . . Close a file stream
fdopen() . . . . . Open a file stream for I/O
feof() . . . . . Discover a file stream's status

ferror() . . . . . Discover a file stream's status
fflush() . . . . . Flush an output buffer
fgetc(). . . . . Get a character
fgetpos() . . . . . Read the file-position indicator
fgets(). . . . . Get a string
fgetw() . . . . . Get a word
fileno() . . . . . Get a file descriptor from a **FILE** structure
fopen() . . . . . Open a file stream
fprintf(). . . . . Format and print to a file stream
fputc() . . . . . Output a character
fputs() . . . . . Output a string
fputw() . . . . . Output a word
fread() . . . . . Read a file stream
freopen() . . . . . Open a file stream
fscanf() . . . . . Format and read from a file stream
fseek() . . . . . Seek in a file stream
fsetpos() . . . . . Set the file-position indicator
ftell() . . . . . Return file pointer position
fwrite() . . . . . Write to a file stream
getc() . . . . . Get a character
getchar() . . . . . Get a character
gets() . . . . . Get a string
getw(). . . . . Get a word
pclose(). . . . . Close a pipe
popen() . . . . . Open a pipe
printf() . . . . . Print a formatted string
putc() . . . . . Output a character
putchar(). . . . . Output a character
puts() . . . . . Output a string
putw(). . . . . Output a word
rewind() . . . . . Reset a file pointer
scanf() . . . . . Format and input from standard input
setbuf() . . . . . Set alternative file-stream buffer
setvbuf() . . . . . Set alternative file-stream buffer
sprintf() . . . . . Format and print to a string
sscanf(). . . . . Format and read from a string
tmpfile() . . . . . Create a temporary file
tmpnam() . . . . . Generate a unique name for a temporary file
ungetc() . . . . . Return character to file stream
vfprintf() . . . . . Format and print to a file stream
vprintf() . . . . . Print a formatted string
vsprintf(). . . . . Format and print to a string

String Functions

The character string is a common formation in C programs. The runtime representation of a string is an array of ASCII characters that is terminated by a null character ('\0'). COHERENT uses this representation when a program contains a string constant; for example:

```
"I am a string constant"
```

The address of the first character in the string is used as the starting point of the string. A pointer to a string holds only this address. Note, too, that an array of 20 characters can hold a string of 19 (*not* 20) non-null characters; the 20th character is the null character that terminates the string.

The following routines are available to help manipulate strings. The prototypes for most are declared in the header file **string.h**:

bcmp() . . . . . Berkeley function to compare two chunks of memory
bcopy() . . . . . Berkeley function to copy memory
bzero() . . . . . Berkeley function to initialize memory to NUL
fnmatch() . . . . . Match a string with a normal expression
index() . . . . . Search string for a character; use **strchr()** instead
memccpy(). . . . . Copy a region of memory up to a set character

LEXICON

memchr() . . . . . Search a region of memory for a character
memcmp() . . . . . Compare two regions of memory
memcpy() . . . . . Copy one region of memory into another
memmove() . . . . . Copy one region of memory into another with which it overlaps
memset() . . . . . Fill a region of memory with a character
pnmatch() . . . . . Match string pattern
rindex() . . . . . Find rightmost occurrence of a character in a string
strcat() . . . . . Concatenate two strings
strcmp() . . . . . Compare two strings
strncat() . . . . . Append one string onto another
strncmp() . . . . . Compare two lengths for a set number of bytes
strcpy() . . . . . Copy a string
strncpy() . . . . . Copy a portion of a string
strcoll() . . . . . Compare two strings, using locale information
strcspn() . . . . . Return length one string excludes characters in another
strdup() . . . . . Duplicate a string
strerror() . . . . . Translate an error number into a string
strlen() . . . . . Measure a string
strpbrk() . . . . . Find first occurrence in string of character from another string
strchr() . . . . . Find leftmost occurrence of character in a string
strrchr() . . . . . Find rightmost occurrence of character in a string
strspn() . . . . . Return length one string includes character in another
strstr() . . . . . Find one string within another string
strtok() . . . . . Break a string into tokens
strxfrm() . . . . . Transform a string, using locale information

System Logs

The following functions manipulate the files **/etc/utmp** and **/usr/adm/wtmp**, which record login events on your system. The former file records every login that is still executing (i.e., the user has logged in and has not yet logged), and every past login.

endutent() . . . . . Close the logging file.
getutent() . . . . . Read the next entry from **/etc/utmp**.
getutid() . . . . . Find an entry in **/etc/utmp** by login identifier.
getutline() . . . . . Find an entry in **/etc/utmp** by login device.
pututline() . . . . . Write a record into **/etc/utmp**.
setutent() . . . . . Rewind the input stream that is reading **/etc/utmp**
utmpname() . . . . . Manipulate a file other than **/etc/utmp**.

Terminals

The following functions help you cope with terminals.

isatty() . . . . . Check if a device is a terminal
ttyname() . . . . . Identify a terminal
ttyslot() . . . . . Return a terminal's line number

Standard Time Functions

libc includes the following functions to manipulate time:

asctime() . . . . . Convert time structure to ASCII string
clock() . . . . . Get processor time
ctime() . . . . . Convert system time to an ASCII string
difftime() . . . . . Return difference between two times
gmtime() . . . . . Convert system time to calendar structure
localtime() . . . . . Convert system time to calendar structure
mktime() . . . . . Turn broken-down time into calendar time
strftime() . . . . . Format locale-specific time
tzset() . . . . . Set local time zone

System Calls

The COHERENT kernel makes many services available to the C programmer. A programmer can use a COHERENT service through a system call. **libc** includes intefaces to the following system calls:

access() . . . . . Check if file can be accessed in given mode
acct() . . . . . Enable/disable process accounting
alarm() . . . . . Set an alarm
brk() . . . . . Change size of data area
chdir() . . . . . Change working directory
chmod() . . . . . Change file protection modes
chown() . . . . . Change ownership of a file
chroot() . . . . . Change process's root directory
chsize() . . . . . Change the size of a file
close() . . . . . Close a file
creat() . . . . . Create/truncate a file
dup() . . . . . Duplicate a file descriptor
execve() . . . . . Execute a load module
exit() . . . . . Terminate a program gracefully
fcntl() . . . . . Manipulate an open file
fork() . . . . . Create a new process
fpathconf() . . . . . Get a file variable by file descriptor
fstat() . . . . . Get information about a file system
fstatfs() . . . . . Get information about a file system
ftime() . . . . . Get current system time
getdents() . . . . . Read directory entries
getegid() . . . . . Get effective group id
geteuid() . . . . . Get effective user id
getgid() . . . . . Get real group id
getgroups() . . . . . Read the supplemental group-access list
getmsg() . . . . . Get the next message from a stream
getpgrp() . . . . . Get process-group identifier
getpid() . . . . . Get process id
getppid() . . . . . Get process id of parent process
getuid() . . . . . Get real user id
gtty() . . . . . Get terminal modes
ioctl() . . . . . Device-dependent control
kill() . . . . . Send a signal to a process
link() . . . . . Create a link
lseek() . . . . . Set read/write position
mkdir() . . . . . Create a directory
mkfifo() . . . . . Create a FIFO
mknod() . . . . . Create a special file
mount() . . . . . Mount a file system
nap() . . . . . Sleep briefly
open() . . . . . Open a file
pathconf() . . . . . Get a file variable by path name
pause() . . . . . Wait for signal
pipe() . . . . . Create a pipe
poll() . . . . . Query several I/O devices
ptrace() . . . . . Trace process execution
putmsg() . . . . . Place a message onto a stream
read() . . . . . Read from a file
rename() . . . . . Rename a file
rmdir() . . . . . Remove a directory
setgid() . . . . . Set group id and user id
setgroups() . . . . . Set the supplemental group-access list
setpgid() . . . . . Set the process-group identifier
setpgrp() . . . . . Make a process a process-group leader
setsid() . . . . . Set session identifier
setuid() . . . . . Set user id
sigaction() . . . . . Perform detailed signal management
sigaddset() . . . . . Add a signal to a set of signals
sigdelset() . . . . . Delete a signal from a set
sigemptyset() . . . . . Initialize a set of signals
sigfillset() . . . . . Initialize a set of signals
sighold() . . . . . Place a signal on hold

sigignore() . . . . . Tell the system to ignore a signal
sigismember(). . . . . Check if a signal is a member of a set
signal() . . . . . Specify action to take upon receipt of a given signal
sigpause() . . . . . Pause until a given signal is received
sigpending(). . . . . Examine signals that are blocked and pending
sigprocmask(). . . . . Examine or change the signal mask
sigrelse() . . . . . Release a signal for processing
sigset() . . . . . Specify action to take upon receipt of a given signal
sigsuspend(). . . . . Install a signal mask and suspend process
stat() . . . . . Find file attributes
statfs() . . . . . Get information about a file system
stime() . . . . . Set the time
stty() . . . . . Set terminal modes
sync(). . . . . Flush system buffers
sysconf() . . . . . Get configurable system variables
sysi86(). . . . . Identify parts within Intel-based machines
time(). . . . . Get current system time
times() . . . . . Obtain process execution times
ulimit(). . . . . Get/set limits for a process
umask(). . . . . Set file creation mask
umount(). . . . . Unmount a file system
uname() . . . . . Get name and version of COHERENT
unlink(). . . . . Remove a file
ustat() . . . . . Get statistics on a file system
utime() . . . . . Change file access and modification times
wait() . . . . . Await completion of child process
waitpid() . . . . . Wait for a process to terminate
write() . . . . . Write to a file

Miscellaneous

The following functions do not fit neatly into any of the above categories.

bsearch(). . . . . Search an array
coffnlist() . . . . . Symbol table lookup
crypt() . . . . . Encryption using rotor algorithm
getenv() . . . . . Read environmental variable
getopt(). . . . . Get a command-line option
l3tol(). . . . . Convert file system block number to long integer
lockf(). . . . . Lock a file or a section of a file
longjmp(). . . . . Perform a non-local goto
ltol3(). . . . . Convert long integer to file system block number
mtype(). . . . . Return symbolic machine type
perror(). . . . . System call error messages
putenv() . . . . . Add a string to the environment
qsort() . . . . . Sort arrays in memory
setjmp() . . . . . Save machine state for non-local goto
siglongjmp(). . . . . Perform a non-local goto and restore signal mask
sigsetjmp(). . . . . Save machine state and signal mask for non-local jump
shellsort() . . . . . Sort arrays in memory
swab(). . . . . Swap a pair of bytes
system() . . . . . Pass a command to the shell for execution
tempnam(). . . . . Generate a unique name for a temporary file

See Also

libraries

Notes

You do not need to link **libc** explicitly into your programs. The command **cc** always includes it by default.

The macro **offsetof()** is not described above because it does not "live" in **libc**; however, it is a useful, general-purpose entity. For details, see its Lexicon entry.

libcurses — Library

Library of screen-handling functions

libcurses is the library that holds the **curses** screen-handling functions. With **curses**, you can perform rudimentary graphics, even on dumb terminals; the range of routines includes mapping portions of the screen, drawing pop-up windows, creating forms with fields for data entry, and highlighting portions of text.

Implementations of curses

COHERENT uses the Cornell edition of **curses**. This implementation of **curses** reads the **terminfo** data base. It uses eight-bit characters; thus, the COHERENT edition of **curses** can display characters with accents and diacritical marks. Library **libcurses** contains the functions needed to read **terminfo** capability codes; thus, to compile the program **curses_ex.c**, use the following command line:

```
cc curses_ex.c -lcurses
```

Programs that wish to use **curses** *must not* link in both **libcurses** and **libterm**; doing so will cause collisions among library routines. Rather, these programs must link in *only* **libcurses**. On the other hand, if you wish to use the functions that read **terminfo** descriptions, you must link library **libcurses** into your program, even if you are not using any **curses** routines.

If you have special terminal descriptions under **termcap**, the command **captoinfo** converts a **termcap** description into its **terminfo** analogue.

See the Lexicon entries for **termcap** and **terminfo** for more information on this rather confusing topic.

How curses Works

curses organizes the screen into a two-dimensional array of cells, one cell for every character that the device can display. It maintains in memory an image of the screen, called the *curscr*. A second image, called the *stdcur*, is manipulated by the user; when the user has finished a given manipulation, **curses** copies the changes from the *stdcur* to the *curscr*, which results in their being displayed on the physical screen. This act of copying from the *stdscr* to the *curscr* is called *refreshing* the screen. **curses** keeps track of where all changes have begun and ended between one refresh and the next; this lets it rewrite only the portions of the *curscr* that the user has changed, and so speed up rewriting of the screen.

curses records the position of a "logical cursor", which points to the position in the *stdscr* that is being manipulated by the user, and also records the position of the physical cursor. Note that the two are not necessarily identical: it is possible to manipulate the logical cursor without repositioning the physical cursor, and vice versa, depending on the task you wish to perform.

Most **curses** routines work by manipulating a **WINDOW** object. **WINDOW** is defined in the header **curses.h**.

curses defines **WINDOW** as follows:

```
#define WINDOW _win_st
struct _win_st {
        short           _cury, _curx;
        short           _maxy, _maxx;
        short           _begy, _begx;
        short           _flags;
        chtype          _attrs;
        bool            _clear;
        bool            _leave;
        bool            _scroll;
        bool            _idlok;
        bool            _use_keypad;/* 0=no, 1=yes, 2=yes/timeout */
        bool            _use_meta;/* T=use the meta key */
        bool            _nodelay;/* T=don't wait for tty input */
        chtype          **_line;
        short           _firstchar;/* First changed character in the line */
        short           *_lastchar;/* Last changed character in the line */
        short           *_numchngd;/* Number of changes made in the line */
        short           _regtop;/* Top and bottom of scrolling region */
        short           _regbottom;
};
```

Type **bool** is defined in **curses.h**; an object of this type can hold the value of true (nonzero) or false (zero).

The following describes selected **WINDOW** fields in detail.

_cury, _curx Give the Y and X positions of the logical cursor. The upper left corner of the window is, by definition, position 0,0. Note that **curses** by convention gives positions as Y/X (row/column) rather than X/Y, as is usual elsewhere.

_maxy, _maxx Width and height of the window.

_begy, _begx Position of the upper left corner of the window relative to the upper left corner of the physical screen. For example, if the window's upper left corner is five rows from the top of the screen and ten columns from the left, then **_begy** and **_begx** will be set to ten and five, respectively.

_flags One or more of the following flags, logically OR'd together:

_SUBWIN — Window is a sub-window
_ENDLINE — Right edge of window touches edge of the screen
_FULLWIN — Window fills the physical screen
_SCROLLWIN — Window touches lower right corner of physical screen
_FULLINE — Window extends across entire physical screen
_STANDOUT — Write text in reverse video
_INSL — Line has been inserted into window
_DELL — Line has been deleted from window

_ch_off Character offset.

_clear Clear the physical screen before next refresh of the screen.

_leave Do not move the physical cursor after refreshing the screen.

_scroll Enable scrolling for this window.

_y Pointer to an array of pointers to the character arrays that hold the window's text.

_firstch Pointer to an array of integers, one for each line in the window, whose value is the first character in the line to have been altered by the user. If a line has not been changed, then its corresponding entry in the array is set to **_NOCHANGE**.

_lastch Same as **_firstch**, except that it indicates the last character to have been changed on the line.

_nextp Point to next window.

_orig Point to parent window.

When **curses** is first invoked, it defines the entire screen as being one large window. The programmer has the choice of subdividing an existing window or creating new windows; when a window is subdivided, it shares the same *curscr* as its parent window, whereas a new window has its own *stdscr*.

Multiple Terminals

Some applications need to display text on more than one terminal, controlled by the same process. **curses** can handle this, even if the terminals are of different types. The following describes how **curses** output to multiple terminals.

All information about the current terminal is kept in a global variable **struct screen *SP**. Although the screen structure is hidden from the user, the C compiler will accept declarations of variables which are pointers. The user program should declare one screen pointer variable for each terminal it wishes to handle.

The function **newterm()** sets up a new terminal of the given terminal type that is accessed via file-descriptor *fp*. To use more than one terminal, call **newterm()** for each terminal and save the value returned as a reference to that terminal.

To switch to a different terminal, call **set_term()**. It returns the current contents of **SP**. Do not assign directly to **SP** because certain other global variables must also be changed.

All **curses** routines always affect the current terminal. To handle several terminals, switch to each one in turn with **set_term()**, and then access it. Each terminal must first be set up with **newterm()**, and closed down with **endwin()**.

Video Attributes

curses lets you display any combination of video attributes on any terminal. Each character position on the screen has 16 bits of information associated with it. Seven bits are the character to be displayed, leaving bits for nine video attributes. These bits are used for the following modes respectively: standout, underline, reverse video, blink, dim, bold, blank, protect, and alternate-character set. Standout is whatever highlighting works best on the terminal, and should be used by any program that does not need specific or combined attributes. Underlining, reverse video, blink, dim, and bold are the usual video attributes. Blank means that the character is displayed as a space, for security reasons. Protected and alternate character set depend on the terminal. The use of these last three bits is subject to change and not recommended.

The routines to use these attributes include **attron()**, **attroff()**, **attrset()**, **standend()**, **standout()**, **wattroff()**, **wattron()**, **wattrset()**, **wstandend()**, and **wstandout()**. All are described below.

Attributes, if given, can be any combination of **A_STANDOUT**, **A_UNDERLINE**, **A_REVERSE**, **A_BLINK**, **A_DIM**, **A_BOLD**, **A_INVIS**, **A_PROTECT**, and **A_ALTCHARSET**, OR'd together. These constants are defined in **curses.h**. If the particular terminal does not have the particular attribute or combination requested, **curses** will attempt to use some other attribute in its place. If the terminal has no highlighting, all attributes are ignored.

Function Keys

Many terminals have special keys, such as arrow keys, keys to erase the screen, insert or delete text, and keys intended for user functions. The particular sequences these terminals send differs from terminal to terminal. **curses** lets you handle these keys.

A program using function keys should turn on the keypad by calling **keypad()** at initialization. This causes special characters to be passed through to the program by the function **getch()**. These keys have constants that are defined in **curses.h**. They have values starting at 0401, so they should not be stored in a **char** variable, as significant bits will be lost.

A program that uses function keys should avoid using the **<esc>** key: most sequences start with **<esc>**, so an ambiguity will occur. **curses** sets a one-second alarm to deal with this ambiguity, which will cause delayed response to the **<esc>** key. It is a good idea to avoid **<esc>** in any case, because there is eventually pressure for nearly *any* screen-oriented program to accept arrow-key input.

Scrolling Region

Most terminals have a user-accessible scrolling region. Normally, this is set to the entire window, but the calls **setscrreg()** and **wsetscrreg()** set the scrolling region for *stdscr* or the given window to any combination of top and bottom margins. If scrolling has been enabled with **scrollok()**, scrolling takes place only within that window.

TTY Mode Functions

In addition to the save/restore routines **savetty()** and **resetty()**, **curses** contains routines for going into and out of normal tty mode.

The normal routines are **resetterm()**, which puts the terminal back in the mode it was in when curses was started, and **fixterm()**, which undoes the effects of **resetterm()**, that is, restores the "current curses mode". **endwin()** automatically calls **resetterm()**. These routines are also available at the *terminfo* level.

No-Delay Mode

curses offers the call **nodelay()**, which puts the terminal into "no-delay mode". While in no-delay mode, any call to *getch()* returns -1 if nothing is waiting to be read. This is useful for programs that require real-time behavior, where the user watches action on the screen and presses a key when he wants something to happen. For example, the cursor can be moving across the screen, and the user can press an arrow key to change direction. This mode is especially useful for games.

Portability

curses contains several routines to improve portability. Because these routines do not directly relate to terminal handling, their implementation differs from system to system, and the differences can be isolated from the user program by including them in **curses**.

Functions **erasechar()** and **killchar()** return the characters that, respectively, erase one character and kill the entire input line. The function **baudrate()** returns the current baud rate, as an integer. (For example, at 9600 baud, **baudrate()** returns the integer 9600, not the value B9600 from **<sgtty.h>**.) The routine **flushinp()** throws away all typeahead. call **resetterm()** to restore the tty modes. After the shell escape, **fixterm()** can be called to set the tty modes back to their internal settings. These calls are now required, because they perform system-dependent processing.

LEXICON

curses Routines

The following table summarizes the functions and macros that comprise the **curses** library. These routines are declared and defined in the header file **curses.h**.

addch(*ch***) char** *ch***;**
Insert a character into *stdscr*.

addstr(*str***) char** ***str***;**
Insert a string into *stdscr*.

attroff(*at***) int** *at***;**
Turn off video attributes on *stdscr*.

attron(*at***) int** *at***;**
Turn on video attributes on *stdscr*.

attrset(*at***) int** *at***;**
Set video attributes on *stdscr*.

baudrate()
Return the baud rate of the current terminal.

beep() Sound the audible bell.

box(*win, vert, hor***) WINDOW** ***win***; char** *vert, hor***;**
Draw a box. *vert* is the character used to draw the vertical lines, and *hor* is used to draw the horizontal lines. The call

```
box(win, 0, 0)
```

draws a box with solid lines. The call

```
box(win, '|', '-');
```

draws a box around window **win**, using '|' to draw the vertical lines and '-' to draw the horizontal lines. Do not use non-ASCII characters unless you are very sure of the output terminal's identity.

cbreak()
Turn on cbreak mode.

clear() Clear the *stdscr*.

clearok(*win,bf***) WINDOW** ***win***; bool** *bf***;**
Set the clear flag for window *win*. This will clear the screen at the next refresh, but not reset the window.

clrtobot()
Clear from the position of the logical cursor to the bottom of the window.

clrtoeol()
Clear from the logical cursor to the end of the line.

crmode()
Turn on control-character mode; i.e., force terminal to receive cooked input.

delch() Delete a character from *stdscr*; shift the rest of the characters on the line one position to the left.

deleteln()
Delete all of the current line; shift up the rest of the lines in the window.

delwin(*win***) WINDOW** ***win***;**
Delete window *win*.

doupdate()
Update the physical screen.

echo() Turn on both physical and logical echoing; i.e., characters are automatically inserted into the current window and onto the physical screen.

endwin()
Terminate text processing with **curses**.

erase() Erase a window; do not clear the screen.

char *erasechar()
Return the erase character of the current terminal.

flash() Execute the visual bell.

flushinp()
Flush input from the current terminal.

getch() Read a character from the terminal.

getstr(*str***) char** **str**;
Read a string from the terminal.

getyx(*win,y,x***) WINDOW** ***win*****; short** *y,x*;
Read the position of the logical cursor in *win* and store it in *y,x*. Note that this is a macro, and due to its construction the variables *y* and *x* must be integers, not pointers to integers.

idlok(*win*, *flag***) WINDOW** ***win*****; int** *flag*;
Enable insert/delete line operations for window *win*. *flag* must contain the OR'd operations you desire.

inch() Read the character pointed to by the *stdscr*'s logical cursor.

WINDOW *initscr()
Initialize **curses**. Among other things, this function initializes the global variables **LINES** and **COLS**, which give, respectively, the number of lines and the number of columns on your screen.

This is of most use under X Windows. When you change the size of an **xterm** or **xvt** window, the command

```
eval 'resize'
```

resets these variables. The next time you invoke a **curses**-based program, its size will reflect new the dimensions of window.

insch(*ch***) char** *ch*;
Insert character *ch* into the *stdscr*.

insertln()
Insert a blank line into *stdscr*, above the current line.

keypad(*win,flag***) WINDOW** ***win*****; int** *flag*;
Enable keypad-sequence mapping.

char *killchar()
Return the kill character for the current terminal.

leaveok(*win,bf***) WINDOW** ***win*****; bool** *bf*;
Set *win*->**_leave** to *bf*. If set to **TRUE**, refresh will leave the cursor after the last character changed by refresh. This makes sense if you want to minimize the commands sent to the screen and it does not matter where the cursor is.

char *longname(*termbuf, name***) char** ***termbuf, *name*;
Copy the long name for the terminal from *termbuf* into *name*.

meta(*win*, *flag***) WINDOW** ***win*****; int** *flag*;
Enable use of the **meta** key.

move(*y,x***) short** *y,x*;
Move logical cursor to position *y,x* in *stdscr*.

mvaddbytes(*y,x,da,count***) int** *y,x*; **char** ***da*****; int** *count*;
Move to position *y,x* and print *count* bytes from the string pointed to by *da*.

mvaddch(*y,x,ch***) short** *y,x*; **char** *ch*;
Move the logical cursor to position *y,x* and insert character *ch*.

mvaddstr(*y,x,str***) short** *y,x***; char** **str***;*
Move the logical cursor to position *y,x* and insert string *str*.

mvcur(*y_cur,x_cur,y_new,x_new***) int** *y_cur***,** *x_cur***,** *y_new***,** *x_new***;**
Move cursor from position *y_cur,x_cur* to position *y_new,x_new*.

mvdelch(*y,x***) short** *y,x***;**
Move to position *y,x* and delete the character found there.

mvgetch(*y,x***) short** *y,x***;**
Move to position *y,x* and get a character through *stdscr*.

mvgetstr(*y,x,str***) short** *y,x***; char** **str***;*
Move to position *y,x*, get a string through *stdscr*, and copy it into *string*.

mvinch(*y,x***) short** *y,x***;**
Move to position *y,x* and get the character found there.

mvinsch(*y,x,ch***) short** *y,x***; char** *ch***;**
Move to position *y,x* and insert a character into *stdscr*.

mvwaddbytes(*win,y,x,da,count***) WINDOW** **win***;* **int** *y,x***; char** **da***;* **int** *count***;**
Move to position *y,x* and print *count* bytes from the string pointed to by *da* into window *win*.

mvwaddch(*win,y,x,ch***) WINDOW** **win***;* **int** *y,x***; char** *ch***;**
Move to position *y,x* and insert character *ch* into window *win*.

mvwaddstr(*win,y,x,str***) WINDOW** **win***;* **short** *y,x***; char** **str***;*
Move to position *y,x* and insert string *str*.

mvwdelch(*win,y,x***) WINDOW** **win***;* **int** *y,x***;**
Move to position *y,x* and delete character *ch* from window *win*.

mvwgetch(*win,y,x***) WINDOW** **win***;* **short** *y,x***;**
Move to position *y,x* and get a character.

mvwgetstr(*win,y,x,str***) WINDOW** **win***;* **short** *y,x***; char** **str***;*
Move to position *y,x*, get a string, and write it into *str*.

mvwin(*win,y,x***) WINDOW** **win***;* **int** *y,x***;**
Move window *win* to position *y,x*.

mvwinch(*win,y,x***) WINDOW** **win***;* **short** *y,x***;**
Move to position *y,x* and get character found there.

mvwinsch(*win,y,x,ch***) WINDOW** **win***;* **short** *y,x***; char** *ch***;**
Move to position *y,x* and insert character *ch* there.

struct screen ***newterm(***type***,** *fd***) char** **type***;* **int** *fd***;**
Initialize the new terminal *type*, which is accessed via file-descriptor *fd*.

WINDOW ***newwin(***lines***,** *cols***,** *y1***,** *x1***)**
int *lines***,** *cols***,** *y1***,** *x1***;**
Create a new window. The new window is *lines* lines high, *cols* columns wide, with the upper-left corner at position *y1,x1*. It returns a pointer to the **WINDOW** structure that defines the newly created window.

nl() Turn on newline mode; i.e., force terminal to output **<newline>** after **<linefeed>**.

nocbreak()
Turn off cbreak mode.

nocrmode()
Turn off control-character mode; i.e., force terminal to accept raw input.

nodelay(*win***,** *flag***) WINDOW** **win***;* **int** *flag***;**
Make **getch()** non-blocking.

noecho()
Turn off echo mode.

nonl() Turn off newline mode.

noraw()
Turn off raw mode.

overlay(*win1,win2***) WINDOW** ***win1***, *win2***;**
Copy all characters, except spaces, from their current positions in *win1* to identical positions in *win2*.

overwrite(*win1,win2***) WINDOW** ***win1***, *win2***;**
Copy all characters, including spaces, from *win1* to their identical positions in *win2*.

printw(*format[,arg1,...argN]***) char** ***format***; [***data type***] *arg1,..argN***;**
Print formatted text on the standard screen.

raw() Turn on raw mode; i.e., kernel does not process what is typed at the keyboard, but passes it directly to **curses**. In normal (or *cooked*) mode, the kernel intercepts and processes the control characters <*ctrl-C*>, <*ctrl-S*>, <*ctrl-Q*>, and **<ctrl-Y>**. See the entry for **stty** for more information.

refresh()
Copy the contents of *stdscr* to the physical screen.

resetty()
Reset the terminal flags to values stored by earlier call to **savetty**.

saveterm()
Save the current state of the terminal.

savetty()
Save the current terminal settings.

scanw(*format[,arg1,...argN]***) char** ***format***; [***data type***] *arg1,..argN***;**
Read the standard input; translate what is read into the appropriate data type.

scroll(*win***) WINDOW** ***win***;**
Scroll *win* up by one line.

scrollok(*win,bf***) WINDOW** ***win***; bool** *bf***;**
Permit or forbid scrolling of window *win*, depending upon whether *bf* is set to true or false.

setscrreg(*top***,** *bottom***) int** *top***, bottom;**
Set the scrolling region on *stdscr*.

setterm(*name***) char** ***name***;**
Set term variables for *name*.

struct screen *set_term(*new***) struct screen** ***new***;**
Switch output to terminal *new*. It returns a pointer to the previously used terminal.

standend()
Turn off standout mode.

standout()
Turn on standout mode for text. Usually, this means that text will be displayed in reverse video.

WINDOW *subwin(*win***,** *lines***,** *cols***,** *y1***,** *x1***)**
int *win,lines,cols,y1,x1***;**
Create a sub-window in window *win*. The new sub-window is *lines* lines high, *cols* columns wide, and is fixed at position *y1,x1*. Note that the position is relative to the upper-left corner of the physical screen. This function returns a pointer to the **WINDOW** structure that defines the newly created sub-window.

touchwin(*win***) WINDOW** ***win***;**
Copy all characters in window *win* to the screen.

traceoff()
Turn off debugging output.

traceon()
Turn on debugging output

LEXICON

unctrl(*ch***) char** *ch*;
Output a printable version of the control-character *ch*.

waddch(*win,ch***) WINDOW** **win*; **char** *ch*;
Add character *ch* to window *win*.

waddstr(*win,str***) WINDOW** **win*; **char** **str*;
Add the string pointed to by *str* to window *win*.

wattroff(*win,att***) WINDOW** **win*; **int** *att*;
Turn off video attributes *att* for the window pointed to by *win*.

wattron(*win,att***) WINDOW** **win*; **int** *att*;
Turn on video attributes *att* for the window pointed to by *win*.

wattrset(*win,at***) WINDOW** **win*; **int** *att*;
Set the video attributes *att* for the window pointed to by *win*.

wclear(*win***) WINDOW** **win*;
Clear window *win*. Move cursor to position 0,0 and set the screen's clear flag.

wclrtobot(*win***) WINDOW** **win*;
Clear window *win* from current position to the bottom.

wclrtoeol(*win***) WINDOW** **win*;
Clear window *win* from the current position to the end of the line.

wdelch(*win***) WINDOW** **win*;
Delete the character at the current position in window *win*; shift all remaining characters to the right of the current position one position left.

wdeleteln(*win***) WINDOW** **win*;
Delete the current line and shift all lines below it one line up.

werase(*win***) WINDOW** **win*;
Clear window *win*. Move the cursor to position 0,0 but do not set the screen's clear flag.

wgetch(*win***) WINDOW** **win*;
Read one character from the standard input.

wgetstr(*win,str***) WINDOW** **win*; **char** **str*;
Read a string from the standard input; write it in the area pointed to by *str*.

winch(*win***) WINDOW** **win*;
Force the next call to **refresh()** to rewrite the entire screen.

winsch(*win,ch***) WINDOW** **win*; **char** *ch*;
Insert character *ch* into window *win* at the current position. Shift all existing characters one position to the right.

winsertln(*win***) WINDOW** **win*;
Insert a blank line into window *win* at the current position. Move all lines down by one position.

wmove(*win,y,x***) WINDOW** **win*; **int** *y*, *x*;
Move current position in the window *win* to position *y,x*.

wnoutrefresh(*win***) WINDOW** **win*;
Copy the window pointed to by *win* to the virtual screen; do not refresh the real screen.

wprintw(*win,format[,arg1,...argN]***)**
WINDOW **win*; **char** **format*;
[*data type***] arg1,..argN;**
Format text and print it to the current position in window *win*.

wrefresh(*win***) WINDOW** **win*;
Refresh a window.

wscanw(*win,format[,arg1,...argN]***)**

WINDOW *_win_**; char** *_format_**;**
[_data type_**]** _arg1,..argN_**;**
Read standard input from the current position in window _win_, format it, and store it in the indicated places.

wstandend(_win_**) WINDOW** *_win_**;**
Turn off standout (reverse video) mode for window _win_.

wstandout(_win_**) WINDOW** *_win_**;**
Turn on standout (reverse video) mode for window _win_.

wsetscrreg(_win_**,**_top_**,**_bottom_**)WINDOW** *_win_**; int** _top_**,** _bottom_**;**
Set the scrolling region on the window pointed to by _win_.

Color Support

Beginning with release 4.2, COHERENT's implementation of **curses** supports color. **curses** defines colors as a video attribute, like any other. It actually handles pairs of colors — one for the foreground and one for the background. You must initialize a color pair and given it a unique identifying number; then pass the identifier of the color pair to the function **wattron()** to turn on, like any other video attribute.

The header file **<terminfo.h>** defines the following colors, which **curses** recognizes:

COLOR_BLACK
COLOR_RED
COLOR_GREEN
COLOR_YELLOW
COLOR_BLUE
COLOR_MAGENTA
COLOR_CYAN
COLOR_WHITE

Header file **<curses.h>** defines the variables **COLORS**, which holds the maximum number of colors that your console or terminal recognizes; and **COLOR_PAIRS**, which holds the maximum number of color pairs that your console or terminal recognizes. The function **start_color()** initializes both variables.

The following gives the functions and macros with which you can manipulate colors:

can_change_colors()
This function returns TRUE if you can change the definition of a color on your device, and FALSE if you cannot. You should call this function before you invoke the function **init_color()**, described below.

color_content(_color_**,**_r_**,**_g_**,**_b_**); int** _color_**,***_r_**,***_g_**,***_b_**;**
Read the RGB settings for _color_ and write them at the addresses given by _r_, _g_, and _b_.

COLOR_PAIR(_pairnum_**); int** _pairnum_**;**
Return the definition of the color pair identified by _pairnum_. The color pair must have been initialized by a call to **init_pair()**.

has_colors()
Return TRUE if your console or terminal supports color and FALSE if it does not.

init_color(_color_**,**_r_**,**_g_**,**_b_**); int** _color_**,**_r_**,**_g_**,**_b_**;**
Initialize _color_ to the RGB values _r_, _g_, and _b_. _color_ must be greater than zero and less than **COLORS**. _r_,' _g_, and _b_ must each be between zero and 1,000. Not every console or terminal permits you to reset its colors. Call **can_change_colors()** to see if you can alter your device's colors.

init_pair(_pairnum_**,**_fc_**,**_bc_**) int** _pairnum_**,** _fc_**,** _bc_**;**
Initialize the color pair _pairnum_ to the foreground color _fc_ and the background color _bc_. _pairnum_ must be greater than zero and less than **COLOR_PAIRS**. _fc_ and _bc_ must be greater than -1 and less than **COLORS**.

pair_content(_pairnum_**,**_fc_**,**_bc_**) int** _pairnum_**,***_fc_**,***_bc_**;**
Read the foreground and background colors represented by color pair _pairnum_ and write them into the areas pointed to by _fg_ and _bg_.

start_color()
Turn on color processing. This function must precede all other color routines; usually, it immediately follows the function **initscr()**.

LEXICON

A brief example of how to colors appears in the **Examples** section, below.

terminfo Routines

As noted above, **curses** reads terminal descriptions from **terminfo** rather than **termcap**. The library **libcurses** also holds the following functions, with which you can read a **terminfo** description:

fixterm() . . . . . Set the terminal into program mode
putp() . . . . . Write a string into *stdwin*
resetterm() . . . . . Reset the terminal into a saved mode
setupterm() . . . . . Initialize terminal capabilities
tparm() . . . . . Output a parameterized string
tputs() . . . . . Process a capability string
vidattr() . . . . . Set the terminal's video attributes
vidputs() . . . . . Set video attributes into a function

For more information on these routines, see the Lexicon entry **terminfo**, or see each routine's entry in the Lexicon.

If you define the environment variable **TERMINFO**, **curses** checks for the terminal definition in the directory that **TERMINFO** names rather than in the standard directory **/usr/lib/terminfo**. For example, if you set the environmental variable **TERM** is set to **vt100**, then the compiled **terminfo** definition is kept in directory **/usr/lib/terminfo/v/vt100**. However, if you define **TERMINFO** to be **$HOME/testterm**, **curses** first checks **$HOME/testterm/v/vt100**; if that fails, it then checks **/usr/lib/terminfo/v/vt100**. This is useful when you are debugging a **terminfo** entry.

Structure of a curses Program

To use the **curses** routines, a program must include the header file **curses.h**, which declares and defines the functions and macros that comprise the **curses** library.

Before a program can perform any screen operations, it must call the function **initscr()** to initialize the **curses** environment.

As noted above, **curses** manipulates text in a copy of the screen that it maintains in memory. After a program has manipulated text, it must call **refresh()** to copy these alterations from memory to the physical screen. (This is done because writing to the screen is slow; this scheme permits mass alterations to be made to copy in memory, then written to the screen in a batch.)

Finally, when the program has finished working with **curses**, it must call the function **endwin()**. This frees memory allocated by **curses**, and generally closes down the **curses** environment gracefully.

Examples

The first example shows what modes and characters are visible on your system.

```
#include <curses.h>
#define A_ETX 0x03

main()
{
      int c, y = 0, x = 0, attr = A_NORMAL, mask, state = 0, hibit = 0;

      initscr();
      noecho();
      raw();

      erase();
      move(y, 0);
      addstr(
"+ sets - clears Normal Bold Underline blInk Reverse Standout Altmode");
      move(++y, 0);
      refresh();

      for (;;) {
            if (!state) {
                  switch (c = getch()) {
                  case '+':
                        state = 1;
                        break;
```

```
		case '-':
			state = 2;
			break;

		case '\b':
			if (!x)
				break;
			move(y, --x);
			addch(' ');
			move(y, x);
			refresh();
			break;

		case '\r':
		case '\n':
			move(++y, x = 0);
			refresh();
			break;

		case A_ETX:
			noraw();
			echo();
			endwin();
			exit(0);

		default:
			x++;
			addch(c | hibit);
			refresh();
		}
	} else {
		switch (c = getch()) {
		case 'A':   /* turn on high bit of input */
			hibit = (state & 1) << 7;
			state = 0;
			continue;

		case 'B':
			mask = A_BOLD;
			break;

		case 'U':
			mask = A_UNDERLINE;
			break;

		case 'I':
			mask = A_BLINK;
			break;

		case 'R':
			mask = A_REVERSE;
			break;

		case 'S':
			mask = A_STANDOUT;
			break;

		case 'N':   /* normal is an absence of bits */
			if (state == 1) {
				attr = A_NORMAL;
				hibit = state = 0;
				continue;
			}

		default:
			beep();
			continue;
		}
```

```
                if (state == 1)
                      attr |= mask;
                else
                      attr &= ~mask;
                attrset(attr);
                state = 0;
          }
      }
}
```

The next example demonstrates how to use colors in a **curses** program. It selects colors randomly, builds color pairs, and displays a 40-character text string to demonstrate the newly build color pair.

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

void goodbye()
{
      move(23, 0);
      noraw();
      echo();
      endwin();
      exit(EXIT_SUCCESS);
}

main()
{
      int x, y, i;

      initscr();
      start_color();
      noecho();
      raw();

      srand(time(NULL));
      erase();
      if (!has_colors())
            goodbye();

      for (x = 0, y = 0, i = 1; i < COLOR_PAIRS; i++, y++) {
            if (y == 23) {
                  x = 40;
                  y = 0;
            }

            move(y, x);
            init_pair(i, (rand() % COLORS), (rand() % COLORS));
            attrset(COLOR_PAIR(i));
            addstr("Pack my box with five dozen liquor jugs.");
      }

      refresh();
      goodbye();
}
```

The next example shows how to read function keys under **curses**:

```
#include <curses.h>
void text();

main()
{
      int input;

      initscr();
      raw();
      noecho();
      keypad(stdscr, TRUE);
      refresh();
```

```
      while (TRUE) {
            input = wgetch(stdscr);
            switch (input) {
            case 'q':
            case 'Q':
                  endwin();
                  exit(0);

            case KEY_UP:
                  text("cursor up");
                  break;

            case KEY_DOWN:
                  text("cursor down");
                  break;

            case KEY_LEFT:
                  text("cursor left");
                  break;

            case KEY_RIGHT:
                  text("cursor right");
                  break;

            case KEY_F(1):
                  text("function key 1");
                  break;

            case KEY_F(2):
                  text("function key 2");
                  break;

            default:
                  text("some other key");
                  break;
            }
      }
}

void text(s)
register char *s;
{
      move(0, 0);
      clrtoeol();
      printw("Your input was: %s", s);
      refresh();
}
```

See Also

curses.h, libraries, termcap, terminfo
Strang J: *Programming with curses.* Sebastopol, Calif, O'Reilly & Associates Inc., 1986.

Notes

The implementation of **curses** used by the COHERENT system was written by Pavel Curtis of Cornell University. It was ported to COHERENT by Udo Munk.

libedit — Library

Routines to gather and edit user input
/usr/lib/libedit.a

libedit.a is a library of routines that implement a simple tool for entering and editing lines of data. The includes two routines that can be called from a user-level program:

readline()
Read a line of input from the standard input, and let the user edit it.

add_history()
Add a line of edited input into a history buffer, from which the user can retrieve it for further editing and use.

LEXICON

Each is described in its own Lexicon entry.

These routines implement a simple, EMACS-like line-editing interface, much like the one available under the Korn shell **ksh**. To include these routines in an application, just call them as you would any other library function, and link the library **libedit.a** into the executable.

See Also

add_history(), libraries, readline()

Notes

libedit was written by Simmule R. Turner <uunet.uu.net!capitol!sysgo!simmy> and Rich Salz <rsalz@osf.org>.

libgdbm — Library

Library for GNU DBM functions
/usr/lib/libgdbm.a

Archive **libgdbm** contains GNU data-base management (DBM) library of functions. These functions implement a version of the standard UNIX DBM functions, which let you create and manipulate a simple hashed data base.

What is a Hashed Data Base?

A hashed data base consists of a set of records. Each record, in turn, has two elements: a *key* that uniquely identifies the record, and a mass of *data*. For example, if you were creating a data base of the persons who have accounts on your system, the key would be the user's login identifier (because that must be unique), and the data could be the user's full name.

When the GDBM routines add a record to a data base, they do the following:

1. They write the record within a file.
2. They note the size of the record, and its offset within the file.
3. They "hash" each key into a unique number, then write that hash index within a separate index file, along with the offset and size of its associated record. (For a further explanation of hashing and an example implementation, see the Lexicon entry for **strtoul()**.)

By indexing a text file in this manner, a program can find a record much more quickly than it could by simply reading the file from beginning to end. For example, when you mail a message via the mail program **smail**, that program reads a set of aliases to ensure that the message is sent to the right person. If the aliases were kept in a text file, **smail** would have to open the file and read its entire contents every time you sent a mail message; and on a busy system that has a large number of aliases, this can cause a noticeable delay in dispatching a message. By keeping its aliases within a hashed index data base, **smail** greatly reduces the time needed to look up an alias, and so speeds the dispatching of your mail.

Sets of Routines

Library **libgdbm** contains three sets of functions.

- The "GNU DBM" (GDBM) routines. Each of these functions has the prefix **gdbm_**, and is declared in the header file **<gdbm.h>**.
- DBM routines. These re-implement the original UNIX DBM routines. They are declared in header file **<dbm.h>**.
- "New DBM" (NDBM) routines. These re-implement the extended version of the UNIX DBM routines. Each of these functions has the prefix **dbm_**, and is declared in the header file **<ndbm.h>**.

Each set implements a hashed data base, but each has a different provenance, and somewhat different properties and syntax. This library includes all three sets to support the widest possible range of third-party software. If you are writing new software, however, we urge you to use the GDBM routines.

Note that you cannot mix routines from the three sets — you must pick one set, and stick with it. Please note, too, that although this library re-creates the DBM and NDBM sets of routines with regard their calling conventions and return values, internally these re-creations work somewhat different than the UNIX originals; thus, you cannot expect programs compiled with these routines to read binary data bases created by the UNIX originals.

GDBM Routines

The following summarizes the GDBM routines:

gdbm_close() . . . . . Close a GDBM data base
gdbm_delete(). . . . . Delete a record from a GDBM data base
gdbm_exists() . . . . . Check whether a GDBM data base contains a given record
gdbm_fetch() . . . . . Retrieve a record from a GDBM data base
gdbm_firstkey(). . . . . Return the first record from a GDBM data base
gdbm_nextkey() . . . . Return the next record from a GDBM data base
gdbm_open(). . . . . Open a GDBM data base
gdbm_reorganize() . . . Reorganize a GDBM data base
gdbm_setopt(). . . . . Set GDBM options
gdbm_store() . . . . . Add records to a GDBM data base
gdbm_strerror(). . . . . Translate a GDBM error code into text
gdbm_sync(). . . . . Flush buffered GDBM data into its data base

As noted above, these routines are declared in header file **<gdbm.h>**. This header file also defines two structures that the GDBM routines use. The first, **datum**, defines the structure of a data element, either a key or its associated data set:

```
typedef struct {
      char *dptr;
      int dsize;
} datum;
```

This structure lets you have a key and a data element of unlimited length. This is a departure from the orthodox UNIX DBM functions, in which the sizes of the key and the datum are both static.

The other structure, **GDBM_FILE**, holds the information that the GDBM routines use to access a GDBM data base:

```
typedef struct {int dummy[10];} *GDBM_FILE;
```

Error codes are written into global variable **gdbm_errno**, and are defined in header file **<gdbmerrno.h>**.

DBM Routines

The following summarizes the DBM routines:

dbmclose(). . . . . Close a DBM data base
dbminit(). . . . . Open a DBM data base
delete() . . . . . Delete a record from a DBM data base
fetch() . . . . . Fetch a record from a DBM data base
firstkey(). . . . . Retrieve the first record from a DBM data base
nextkey(). . . . . Retrieve the next record from a DBM data base
store() . . . . . Write a record into a DBM data base

As noted above, these routines are declared in header file **<dbm.h>**. It also defines the structure **datum**, which holds a data element, either a key or its associated data set:

```
typedef struct {
      char *dptr;
      int dsize;
} datum;
```

The sizes of the key and its datum together cannot exceed **BSIZE** bytes — that is, the size of one file-system block. **BSIZE** is defined in header file **<sys/buf.h>**; at present, it equals 512 bytes.

Please note that the function **dbmclose()** is non-standard. Programs that use it cannot be recompiled on an orthodox UNIX system.

NDBM Routines

The following summarizes the NDBM routines:

dbm_close() . . . . . Close an NDBM data base
dbm_delete() . . . . . Delete records from an NDBM data base
dbm_dirfno() . . . . . Return the file descriptor for an NDBM **.dir** file
dbm_fetch() . . . . . Fetch a record from an NDBM data base
dbm_firstkey() . . . . . Retrieve the first key from an NDBM data base

LEXICON

dbm_nextkey() . . . . . Retrieve the next key from an NDBM data base
dbm_open() . . . . . Open an NDBM data base
dbm_pagfno() . . . . . Return the file descriptor for an NDBM **.pag** file
dbm_rdonly() . . . . . Set an NDBM data base into read-only mode
dbm_store() . . . . . Store a record into an NDBM data base

As noted above, these routines are declared in header file **<ndbm.h>**. This header file also defines two structures that the NDBM routines use. The first, **datum**, defines the structure of a data element, either a key or its associated data set:

```
typedef struct {
      char *dptr;
      int dsize;
} datum;
```

This structure lets you have a key and a data element of unlimited length.

The other structure, **DBM**, holds the information that the NDBM routines use to access a NDBM data base:

```
typedef struct {int dummy[10];} DBM;
```

See Also

libraries, Programming COHERENT

Notes

libgdbm was written by Philip A. Nelson of the Computer Science Department, Western Washington University, Bellingham (phil@cs.wwu.edu). This Lexicon entry is based on an **info** file written by Pierre Gaumond. **libgdbm** and its documentation are copyright © 1989-1993 by the Free Software Foundation, Inc.

Permission is granted to make and distribute verbatim copies of this manual provided the copyright notice and this permission notice are preserved on all copies.

Permission is granted to copy and distribute modified versions of this manual under the conditions for verbatim copying, provided also that the entire resulting derived work is distributed under the terms of a permission notice identical to this one.

For a full statement of the rights and obligations attached to **libgdbm**, see the file **COPYING** that accompanies the source code to this library.

libm — Library

COHERENT mathematics library
/lib/libm.a

The COHERENT mathematics library **libm** contains the following useful mathematics functions:

acos() . . . . . Calculate inverse cosine
asin() . . . . . Calculate inverse sine
atan() . . . . . Calculate inverse tangent
atan2() . . . . . Calculate inverse tangent of quotient
cabs() . . . . . Calculate complex absolute value
ceil() . . . . . Set numeric ceiling
cos(). . . . . Calculate cosine
cosh() . . . . . Calculate hyperbolic cosine
exp() . . . . . Calculate exponent
fabs() . . . . . Calculate absolute value function
floor() . . . . . Calculate floor function
fmod() . . . . . Calculate modulus for floating-point number
hypot() . . . . . Calculate hypotenuse
j0() . . . . . Calculate Bessel function, order 0
j1() . . . . . Calculate Bessel function, order 1
jn() . . . . . Calculate Bessel function, order *n*
log() . . . . . Calculate natural logarithm
log10() . . . . . Calculate common logarithm
pow() . . . . . Calculate power
sin() . . . . . Calculate sine
sinh() . . . . . Calculate hyperbolic sine

sqrt() . . . . . Calculate square root
tan(). . . . . Calculate tangent
tanh(). . . . . Calculate hyperbolic tangent

See Also

libmp, libraries, math.h

Hart, J.F., et al.: *Computer Approximations.* New York, John Wiley & Sons, 1968.

Press, W.H., Flannery, B.P., Teukolsky, S.A., Vetterling, W.T.: *Numerical Recipes in C.* New York, Cambridge University Press, 1988. *Highly recommended.*

Notes

When programs that contain mathematics routines are compiled, you must explicitly name the mathematics library on the **cc** command line. For example, to compile the example presented under the entry for **acos()**, use the following **cc** command line:

```
cc -f acos.c -lm
```

The **-f** option links in the floating point routines for **printf()**, while the **-lm** option links in the mathematics libraries. Note that the **-lm** option must come *last* on the **cc** command line, or the library will not be searched properly.

The related library **libmp** performs multi-precision arithmetic. With these routines, you can perform arithmetic on extremely large numbers, to an extremely fine precision. For details, see the Lexicon entry for **libmp**.

libmisc — Library

Library of miscellaneous functions

libmisc is a library of miscellaneous C functions. These functions perform such useful tasks as handling such programming tasks as allocation of memory, copying strings, displaying variables from C with COBOL-like "picture" descriptions, and supporting virtual arrays via secondary storage.

Source code for **libmisc** is kept in the compressed **tar** archive **/usr/src/misc.tar.Z**. To extract the files into a new subdirectory called **misc**, type the command:

```
zcat /usr/src/misc.tar.Z | tar xvf -
```

To build the library, type the following:

```
cd misc
make
```

This compiles the **libmisc** routines and builds the library **libmisc.a**.

Archive **misc.tar** also includes the header file **misc.h** which protypes these functions, and declares the global variables and constants they use. You must include this header file in any program that uses any of the **libmisc** functions.

Functions

The following summarizes the functions in **libmisc.a**:

char * **alloc(***n***) unsigned** *n***;**
malloc() *n* bytes and initialize them to zero. Abort on failure.

int approx(*a***,** *b***) double** *a***,** *b***;**
If *a* and *b* are within epsilon, return one; otherwise, return zero. epsilon is a visable **double**.

char ***ask(***reply***,** *msg***, ...) char** ****reply***, ****msg***;**
Print a message and retrieve the user's reply. *msg* is a **printf()**-style format string that formats the text pointed to by any trailing arguments. **ask()** constructs the prompt message from *msg* and prints it on the standard output; then reads a line from stdin, stores it in the place pointed to by *reply*, and returns its address. *reply* must point to enough space to hold the user's reply.

For example,

```
sscanf(ask(buff, "%d numbers", 3), &a, &b, &c);
```

prints the message

```
Enter: 3 numbers
```

writes the user's reply into **buff**, and hands its address to **sscanf()**.

void banner(*word***,** *pad***) char** ***word*****; int** *pad***;**
Print *word* on stdout as a banner, preceded by *pad* spaces. Each letter of the banner is fashioned from many occurrences of itself. This is especially useful if you wish your listings to look like truly professional, mainframe printouts.

bedaemon()
bedaemon() turns the calling program into a daemon. A *daemon* is a process that executes in the background, without the usual connections to standard I/O streams and terminals. Examples are **cron** and **uuxqt**. To ensure proper operation in connection with other system software, any program that you intend to run as a daemon should call **bedaemon()** as its first step. This call closes all inherited, open-file descriptors, detaches the process from its inherited process group and controlling terminal, sets current directory to '/', and sets **umask** to zero. For further information on daemon processes, see *Unix Network Programming* by W. Richard Stevens (Englewood Cliffs, NJ, Prentice-Hall Inc, 1990), §2.6.

unsigned short crc16(*p***) char** ***p*****;**
Compute the 16-bit cyclic redundency check (crc16) of the string pointed to by *p*, and return it. This function is very useful for building hash tables or checking differences between strings.

void fatal(*msg***, ...) char** ***msg*****;**
Print an error message and call **exit(1)**. *msg* is a **printf()**-style format string; trailing arguments must to point to data.

char ***getline(***ifp***,** *lineno***) FILE** ***ifp*****; int** ***lineno*****;**
Get a line from the input file pointed to by *ifp*. This function returns the address of the line, or NULL to indicate the end of file. **getline()** calls **malloc()** to acquire space for the line, and allows lines to be continued with a \-whitespace. It also implements **lineno**.

getline() recognizes the following escape sequences:

#	to end of line is passed
****	whitespace through end of line is passed
\n	newline
\p	literal '#'
\a	alarm
\b	backspace
\r	carrage return
\f	form feed
\t	tab
****	backslash
*****ddd*	octal number

All other \ sequences are errors that **getline()** reports on stderr.

tm_t ***jday_to_tm(***jd***) jday_t** *jd***;**
Turn a Julian date to **tm** (time) structure. The Julian date is the number of days since the beginning of the Julian calendar, January 1, 4713 B.C.; it is a good way to store dates in a system-independent manner, such as in a data base. Structure **jday_t** is defined in **misc.h**. Structure **tm** is defined in **<time.h>**.

time_t jday_to_time(*jd***) jday_t** *jd***;**
Turn Julian date structure to COHERENT time. Type **time_t** is defined in header file **<sys/types.h>**.

void splitter(*ofp***,** *line***,** *limit***) FILE** ***ofp*****; char** ***line*****; int** *limit***;**
Write *line* into file *ofp*, splitting it into chunks less than *limit* bytes long. **splitter()** inserts a \ between chunks, and attempts to do this on white-space boundaries. **splitter()** produces a long line rather than split on non-whitespace. If *line* does not end in a newline, **splitter()** adds one. This is the inverse of **getline()**.

int is_fs(*special***) char** ***special*****;**
Check whether *special* names a well-formed file system. Users should never put file systems on **/dev/ram1**, but for multi-system software, like **compress**, it is smart to test.

is_fs() returns -1 if *special* is not a device, or if **open()**, **read()**, or **seek()** fails. It returns zero if no file system was found, or one if *special* names a legal file system.

char *lcase(*st***) char ***str***;**
Convert every character in *str* to lower case. Note that this works only with the U.S. dialect of English; it does not work with German or other languages that use characters in the upper half of the ASCII table.

char *match(*string***, *** pattern***, *** fin***) char ***string***, ***pattern***, **** fin***;**
match() resembles **pnmatch()**, except that it returns the address of the pattern matched. *fin* is aimed past the end of the pattern found; that is, **match()** finds a pattern and tells you where it is.

char *metaphone(*word***) char ***word***;**
Translate *word* into a short phonetic equivalent for easy lookup. It resembles Knuth's **soundex** method, except that it uses a superior algorithm.

char *newcpy(*str***) char ***str***;**
Create a NUL-terminated copy of *str* and return its address. Call **fatal()** if there is no space.

char *pathn(*name***, *** envpath***, *** deflpath***, *** access***)**
char ***name*, ***envpath***, ***deflpath***, ***access***;**
pathn() looks for file *name*. It searches the directories named in the environmental variable *envpath*. If the user has not set *envpath*, or if it is NULL, **pathn()** searches the default path *deflpath*. *name* must have *access* permission. **pathn()** returns the full path to the file found. For example:

```
pathn("helpfile", "LIBPATH", "/lib", "r")
```

searches the directories named in **LIBPATH** for file **helpfile**, for which the user must have read permission. If **LIBPATH** is not set, **pathn()** searches **/lib** for **helpfile**.

#include <regexp.h>
regexp *regcomp(*exp***) char ***exp***;**
int regexec(*prog***, *** string***) regexp ***prog***; char ***string***;**
regsub(*prog***, *** source***, *** dest***) regexp ***prog***; char ***source***; char ***dest***;**
regerror(*msg***) char ***msg***;**
These functions implement a standard method for parsing regular expressions. **regcomp()** turns a regular expression into a structure of type **regexp** and returns a pointer to it. **regexec()** matches *string* against the regular expression in *prog*. It returns one if *string* matches *exp*, and zero if it does not. **regsub()** copies *source* to *dest*, and makes substitutions according to the most recent **regexec()** performed using *prog*. **regerror()** is called whenever an error is detected in **regcomp()**, **regexec()**, or **regsub()**. See **regexp.doc** for details.

long randl()
Return a long random number uniformly distributed between 1 and 2,147,483,562. This comes from *Communications of the ACM*, volume 31, number 6. See **srandl()**, below.

char *replace(*s1***, *** pat***, *** s3***, *** all***, *** matcher***) char ***s1***, ***pat***, ***s3***, (***matcher***)();**
Replace one or all occurrences of *pat* in string *s1* by *s3*, and return the result. The definition of match is set by *matcher*. This calls the user-defined function

matcher(*sw***, *** pat***, &***fin***).**

The *matcher* must return the address of the pattern match and its end in **&***fin***. match()** is a valid example of *matcher*. It replaces the first occurrence, or all occurrences of the pattern, and returns the new pattern. The new pattern has been **alloc()**'d.

showflag(*data***, *** flags***, *** output***) long ***data***; char ***flags***, ***output***;**
Turn the bits in *data* to the flags in *flags* or '-' in the string *output*, which must be as long as *flags*.

char *skip(*s1***, *** matcher***, *** fin***) char ***s1***, ****fin***; int (****matcher***)();**
Skip all initial characters in string *s1* that fail when examined *matcher*. *matcher* is usually a character function, e.g., **isdigit()**. It returns the first character skipped. **skip()** points *fin* at the character after the skip.

char *span(*s1***, *** matcher***, *** fin***) char ***s1***, ****fin***; int (****matcher***)();**
Span all initial characters in string *s1* that match when examined by *matcher*. *matcher* is usually a character function, e.g., **isdigit()**. It returns the first character spanned. **span()** points *fin* at the character after the span.

LEXICON

srandl(*seed1***,** *seed2***) long** *seed1***,** *seed2***;**
randl() needs two seeds; **srandl()** sets them. Use it only if you need to repeat a random-number sequence.

strchtr(*from***,** *to***,** *c***,** *def***)**
char **from***, **to***; **int** *c***,** *def***;**
Look up the character *c* in the string *from*. Return the corresponding character in the string *to* if it is found; otherwise, return the default character *def*.

For example, consider the call:

```
strchtr("ab", "xy", c, d);
```

If variable *c* equals 'a', then **strchtr()** returns 'x'; if *c* equals 'b', then it returns 'y'; otherwise, it returns the value of *d*

strcmpl(*s1***,** *s2***)**
Case-insensitive string comparison. Resembles **strcmp()**.

jday_t time_to_jday(*time***) time_t** *time***;**
Turn COHERENT time to Julian date structure. The Julian date is the number of days since the beginning of the Julian calendar, January 1, 4713 B.C. The structure **jday_t** is defined in **misc.h**. Type **time_t** is defined in **<sys/types.h>**.

jday_t tm_to_jday(*tm***) tm_t** **tm***;**
Turn the time structure **tm** date into Julian date structure. Structure **tm** is defined in **<time.h>**.

char **trim(***s***) char** **s***;**
Remove trailing whitespace from string *s*.

ucase(*s***) char** **s***;**
Convert a string to upper case.

usage(*s***) char** **s***;**
Print string *s* and call **exit(1)**.

xdump(*p***,** *length***) char** **p***;**
Print on stdout a vertical hexadecimal dump of string *p*.

A vertical hexadecimal dump prints as three lines. The top line is the display character, or '.' if the character cannot be displayed cleanly. The next two lines are the hexadecimal numerals. The data are blocked into groups of four bytes.

xopen(*filename***,** *acs***) char** **filename***, **acs***;**
Like **fopen()**, but call **fatal()** if the open fails.

yn(*question***, ...) char** **question***;**
Ask a question and retrieve a 'Y' or 'N' answer. *question* is a **printf()**-style format string; any trailing parameters should point to data used in *question*. **yn()** returns one if the user answers 'Y' or 'y', and returns zero if she answers 'N' or 'n'.

Virtual Memory

The following functions are part of a virtual memory system for COHERENT 286. Occasionally, users port programs like **compress** to COHERENT 286 that use a small number of very large arrays. Because COHERENT 286 is a SMALL-model operating system, special provision must be made for arrays too large to fit into a 64-kilobyte data segment. The following functions enable programs that are to be run under COHERENT 286 use very large arrays:

void vinit(*filename***,** *ram***) char** **filename***; **unsigned** *ram***;**
Initialize the virtual-memory system, using *filename* for work. *filename* may be a raw device, such as */dev/rram1*. *ram* is the amount of buffer space to give the system — the more, the better.

void vshutdown()
Shut the virtual-memory system, and make it restartable.

unsigned vopen(*amt***) unsigned long** *amt***;**
Set up a virtual-memory object. For example, if you want to emulate having a 100,000-byte array and a 50,000-byte array, use the call

```
vid1 = vopen(100000L); vid2 = vopen(50000L);
```

This does some checking and tells the system that any reference to *vid2* will be between 100,000 and 150,000 in the virtual file.

char ***vfind(***vid***,** *disp***,** *dirty***)**
unsigned *vid***,** *dirty***; unsigned long** *disp***;**
Find a character in the virtual system, mark the block's dirty bit if the access is to write. Given the example in **vopen()**, if you want to find the 1,000th byte in *vdi1*, use the call:

```
c = *(vfind(vdi1, 1000L, 0));
```

To change the 2000th byte in *vid2* **d**, use the call

```
*(vfind(vid2, 2000L, 1)) = d;
```

Note that the dirty indicator tells the system of the change so that the block will be written back before it is read over. Blocks are 512 bytes long, so **int**'s or **long**'s can be read or written without multiple accesses to **vfind()**.

File Locking

libmisc holds a number of routines with which you can lock and unlock files and devices. It is adapted from the mechanism used by the COHERENT implementation of UUCP.

Lock files are created in **$SPOOLDIR**. A lock file is given the name **LCK..***resource*. It contains a decimal representation of the process identifier (pid) of the process that created the lock.

You can provide an alternate pid by using one of the 'n' routines — i.e., **locknrm()**, **lockntty()**, and **unlockntty()**. The unlocking routines regard a pid of zero as an override — they remove the lock regardless of which process created the lock.

For a tty device, *resource* is a string that consists of a decimal representation of its major number, a decimal point, and the lower five bits of its minor number.

Each routine takes a string that names the resource to lock or unlock. The "tty" routines (i.e., **lockntty()**, **locktty()**, **unlockntty()**, and **unlocktty()**) want the base name of the tty to be locked (without the **/dev/** part).

Every lock routine returns zero on failure and one on success.

lockit(*resource***)** char ***resource***;**
Use a resource string to lock a tty.

lockexist(*resource***)** char ***resource***;**
Check whether a lock file exists for the tty with *resource*.

locknrm(*resource***,** *pid***) char** ***resource***; int** *pid***;**
Remove a lock file for a tty owned by process *pid*.

lockntty(*tty***,** *pid***) char** ***tty***; int** *pid***;**
Lock a tty for process *pid*.

lockrm(*resource***)** char ***resource***;**
Remove a lock file for tty with *resource*.

locktty(*tty***) char** ***tty***;**
Lock a tty.

lockttyexist(*tty***) char** ***tty***;**
Check whether a given tty is locked.

unlockntty(*tty***,** *pid***) char** ***tty***; int** *pid***;**
Unlock a tty for process *pid*. Unlocking always succeeds.

unlocktty(*tty***) char** ***tty***;**
Unlock a tty that the current process owns.

unlockit(*resource***,** *pid***) char** ***resource***; int** *pid***;**
Unlock something for process *pid*.

LEXICON

Templates and Pictures

libmisc includes a function, **picture()**, for formatting numeric strings. It has the following syntax:

double picture(*dble***,** *format***,** *output***)**
double *dble***; char** **format***,* **output***;*

picture() performs numeric formatting under C. It resembles masking functions built into COBOL and BASIC, but is superior to either. *dble* gives the number to format; *format* gives the format mask; and *output* points to the area into which the formatted number is written. *output* must be at least as large as *format*. If *dble* overflows the picture, **picture()** returns the overflow.

The following summarizes the values that can appear in the *format* string. Note that throughout, the symbol **<sp>** indicates a space character, not the literal string "<sp>".

9 Provide a slot for a number. Passing 5.000 through a mask of **999 CR** gives "005". Passing -5.000 through a mask of **999 CR** yields "005 CR". Note that **picture()** does not recognize the characters 'C' and 'R' as being special. Trailing non-special characters print only if the number is negative.

Z Provide a slot for a number, but suppress leading zeroes. For example, passing 1034.000 through a mask of **ZZZ,ZZZ** gives "<sp><sp>1,034". Note that **picture()** does not recognize a comma as being a special character. **picture()** prints embedded non-special characters only if they are preceeded by significant digits.

J Provide a slot for a number, but shrink out leading zeros. For example, passing 1034.000 through a mask of **JJJ,JJJ** yields "1,034".

K Provide a slot for a number, but shrink out any zeros. For example, passing 70884.000 through a mask of **K9/K9/K9** yields "7/8/84".

$** Float a dollar sign to the left of the displayed number. For example, passing 105.000 through a mask of **$ZZZ,ZZZ yields "<sp><sp><sp><sp>$105".

. Separate the number between decimal and integer portions. For example, passing 105.670 through a mask of **Z,ZZZ.999** yields "<sp><sp>105.670".

T Provide a slot for a number, but suppress trailing zeroes. For example, passing 105.670 through a mask of **Z,ZZ9.9TT** yields "<sp><sp>105.67<sp>".

S Provide a slot for a number, but shrink out trailing zeroes. For example, passing 105.670 through a mask of **Z,ZZ9.9SS** yields "<sp><sp>105.67".

- Float a negative sign in front of negitive numbers. For example, passing 105.000 through a mask of **-Z,ZZZ** yields "<sp><sp><sp<105", whereas passing -105.000 through a mask of **-Z,ZZZ** yields "<sp><sp>-105".

(Acts like **-**, but prints a parenthesis. For example, passing 105.000 through a mask of **(ZZZ)** yields "<sp>105<sp>", whereas passing -5.000 through a mask of **(ZZZ)** yields "<sp><sp>(5)".

+ Float a **+** or **-** in front of the number, depending on its sign. For example, pasing 5.000 through a mask of **+ZZZ** yields "<sp><sp>+5", whereas passing -5.000 through a mask of **+ZZZ** yields "<sp><sp>-5".

***** Fill all spaces to right with asterisks. For example, passing 104.100 through a mask of ***ZZZ,ZZZ.99** yields "*****104.10"; whereas passing 104.100 through a mask of ***$ZZZ,ZZZ.99** yields "*****$104.10". **picture()** returns any overflow as a **double**. For example, passing -1234.000 through a mask of **(ZZZ)** yields "(234)" with an overflow of -1.0; passing 123.400 through a mask of **99** yields "23" with an overflow of 1.0; and passing 1200.000 through a mask of **ZZ** yields "00" with an overflow of 12.0.

Files

/usr/src/misc.tar.Z — Compressed **tar** archive of sources

See Also

libraries, tar, zcat

Notes

The **misc** library is provided on an *as-is* basis only. *Caveat utilitor!*

libmp — Library

Library for multiple-precision mathematics
/usr/lib/libmp.a

The COHERENT library **libmp** contains routines that allow you to perform multiple-precision arithmetic. These functions manipulate a data structure called a **mint**, or "multiple-precision integer," which the header file **mprec.h** defines as follows:

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

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

The following gives the multiple-precision routines:

gcd(). . . . . Set variable to greatest common divisor
ispos() . . . . . Return if variable is positive or negative
itom(). . . . . Create a multiple-precision integer
madd() . . . . . Add multiple-precision integers
mcmp(). . . . . Compare multiple-precision integers
mcopy() . . . . . Copy a multiple-precision integer
mdiv(). . . . . Divide multiple-precision integers
min() . . . . . Read multiple-precision integer from stdin
minit() . . . . . Condition global or auto multiple-precision integer
mintfr(). . . . . Free a multiple-precision integer
mitom() . . . . . Reinitialize a multiple-precision integer
mneg() . . . . . Negate multiple-precision integer
mout() . . . . . Write multiple-precision integer to stdout
msqrt() . . . . . Compute square root of multiple-precision integer
msub() . . . . . Subtract multiple-precision integers
mtoi(). . . . . Convert multiple-precision integer to integer
mtos(). . . . . Convert multiple-precision integer to string
mult(). . . . . Multiply multiple-precision integers
mvfree() . . . . . Free multiple-precision integer
pow() . . . . . Raise multiple-precision integer to power
rpow(). . . . . Raise multiple-precision integer to power
sdiv() . . . . . Divide multiple-precision integers
smult() . . . . . Multiply multiple-precision integers
spow(). . . . . Raise multiple-precision integer to power
xgcd(). . . . . Extended greatest-common-divisor function
zerop() . . . . . Indicate if multi-precision integer is zero

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 writes the quotient and remainder in *q* and *r*. *b* must not be zero. The results of the operation are defined by the following conditions:

1. $a=q*b+r$

2. The sign of *r* equals 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 $0 \le n \le 127$. **sdiv()** is like **mdiv()**, except the second argument is an integer in the range $1 \le n \le 128$, 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 two to 16. Permissible digits are 0 through 9 and A through F (lower-case letters 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 zero and one. **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 **libmp** modules into an executable object, use the argument **-lmp** at the end of the **cc** command.

Example

The following example converts a string into a multi-precision integer.

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

/*
 * "ibase" is an int which contains the input base used by "stom".
 * It should be between 2 and 16.
 */
int ibase = 10;
```

```
/*
 * stom() reads in a number in base ibase from string 'a' and returns
 * pointer to multiple-precision integer.
 */
mint *stom(s)
register char *s;
{
      char cval;
      mint c = {1, &cval};
      register int ch;
      char mifl = 0;    /* leading minus flag */
      static mint number;

      mcopy(mzero, &number);  /* set number to zero */
      if ((ch = *s) == '-') { /* skip leading '-' */
            mifl = 1;
            ch = *++s;
      }

      /* scan thru string 's', building result in "number" */
      while (isascii(ch) && isdigit(ch)) {
            cval = (isdigit(ch) ? ch - '0': ch - 'A');
            smult(&number, ibase, &number);
            madd(&number, &c, &number);
            ch = *++s;
      }

      if (mifl)   /* adjust sign of a "number" */
            mneg(&number, &number);
      return(&number);
}

/* simple test for "stom" */
main()
{
      char  buffer[80];

      printf("Input string ? ");
      gets(buffer);
      mout(stom(buffer)); /* Print in stdout multiple-precision int */
}
```

Files

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

See Also

bc, dc, libraries, malloc(), mprec.h

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.

LIBPATH — Environmental Variable

Directories that hold compiler phases and libraries

LIBPATH names the directories that hold the phases of the COHERENT C compiler, the run-time start-up modules, and libraries. **cc** searches these directories as it orchestrates the compiling and linking of a program written in C or assembly-language.

A typical definition is:

```
export LIBPATH=:/lib:/usr/lib
```

This searches the current directory '.', then **/lib**, then **/usr/lib**.

If you have not set **LIBPATH** in your **.profile**, **cc** uses the default **LIBPATH** that is set in header file **path.h**. This definition is adequate for all standard installations of COHERENT.

See Also

cc, environmental variables, ld

libraries — Overview

A **library** is an archive file of commonly used functions that have been compiled, tested, and stored for inclusion in a program at link time.

The COHERENT system stores its libraries in two directories: **/usr/lib** and **/lib**. or their subdirectories: The following libraries are kept in **/usr/lib**:

libcurses.a . . . . . **curses** library and **terminfo** functions
libedit.a . . . . . Routines to gather and edit user input
libgdbm.a . . . . . Library for GNU DBM functions
libl.a . . . . . **lex** library
libmp.a . . . . . Multi-precision arithmetic library
libsocket.a . . . . . **sockets** emulation library
libterm.a . . . . . Functions to read **termcap** data
liby.a . . . . . **yacc** library
lib.b . . . . . **bc**'s function library (in **bc** source)

The following libraries are kept in **/lib**:

libc.a . . . . . General functions and system calls
libm.a . . . . . Mathematics routines

In addition, COHERENT comes with a library of miscellaneous routines, called **libmisc**. See the Lexicon article **libmisc** for information on how to prepare this library for use.

Lexicon Articles

The following Lexicon articles introduce the library functions included with the COHERENT system:

libc
libcurses
libedit
libgdbm
libm
libmisc
libmp
libsocket
libterm

See Also

ar, C language, Programming COHERENT

libsocket — Library

Library of communications routines

libsocket is a library of routines that emulate the Berkeley **sockets** library. It includes the following functions:

accept() . . . . . Accept a connection on a socket
bind() . . . . . Bind a name to a socket
bitcount() . . . . . Count bits in a bit-mask
connect() . . . . . Connect to a socket
endhostent() . . . . . Close file **/etc/hosts**
endnetent() . . . . . Close network file
endprotoent() . . . . . Close protocols file
endservent() . . . . . Close protocols file
ffs() . . . . . Translate a bit mask into an integer value
getdtablesize() . . . . . Get the number of files a process can open
gethostbyaddr() . . . . . Retrieve host information by address
gethostbyname() . . . . Retrieve host information by name
gethostname() . . . . . Get the name of the local host
getnetbyaddr() . . . . . Get a network entry by address
getnetbyname() . . . . . Get a network entry by address

getnetent() . . . . . Fetch a network entry
getpeername() . . . . . Get name of connected peer
getprotobyname() . . . Get protocol entry by protocol name
getprotobynumb() . . . Get protocol entry by protocol number
getprotoent() . . . . . Get protocol entry
getservbyname() . . . . Get a service entry by name
getservbyport() . . . . . Get a service entry by port number
getservent() . . . . . Get a service entry
getsockname() . . . . . Get the name of a socket
getsockopt(). . . . . Read a socket option
gettimeofday() . . . . . Berkeley time function
inet_addr(). . . . . Transform an IP address from text to binary
inet_network() . . . . . Transform an IP address from text to an integer
listen() . . . . . Listen for a connection on a socket
random(). . . . . Return a random number
recv() . . . . . Receive a message from a connected socket
recvfrom() . . . . . Receive a message from a socket
select() . . . . . Check whether sockets are ready for activity
send(). . . . . Send a message to a connected socket
sendto() . . . . . Send a message to a socket
sethostent(). . . . . Open and rewind file **/etc/hosts**
setnetent(). . . . . Open and rewind file **/etc/networks**
sethostent(). . . . . Open and rewind file **/etc/hosts**
setprotoent() . . . . . Open the protocols file
setservent() . . . . . Open the services file
setsockopt(). . . . . Set a socket option
shutdown() . . . . . Replace function to shut down system
SOCKADDRLEN(). . . . Return length of an address
socket(). . . . . Create a socket
socketpair() . . . . . Create a pair of sockets
srandom() . . . . . Seed the random-number generator
strcasecmp() . . . . . Case-insensitive string comparison
strcasencmp(). . . . . Case-insensitive string comparison
usleep(). . . . . Sleep briefly

Function **socket()** creates a socket; the caller dictates the type of socket to be created, and the communications protocol that it comprehends. **socket()** returns a descriptor, which resembles a file descriptor and which can be passed to the system calls **read()** and **write()** to exchange information with whatever plugs itself into that socket. (For details, see the **Notes** section at the end of this article.)

Function **bind()** binds the newly created socket to a file that you name. To await a connection with another process, invoke the function **listen()**; this alerts the system to the fact that you (via your socket) await messages of a given type. Function **select()** checks whether one or more sockets are ready to be written to, or hold data that need to be read. When a message becomes available, invoke function **accept()** to accept communication with the process that wishes to connect to your socket. These functions generally are used by "server" sockets.

Function **connect()** directly establishes connection with a server socket via its name (that is, via the file to which it is bound). Once connection is established, information can be exchanged via the COHERENT system calls **read()** and **write()**.

System Files

The socket library manipulates the following files. Each is described in its own Lexicon entry:

hosts . . . . . Names and addresses of hosts on the local network
hosts.equiv . . . . . Name equivalent hosts
hosts.lpd. . . . . Local system name and domain
inetd.conf . . . . . Configure the Internet daemons
networks. . . . . Name remote networks
protocols. . . . . Name supported protocols
services . . . . . List supported TCP/IP services

Example

For following gives a pair of programs that demonstrate sockets. They were written by John Dhuse

(jdhuse@sedona.intel.com).

The example consists of two programs, **server.c** and **client.c**. Compile each with the switch **-lsocket**. To see how they work, run each in its own virtual console or **xterm** window. Do not run them in the background; otherwise, you will not be able to work with them interactively. Be sure to start up **server** first, as it creates the socket into which **client** plugs itself.

Each process gives you a prompt; you can type commands into each. **server** recognizes the following commands:

? Print the command menu

c Call **select()** to check the socket. **client** displays the status of the socket.

s Send a string to **server**. **client** prompts for the string, reads up to 20 characters, and writes it to the socket.

r Read from the socket. **client** prompts for the number of bytes to read, and clips any response to a maximum of 20.

q Close the socket and terminate the server process.

server recognizes the following commands:

? Print the command menu.

c Call **select()** to check the socket.

r Read from the socket. **server** does not prompt for the number of bytes to read, but tries to read the entire contents of the socket, up to a maximum of 20 bytes.

e Echo the read message back to the client. The server cannot send its own message the client, just echo what it received.

q Close the socket, terminate the server, and **unlink()** the socket file.

The following gives the source for **server.c**:

```
#include <errno.h>
#include <stdio.h>
#include <stdlib.h>
#include <sys/ioctl.h>
#include <sys/socket.h>
#include <sys/time.h>
#include <sys/types.h>
#include <sys/un.h>

main()
{
      int sd, nsd, err,i,j,rdfds[2],wrtfds[2];
      int efds[2],done,r;
      int arg=1;
      struct sockaddr_un server;
      char *sock_name = "u0";
      char buf[20];
      char command,line[80];
      struct timeval timeout;

      /* clear our address */
      bzero((char *)&server, sizeof(server));

      /* create socket */
      if ((sd = socket(AF_UNIX, SOCK_STREAM, 0)) <= 0) {
            err = errno;
            fprintf(stderr, "server: can't create socket\n");
            fprintf(stderr, "server: errno = %d\n", err);
            exit(EXIT_FAILURE);
      }

      server.sun_family = AF_UNIX;
      bcopy(sock_name,server.sun_path,strlen(sock_name));
```

```
  /* bind the socket */
if ((bind(sd, (struct sockaddr *)&server, sizeof(server))) != 0) {
      err = errno;
      fprintf(stderr, "server: can't bind socket\n");
      fprintf(stderr, "server: errno = %d\n", err);
      close(sd);
      exit(EXIT_FAILURE);
}

  /* listen on the socket */
if ((listen(sd, 1)) != 0) {
      err = errno;
      fprintf(stderr, "server: can't listen on socket\n");
      fprintf(stderr, "server: errno = %d\n", err);
      close(sd);
      exit(EXIT_FAILURE);
}

/* accept connections on the socket */
if ((nsd = accept(sd, (struct sockaddr*)0, (int *)0)) == -1) {
      err = errno;
      fprintf(stderr, "server: can't accept connection\n");
      fprintf(stderr, "server: errno = %d\n", err);
      close(sd);
      exit(EXIT_FAILURE);
}

printf("accepted client connection fd %d\n",nsd);
  /* set to non-blocking io */
ioctl(nsd,FIOSNBIO,&arg);

/* echo every message back to client, exit on terminate string */
printf("entering command loop\n");
command = 'a';
while (command != 'q') {
      printf("server> ");
      scanf("%s",line);
      sscanf(line,"%c", &command);
      switch (command) {

      case 'c' :
            /* set up for select */
            rdfds[0] = 1 << nsd; rdfds[1] = 0;
            wrtfds[0] = 1 << nsd; wrtfds[1] = 0;
            efds[0] = 1 << nsd; efds[1] = 0;
            timeout.tv_sec = 0; timeout.tv_usec = 0;
            r = select(nsd+1,rdfds,wrtfds,efds,(struct timeval *)NULL);
            err = errno;

            if (r < 0)
                      printf("select() returned errno %d\n",err);
            else {
                    if (rdfds[0] & (1 << nsd))
                        printf("socket has data to be read\n");
                  if (wrtfds[0] & (1 << nsd))
                            printf("data can be written to socket\n");
                  if (efds[0] & (1 << nsd))
                        printf("select reports exception on socket\n");
            }
            break;
```

```
            case 'r' :
                      bzero(&buf[0], sizeof(buf));
                  j = read(nsd,buf,sizeof(buf));
                  err = errno;
                  if (j < 0)
                        printf("read() returned errno %d\n",err);
                  else
                        printf("got %d bytes, msg is >%s<\n",j,buf);
                  break;

            case 'e' :
                  printf("echoing >%s< (%d bytes) to client\n",buf,j);
                  write(nsd,&buf[0],j);
                  break;

            case 'q' :
                  close(nsd);
                  close(sd);
                  unlink(sock_name);
                  break;

            case '?' :
                  printf("commands:\n");
                  printf("   c - check the socket\n");
                  printf("   ? - this help message\n");
                  printf("   r - read from socket\n");
                  printf("   e - echo received message to client\n");
                  printf("   q - close socket and quit\n");
                  break;

              default :
                  printf("\n");
                  break;
            }
      }
}
```

The following gives the source for **client.c**:

```
#include <errno.h>
#include <fcntl.h>
#include <stdio.h>
#include <stdlib.h>
#include <sys/ioctl.h>
#include <sys/socket.h>
#include <sys/time.h>
#include <sys/types.h>
#include <sys/un.h>

main()
{
      int sd, err, i, j, flags;
      int arg=1;
      struct sockaddr_un client;
      char *address="u0";
      char buf[20];
      char command,line[80];
      int rdfds[2],wrtfds[2];
      struct timeval timeout;

      /* clear our address */
      memset((char *)&client,0, sizeof(client));

      /* create socket */
      if ((sd = socket(AF_UNIX, SOCK_STREAM, 0)) <= 0) {
            err = errno;
            fprintf(stderr, "client: can't create socket\n");
            fprintf(stderr, "client: errno = %d\n", err);
            exit(EXIT_FAILURE);
      }
```

```
/* set to blocking so connect hangs, waiting for connect */
arg = 0;
       i = ioctl(sd,FIOSNBIO,&arg);

client.sun_family = AF_UNIX;
memcpy(client.sun_path,address,2);

/* connect to the socket */
if (connect(sd, (struct sockaddr *)&client, sizeof(client))) {
      err = errno;
      fprintf(stderr, "client: can't connect socket\n");
      fprintf(stderr, "client: errno = %d\n", err);
      close(sd);
      exit(EXIT_FAILURE);
}
printf("connected socket fd = %d\n",sd);

arg = 1;
       i = ioctl(sd,FIOSNBIO,&arg);

printf("entering command loop\n");
command = 'a';

while (command != 'q') {
      printf("client> ");
      scanf("%s",line);
      sscanf(line,"%c",&command);

      switch (command) {
      case 's' :
            printf("message to send: ");
            scanf("%s",buf);
            i = write(sd,buf,strlen(buf));
            err = errno;
            if (i < 0)
                  printf("write() returned errno %d\n", err);
            else printf("sent >%s< (%d bytes) to server\n",
                        buf,strlen(buf));
            break;

      case '?' :
            printf("commands:\n");
            printf("   s - send a message\n");
            printf("   ? - this help messge\n");
            printf("   c - check the socket\n");
            printf("   r - read from socket\n");
            printf("   b - set to blocking I/O\n");
            printf("   n - set to non-blocking I/O\n");
            printf("   q - close socket and quit\n");
            break;

      case 'b' :
              arg = 0;
            ioctl(sd,FIOSNBIO,&arg);
            printf("I/O is blocking\n");
            break;

      case 'n' :
              arg = 1;
            ioctl(sd,FIOSNBIO,&arg);
            printf("I/O is non-blocking\n");
            break;
```

```
            case 'c' :
                    /* setup query fields */
                    rdfds[0] = 1 << sd; rdfds[1] = 0;
                  wrtfds[0] = 1 << sd; wrtfds[1] = 0;
                  timeout.tv_sec = 0; timeout.tv_usec = 0;
                  i = select(sd+1,rdfds,wrtfds,(int *)NULL,
                           &timeout);
                  err = errno;
                  if (i < 0)
                          printf("select() returned error %d\n",err);
                  else {
                        if (rdfds[0] & (1 << sd))
                              printf("socket has data to be read\n");
                        if (wrtfds[0] & (1 << sd))
                              printf("data can be written to socket\n");
                    }
                  break;

            case 'r' :
                  printf("number of bytes to read > ");
                  scanf("%d",&i);
                  if (i > sizeof(buf)) i = sizeof(buf);
                    memset(&buf[0],0, sizeof(buf));
                  j = read(sd,buf,i);
                  err = errno;
                  if (j < 0)
                        printf("read() returned errno %d\n",err);
                  else
                        printf("got %d bytes, msg is >%s<\n",j,buf);
                  break;

            case 'q' :
                  close(sd);
                  break;

            default:
                  printf("\n");
                  break;
            }
      }
      exit(EXIT_SUCCESS);
}
```

See Also

device driver, hosts, hosts.equiv, hosts.lpd, inetd.conf, libraries, msgget(), named pipes, networks, pipe(), protocols, semget(), services, shmget(), STREAMS

Notes

The version of sockets included with COHERENT is not built into the kernel. Rather, it uses a library of routines that use named pipes to emulate sockets. You should not invoke the system calls **read()** or **write()** to manipulate directly any descriptor returned by a call to **socket()**, because this descriptor defines only one of a set of named pipes required to mimic a true kernel-level socket. Header file **<sys/socket.h>** replaces these with the macros that perform the task correctly. This means that in every C file where you perform a **read()**, **write()**, **ioctl()**, or **close()** on a socket, you must include **<sys/socket.h>**.

This library was adapted from Berkeley sources by P.Garbha (pgd@compuram.bbt.se), and was extensively revised by Mark Williams Company.

This product includes software developed by the University of California, Berkeley, and its contributors.

libterm — Library

Functions to read termcap descriptions

The library **libterm** holds the following routines, with which a program can read a **termcap** description:

tgetent() Read a **termcap** entry.

tgetflag() Check if a given Boolean capability is present in the terminal's **termcap** entry.

tgetnum() Return the value of a numeric **termcap** feature (e.g., the number of columns on the terminal).

tgetstr() Read and decode a **termcap** string feature.

tgoto() Read and decode a cursor-addressing string.

tputs() Read and decode the leading padding information of a **termcap** string feature.

See the Lexicon entry for each function for details.

See Also

libcurses, libraries, termcap

limits.h — Header File

Define numerical limits
#include <limits.h>

The header file **<limits.h>** defines macros that set the numerical limits for the translation environment.

The following table gives the macros defined in **limits.h**. Each value given is the macro's minimum maximum: a conforming implementation of C must meet these limits, and may exceed them.

CHAR_BIT
Number of bits in a **char**. This must be at least eight.

CHAR_MAX
Largest value representable in an object of type **char**. If the implementation defines a **char** to be signed, then it is equal to the value of the macro **SCHAR_MAX**; otherwise, it is equal to the value of the macro **UCHAR_MAX**.

CHAR_MIN
Smallest value representable in an object of type **char**. If the implementation defines a **char** to be signed, then it is equal to the value of the macro **SCHAR_MIN**; otherwise, it is zero.

INT_MAX
Largest value representable in an object of type **int**; it must be at least 32,767 (0x7FFF).

INT_MIN
Smallest value representable in an object of type **int**; no less more -32,767.

LONG_MAX
Largest value representable in an object of type **long int**; it must be at least 2,147,483,647 (0x7FFFFFFFL).

LONG_MIN
Smallest value representable in an object of type **long int**; it must be at most -2,147,483,647.

MB_LEN_MAX
Largest number of bytes in any multibyte character, for any locale; it must be at least one.

OPEN_MAX
The maximum number of file descriptors that a process can hold at any given time.

Please note that this constant gives a "snapshot" of the state of COHERENT at this time. Using this constant in a program, in particular to size an array, greatly decreases the portability of a program, and may cause it to behave incorrectly. To determine the number of file descriptors that the operating system permits right now, use the system call **sysconf()**. *Caveat utilitor!*

SCHAR_MAX
Largest value representable in an object of type **signed char**; it must be at least 127.

SCHAR_MIN
Smallest value representable in an object of type **signed char**; it must be at most -127.

SHRT_MAX
Largest value representable in an object of type **short int**; it must be at least 32,767 ((**short**)0x7FFF).

SHRT_MIN
Smallest value representable in an object of type **short int**; it must be at most -32,767.

UCHAR_MAX
Largest value representable in an object of type **unsigned char**; it must be at least 255.

UINT_MAX
Largest value representable in an object of type **unsigned int**; it must be at least 65,535 ((**unsigned int**)0xFFFF).

ULONG_MAX
Largest value representable in an object of type **unsigned long int**; it must be at least 4,294,967,295 ((**unsigned long**)0xFFFFFFFFL).

USHRT_MAX
Largest value representable in an object of type **unsigned short int**; it must be at least 65,535 ((**unsigned short**)0xFFFF).

See Also

header files
ANSI Standard, §5.2.4.2.1
POSIX Standard, §2.8

Notes

limits.h sets fixed limits. If a limit is not completely fixed, then the symbol is not defined, and a process must use **sysconf()** or **pathconf()**, as appropriate, to find the limit's value for the current run of the process.

lines — Command

Highly amusing board game
/usr/games/lines

lines is an interactive COHERENT version of a two-player board game by Claude Soucie called *Lines of Action*. The screen displays the game board with "X" and "O" characters marking the positions of the pieces. To see the rules of the game, type "r" and then press **<Enter>**. To see the available interactive commands, type "h" and press **<Enter>**.

Two players can use **lines** to keep track of a game between them by moving with the "M" command. Alternatively, one player can play against the computer by moving with the "m" command. The program uses a tree-search technique to consider possible moves; the player can vary the speed of the program's replies with commands that change the tree search width and depth.

For a more detailed description of *Lines of Action*, see *A Gamut of Games* by Sid Sackson (New York, Random House, 1969).

See Also

commands

link() — System Call (libc)

Create a link
#include <unistd.h>
link(*old, new***)**
char **old, *new***;**

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**.

Examples

The first example, called **lock.c**, demonstrates how **link()** can be used to perform intertask locking. With this technique, a program can start a process in the background and stop any other user from starting the identical process.

```
#include <unistd.h>
main()
{
      if(link("lock.c", "lockfile") == -1) {
            printf("Cannot link\n");
            exit(1);
      }

      sleep(50);  /* do nothing for 50 seconds */
      unlink("lockfile");
      printf("done\n");
      exit(EXIT_SUCCESS);
}
```

The second example demonstrates how to use **link()** and **unlink()** to rename a file.

```
#include <stdio.h>
#include <unistd.h>
main(argc, argv) int argc; char *argv[];
{
      register char *old, *new;

      if (argc != 3) {
            fprintf(stderr, "Usage: rename old new\n");
            exit(EXIT_FAILURE);
      }
      old = argv[1];
      new = argv[2];

      if (link(old, new) == -1) {
            fprintf(stderr, "rename: link(%s, %s) failed\n", old, new);
            exit(EXIT_FAILURE);
      } else if (unlink(old) == -1) {
            fprintf(stderr, "rename: unlink(%s) failed\n", old);
            exit(EXIT_FAILURE);
      }
      exit(EXIT_SUCCESS);
}
```

See Also

find, libc, ln, rename(), unlink(), unistd.h
POSIX Standard, §5.3.4

Diagnostics

link() returns zero when successful. It returns -1 on errors, e.g., *old* does not exist, *new* already exists, attempt to link across file systems, or no permission to create *new* in the target directory.

Notes

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

listen() — Sockets Function (libsocket)

Listen for a connection on a socket
#include <sys/socket.h>
int listen(*socket***,** *backlog***)**
int *socket*, **int** *backlog*;

Function **listen()** "listens" for a connection on *socket*. It also signals the system your process's willingness to accept a connection on that socket. This function applies only to sockets of type **SOCK_STREAM** or **SOCK_SEQPACKET**.

socket is a file descriptor that identifies the socket in question. It must have been returned by a call to **socket()**. *backlog* defines the maximum length to which the queue of pending connections may grow. As of this writing, *backlog* is limited to a maximum of five. If a connection request arrives with the queue full, the client may receive an error with an indication of **ECONNREFUSED**; or if the underlying protocol supports retransmission, the request may be ignored so that retries may succeed.

If all goes well, **listen()** returns zero. If an error occurs, it returns -1 and sets **errno** to an appropriate value. The following lists the errors that can occur, by the value to which **listen()** sets **errno**:

EBADF *socket* is not a valid descriptor.

ENOTSOCK
socket does not identify a socket.

EOPNOTSUPP
socket is not of a type that supports **listen()**.

Example

For an example of this function, see the Lexicon entry for **libsocket**.

See Also

accept(), connect(), libsocket, socket()

lmail — Command

Deliver mail on your local system

Command **lmail** delivers mail on your local system. It receives messages from the mail-routing program **smail**, and copies each into the appropriate user's mailbox.

See Also

commands, mail (overview), mail (command), rmail, smail

ln — Command

Create a link to a file
ln [-f] *oldfile newfile*
ln [-f] *oldfile ... directory*

The COHERENT system knows a file by its i-node number. Each file is also *linked* to one or more file names, each name being stored in a directory. This system means that the same file can be known by multiple names in multiple directories. The command **ln** lets you create a new link to a file.

In its first form, **ln** links the name *newfile* to the file that is already named *oldfile*, provided that *newfile* does not already exist.

In the second form, **ln** links *oldfile* with an identical name in another *directory*. In effect, one file will "live" in two directories.

If *newfile* already exists, **-f** forces **ln** to unlink it and assign its name to *oldfile*.

See Also

commands, cp, ls, mv, rm

Notes

Links across file systems are impossible. For example, if your COHERENT system has two file systems, one mounted on **/f** and the other mounted on **/usr**, you cannot use **ln** to link a file in **/v** to one in **/usr**.

Note, too, that **ln** lets you link a directory to another file. This feature is permitted by POSIX Standard; however, because COHERENT does not yet support symbolic links, this feature at best is useless, and at worst is rather dangerous. *Caveat utilitor.*

localtime() — Time Function (libc)

Convert system time to calendar structure
#include <time.h>
#include <sys/types.h>
struct tm ***localtime(***timep**)**
time_t **timep**;

localtime() converts COHERENT's internal time into the form described in the structure **tm**, which is defined in the header file **<time.h>**.

timep points to the system time. It is of type **time_t**, which is defined in the header file **<sys/types.h>**.

localtime() returns a pointer to the structure **tm**. The function **asctime()** turns **tm** into an ASCII string.

Unlike its cousin **gmtime()**, **localtime()** returns the local time, including conversion to daylight saving time, if applicable. The daylight-saving time flag indicates whether daylight saving time is now in effect, *not* whether it is in effect during some part of the year. Note, too, that the time zone is set by **localtime()** every time the value returned by

```
getenv("TIMEZONE")
```

changes. See the Lexicon entry for **TIMEZONE** for more information on how COHERENT handles time zone settings.

Example

The following example recreates the function **asctime()**. It builds a string somewhat different from that returned by **asctime()** to demonstrate how to manipulate the **tm** structure.

```
#include <time.h>
#include <sys/types.h>

char *month[] = {
      "January", "February", "March", "April",
      "May", "June", "July", "August", "September",
      "October", "November", "December"
};

char *weekday[] = {
      "Sunday", "Monday", "Tuesday", "Wednesday",
      "Thursday", "Friday", "Saturday"
};

main()
{
      char buf[20];
      time_t tnum;
      struct tm *ts;
      int hour = 0;

      time(&tnum);      /* get time from system */

      /* convert time to tm struct */
      ts=localtime(&tnum);

      if (ts->tm_hour == 0)
            sprintf(buf,"12:%02d:%02d A.M.",
                  ts->tm_min, ts->tm_sec);

      else
            if(ts->tm_hour>=12) {
                  hour=ts->tm_hour-12;
                  if (hour==0)
                        hour=12;
                  sprintf(buf,"%02d:%02d:%02d P.M.",
                        hour, ts->tm_min,ts->tm_sec);

            } else
                  sprintf(buf,"%02d:%02d:%02d A.M.", ts->tm_hour,
                        ts->tm_min,ts->tm_sec);

      printf("\n%s %d %s 19%d %s\n",
            weekday[ts->tm_wday], ts->tm_mday,
            month[ts->tm_mon], ts->tm_year, buf);

      printf("Today is the %d day of 19%d\n",
            ts->tm_yday, ts->tm_year);

      printf("Daylight Saving Time %s in effect\n",
            ts->tm_isdst ? "is" : "is not");
}
```

See Also

gmtime(), libc, time [overview], TIMEZONE
ANSI Standard, §7.12.3.4
POSIX Standard, §8.1

LEXICON

Notes

localtime() returns a pointer to a statically allocated data area that is overwritten by successive calls.

lockf() — General Function (libc)

Lock a file or a section of a file
#include <unistd.h>
int
lockf(*fd***,** *cmd***,** *size***)**
int *fd*, *cmd*; **long** *size*;

The COHERENT library function **lockf()** allows a process to lock part or all of a file. If another process calls **lockf()** on the same file to request a lock that conflicts with a previous lock, the later **lockf()** call returns an error or sleeps until the file is unlocked by the first process.

fd gives a file descriptor of an open file; the file must have been opened with **O_WRONLY** or **O_RDWR** permission if **lockf()** is to succeed.

size specifies how many bytes should be locked or unlocked. The lock begins at the current file position and extend forward (if *size* is positive) or backwards (if it is negative). A *size* of zero locks or unlocks the entire file starting from the current position.

cmd specifies the action **lockf()** is to take. **lockf()** recognizes the following four commands, as specified in the header file **<unistd.h>**:

F_TEST Test whether a lock has already been set upon the specified section of the file.

F_LOCK Lock a section of the file, if possible. If the section cannot be locked, sleep until it becomes available for locking.

F_TLOCK Lock a section of the file, if possible. Unlike **F_LOCK, F_TLOCK** does not sleep if the section cannot be locked; rather, it returns -1 and sets **errno** to **EAGAIN** if the lock is not available.

F_ULOCK Unlock a currently existing lock.

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

See Also

creat(), fcntl(), libc, open()

Diagnostics

lockf() returns zero on success, -1 on failure. On failure, it also sets **errno** to an appropriate value. Possible errors include the following:

EINVAL Invalid file descriptor.

EAGAIN Requested section is already locked.

EDEADLK A deadlock would occur if the command slept, or the system lock table is full.

Notes

See the Lexicon entry for **fcntl()** for a fuller description of the COHERENT system's method of file locking.

log() — Mathematics Function (libm)

Compute natural logarithm
#include <math.h>
double log(*z***) double** *z*;

log() returns the natural (base *e*) logarithm of its argument *z*.

Example

The following example is by Sanjay Lal (sanjayl@tor.comm.mot.com). It returns the amount of a quantity of radioactive material that remains after the passage of a period of time. Use it when planning your next nuclear dump. It takes three arguments: the amount of material, in kilograms; the half life, in years; and the time passed, in years. These can be decimal fractions.

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

main(argc, argv)
int argc; char *argv[];
{
      double num, thalf, time;

      if (argc != 4) {
            fprintf(stderr,"Usage: %s amount halflife time\n", argv[0]);
            exit (EXIT_FAILURE);
      }

      num = atof (argv[1]);
      thalf = atof (argv[2]);
      time = atof (argv[3]);
      printf("%f\n", num * exp ( -log(2.0) * (time / thalf)));
}
```

See Also

log10(), libm
ANSI Standard, §7.5.4.4
POSIX Standard, §8.1

Diagnostics

When a domain error occurs (*z* is less than or equal to zero), **log()** sets **errno** to **EDOM** and returns zero.

log10() — Mathematics Function (libm)

Compute common logarithm
#include <math.h>
double log10(*z***) double** *z***;**

log10() returns the common (base 10) logarithm of its argument *z*.

Example

The following example, called **fact.c**, uses **log10()** and **pow()** to compute an approximation of the factorial of an integer. Compile it with the command:

```
cc -f fact.c -lm
```

It is by Brent Seidel (brent_seidel@chthone.stat.com).

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

main()
{
      int num, loop, exponent;
      double sum, fraction;
      char buffer[50];

      fprintf(stderr, "Enter number to compute factoral of: ");
      fflush(stderr);
      if (gets(buffer) == NULL)
            exit(EXIT_FAILURE);

      num = atoi(buffer);
      for (sum = 0, loop = 2; loop <= num; loop++) {
            sum += log10((double) loop);
      }

      exponent = (int) sum;
      fraction = sum - exponent;
      printf("The factoral of %d is %g X 10^%d\n",
            num, pow(10.0, fraction), exponent);
}
```

See Also

log(), libm
ANSI Standard, §7.5.4.5
POSIX Standard, §8.1

Diagnostics

A domain error in **log10()** (*z* is less than or equal to zero) sets **errno** to **EDOM** and returns zero.

login — Command

Log in a user
login [-p] [*login_id* **[***environ_var***[=***value***] ...]]**

The command **login** allows a user to identify himself to your system. A user can invoke it as a command, or the system itself can invoke it (usually through the command **getty**) when a user attempts to log in.

You can invoke **login** as a command. To do so, return to your lowest-level (login) shell, then type either

```
login
```

or:

```
exec /bin/login
```

This invocation replaces the shell with **login**, and so ensures a smooth transition from one user account to another.

If the user does not supply a *login_id* on the **login** command line, **login** prompts him for the login identifier to use. If the account for *login_id* is protected by a password, **login** then asks the user to enter that password. If possible, **login** turns off echoing during the entry of the password to ensure that bystanders (or "kibitzers") cannot see the password displayed on his terminal.

Switches

login executes the file **/etc/default/login**. This file sets switches that control **login**'s behavior. Each switch has the form

*SWITCH***=***VALUE*

where *SWITCH* is the switch being set and *VALUE* is the value to which it is being set. **login** exports some of these switches as environmental variables, to give the programs that **login** invokes a minimal working environment.

login recognizes the following switches by default:

ALTSHELL
If set to **YES**, the login shell's name is recorded in the environment. If set to **NO**, it is not. By default, **login** sets this to **YES**.

CONSOLE
The allowable terminal devices (from **/dev**) from which the superuser **root** can log into your system. If this names more than one device, you must separated them with colons. If this variable is not set, then **root** can log in from any device. A device name can also include the wildcard character '?'.

HZ Your computer's clock tick frequency, in Hertz. **login** does not set a default. **login** exports this switch as an environmental variable.

IDLEWEEKS
The number of weeks before a login is disabled for lack of use. **login** does not set this variable.

NEWUSER
This switch gives a shell command that is to be executed when the file **$HOME/.lastlogin** does not exist. By default, it displays a warning message is displayed. The installation script for COHERENT typically creates a setting for you that executes the file **/etc/default/welcome** instead. This works with the command **/etc/newusr** to provide a "friendly" environment for users who are using COHERENT for the first time.

PASSREQ
If set to **YES**, every user must have a password. If set to **NO**, some users may log in without a password. By default **login** sets this to **YES**.

PATH This variable names the directories that an interactive shell searches for executable files. By default, **login** sets this to **/bin:/usr/bin**. **login** exports this switch as an environmental variable.

SUPATH
The default path for the superuser **root**. By default, **login** sets this to **/bin:/usr/bin**. **login** exports this switch as an environmental variable.

TIMEOUT
The time, in seconds, that **login** waits before it silently terminates and returns control to **getty**. **login** gives the user five "chances" to log in during this time. **login** by default sets this to 120.

TIMEZONE
The current time zone. This variable has the same format as the COHERENT environmental variable **TZ**: that is, it uses the template *NSTHNDT*, where *NST* is a three-character abbreviation for your local standard time (e.g., **CST** for Central Standard Time), *H* gives the number of hours difference between your time zone and Greenwich Mean Time, and *NSD* gives a three-character abbreviation for your local daylight-saving time. **login** exports this switch as an environmental variable.

Note that this variable is set for the benefit of code imported from UNIX. Most COHERENT commands use the environmental variable **TIMEZONE**, which much more detailed information about your local time zone. For details on **TIMEZONE**, see its entry in the Lexicon.

Note, too, that the variable **TZ**, which is set in file **/etc/timezone**, should be set to exactly the same string as **/etc/default/TIMEZONE**; otherwise, much confusion will result.

ULIMIT
The maximum size, in 512-byte blocks, of a file that the user can create. **login** does not set a default. At present, COHERENT ignores this option.

UMASK
This gives the permissions that a shell sets by default for files that the user cretaes. **login** does not set a default value for this variable. **login** exports this switch as an environmental variable.

Logging Failed Attempts

If the user attempts and fails five times to log in, **login** records the erroneous attempts in file **/usr/adm/loginlog** (should that file exist), and it disables the terminal for a period of time. (Note that previous versions of COHERENT recorded failed attempts in file **/usr/adm/failed**.) **login** does not record when the user typed only `(¢)` in response to a prompt for a login identifier. If the user does not succeed in logging in within two minutes (120 seconds), **login** silently disconnects the terminal and returns control of the device to **getty**.

Restrictions on Logging In

If the file **/etc/nologin** exists, **login** refuses to let any users login in, except for the superuser **root** and the (presumably few) users named in file **/etc/trustme**. You can use this mechanism to stop users from logging in at an inopportune time, e.g., when the system is about to be shut down. In response to an attempt to log in, **login** displays the contents of that file, which should contain the system administrator's explanation of why logins are not permitted at that time.

login also reads file **/etc/usrtime**, if it exists. This file gives user identifiers; for each identifier, it gives the tty line from which that user can log in, and the day of the week and time of day during which that user can log in. **login** rejects the user's login if it is from a tty line forbidden to the user, or outside the day and time permitted. If a user's login identifier is not in this file, **login** assumes that that user can log in from any line and at any time. Additional options allow you to control globally all users, or interactive users, UUCP accounts, or SLIP users.

Passwords

login prompts the user for a password when he logs in. **login** takes its copy of the user's password from file **/etc/passwd**. If the password consists of a single asterisk '*', then **login** reads the password from file **/etc/shadow**. This file should be legible only by the superuser **root**. Once the passwords are in **/etc/shadow**, they can be read only by processes that have **root**-level permissions, such as **login**. This protects the encrypted passwords from being read by ordinary users, and perhaps decrypted by a "cracker." For details, see the Lexicon entry for **shadow**.

Note that if a user's password consists of '*' and file **/etc/shadow** does not exist, **login** assumes that the user's password encrypts to '*'. This effectively locks the user out of his account. The lesson is not to remove or modify **/etc/shadow** capriciously.

LEXICON

In addition, **login** reads the files **/etc/dialups** and **/etc/d_passwd**, which hold auxiliary passwords. You can set auxiliary passwords for users on selected tty lines to provide additional security. For details, see these files' entries in the Lexicon.

Success In Logging In

If the user succeeds in logging in, **login** displays on his terminal the date and time that he last logged in, as recorded in file **$HOME/.lastlogin**. **login** updates this file whenever the user logs in. If this file had been modified by a process other than **login**, **login** warns the user of a possible breach in his account's security.

login then prints the contents of the file **/etc/motd**, which holds the message of the day. It also sets the environmental variable **LOGNAME** to the user's login identifier.

As its last action, **login** invokes the user's shell, as set in the last field of his entry in **/etc/passwd**. Under COHERENT, this is either the Bourne shell **sh** or the Korn shell **ksh**. (**login** can also invoke a program in place of a shell, e.g., the command **uucico** for a UUCP account.) If **login** invokes an interactive shell, it does so with the first character of its **argv[0]** set to '-', so that the shell knows that it is a login shell. (For example, if **login** invokes **ksh**, its **argv[0]** is **-ksh**.)

When a shell starts up, it executes the script **/etc/profile**. This script executes the command **umask**, to set the permissions that the shell gives by default to files that that user creates; and then sets the following environmental variables:

HZ The default clock speed for your system. By default, COHERENT sets this to 100.

LOGNAME
The user's login identifier.

MAIL This names the user's mailbox. By default, it is set to **/usr/spool/mail/***login_id*.

PAGER The command used to "page" through files of text. By default, COHERENT sets this to **more**.

PATH The directories that the shell searches for executable files. By default, COHERENT sets these to **/bin** and **/usr/bin**.

TERM The type of terminal at which the user is working. By default, COHERENT reads file **/etc/ttytype** to read the default terminal type for a given port. For details, see the description of this command in the Lexicon.

Finally, **/etc/profile** calls the script **/etc/timezone**, which sets the following environmental variables:

TZ Your time zone, as interpreted by most UNIX software.

TIMEZONE
Your time zone, as interpreted by the COHERENT system. At present, it contains considerably more information about your time zone than does **TZ**. For details of this variable, see its description in the Lexicon.

The shell then executes the script **$HOME/.profile**, should one exist. The COHERENT command **newusr** creates this file when it installs a new user. The user can edit this file to set environmental variables, and to invoke commands for his amusement, e.g., **/usr/games/fortune**.

Command-line Options

If a user invokes **login** as a command, he can set one or more environmental variables on **login**'s command line. If *environ_var* contains an equal sign, then it and *value* are placed into the environment. If *environ_var* does not contain an equal sign, then **login** places it into the environment with the format:

environ_var=*n*

where *n* is a number from zero through the number of environmental variables being so set.

For security reasons, **login** refuses to set from its command line any of the following environmental variables:

CDPATH	**HOME**
HZ	**IFS**
LOGNAME	**MAIL**
PATH	**SHELL**
TZ	

login also recognizes the command-line option **-p**, which tells **login** to preserve the user's current environment when logging in as *login_id*. If it is *not* invoked with this option, **login** "empties" the current user's before it constructs the environment for user *login_id*. If it is invoked with this option **login** replaces existing environmental variables with those it sets during the login process, but it preserves all other environmental variables set in the

original environment.

Subsystem Logins

login supports virtual "subsystems" under COHERENT. If the user's shell as specified in **/etc/passwd** is '*', then **login** makes the user's **HOME** directory into the system's root directory, informs the user that it is executing a "Subsystem login," and then re-executes **login**. The new root directory must have its own versions of the commands **/etc/passwd**, **/bin/login**, and **/dev** files. Once so logged in, the user has, in effect, his own virtual version of the COHERENT system.

Files

/etc/d_passwd — Passwords for shells on dialup lines
/etc/default/login — Default parameters for **login**
/etc/dialups — List of dialup tty lines
/etc/group — File that defines user groups
/etc/nologin — Forbid all logins
/etc/passwd — Password file
/etc/profile — Script executed by **sh** and **ksh** upon invocation
/etc/shadow — Optional file of "shadow" passwords
/etc/trustme — Permit named users to log in despite **nologin**
/etc/ttytype — Default terminal type on a given tty line
/etc/utmp — Identifiers of users who are logged into your system
/etc/usrtime — Login restrictions for user *login_id*
/etc/wtmp — History of who has logged in, and when
/usr/adm/loginlog — Record of failed login attempts
/usr/spool/mail/*name* — Mailbox for *user*
$HOME/.lastlogin — Date of user's last login

See Also

Administering COHERENT, commands, ksh, lastlogin, mail, sh, newgrp, newusr, welcome

Notes

This version of **login** no longer recognizes the remote-access account **remacc**. To duplicate the function of this account, set the files **/etc/dialups** and **/etc/d_passwd**. For details, see their entries in the Lexicon.

This version of **login** was written by Tony Field (tony@ajfcal.cuc.ab.ca), with help from Uwe Doering (gemini@geminix.in-berlin.de). It was ported to COHERENT by Harry Pulley (hcpiv@snowhite.cis.uoguelph.ca), with help from Udo Munk (udo@umunk.gun.de).

login — System Administration

Set default values for logging in
/etc/default/login

The command **login** reads the file **/etc/default/login**, which gives **login**'s default settings. These settings dictate some of **login**'s behaviors. **login** exports some settings as environmental variables, to help control the behavior of some other programs within COHERENT.

For a list of the settings normally set by **/etc/default/login**, see the Lexicon entry for the command **login**.

See Also

Administering COHERENT, login

loginlog — System Administration

Log of failed login attempts
/usr/adm/loginlog

File **/usr/adm/loginlog** logs all failed attempts to log in. **login** places an entry into this log either when a login attempt does not succeed within the number of seconds set by the environmental variable **TIMEOUT** (default, 120), or fails to log in correctly with five times within that time.

See Also

Administering COHERENT, login

Notes

Earlier implementations of **login** logged failed attempts in file **/usr/adm/failed**.

logmsg — System Administration

Hold COHERENT Login Message
/etc/logmsg

The file **/etc/logmsg** holds the message that COHERENT displays to prompt the user to log in. The superuser **bin** can edit this message to whatever she prefers.

See Also

Administering COHERENT

Notes

The default message consists of the bell character **<ctrl-G>** followed by the text **Coherent login:**. If the bell annoys you, simply delete the **<ctrl-G>** from **/etc/logmsg**.

LOGNAME — Environmental Variable

Name user's identifier
LOGNAME=*user_identifier*

The environmental variable **LOGNAME** names your login identifier. For example, if your login identifier is **fwb**, then by typing **set** you will see the entry **LOGNAME=fwb**. **LOGNAME** is set in **/etc/profile**.

See Also

environmental variables, ksh, login, sh, USER

long — C Keyword

Data type

A **long** is a numeric data type. The ANSI standard states that **long** is the largest integer data type. It cannot be smaller than an **int**, although an **int** and a **long** can be the same size.

COHERENT defines an **long** to be four bytes long; that is, **sizeof long** equals 4 (four **char**s, or 31 data bits plus a sign bit). A **long** can hold any value from -2,147,483,647 to 2,147,483,647.

A **long** normally is sign extended when cast to a larger data type; an **unsigned long**, however, will be zero extended.

See Also

C keywords, data formats, int
ANSI Standard, §6.1.2.5

longjmp() — General Function (libc)

Perform a non-local goto
#include <setjmp.h>
int longjmp(*env***,** *rval***)**
jmp_buf *env***; int** *rval***;**

The function call is the only mechanism that C provides to transfer control between functions. This mechanism is inadequate for some purposes, such as handling unexpected errors or interrupts at lower levels of a program. To answer this need, **longjmp** provides a non-local *goto*.

longjmp() restores an environment that had been saved by a previous call to the function **setjmp()**. It returns the value *rval* to the caller of **setjmp()**, just as if the **setjmp()** call had just returned. Note that **longjmp()** must not restore the environment of a routine that has already returned. The type declaration for **jmp_buf** is in the header file **setjmp.h**. The environment saved includes the program counter, stack pointer, and stack frame. These routines do not restore register variables in the environment returned.

Example

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

See Also

libc, setjmp(), siglongjmp()
ANSI Standard, §7.6.2.1
POSIX Standard, §8.1

Notes

Programmers should note that many user-level routines cannot be interrupted and reentered safely. For that reason, improper use of **longjmp()** and **setjmp()** can result in the creation of mysterious and irreproducible bugs. Do not attempt to use **longjmp()** within an exception handler.

look — Command

Find matching lines in a sorted file
look [-df] *string* [*file*]

The command **look** scans the sorted *file* and prints each line that begins with *string*.

The following options specify the order of the search:

-d Use dictionary order: the only characters tested are alphanumerics and blanks.

-f Convert all alphabetic characters to upper case.

If no *file* is specified, **look** uses **/usr/dict/words** with the **-df** option.

Example

For an example of how to use this command, see the entry for **spell**.

Files

/usr/dict/words — File of words (sorted with **sort -df**).

See Also

commands, sort

Notes

Because the file **/usr/dict/words** is quite large, you may not have installed it or uncompressed it when you installed your COHERENT system. If this is the case, **look** will not work correctly.

lp — Command

Spool a file for printing
lp [-d*printer***] [-t** *title***] [-n***copies***] [-R** *page* **[***page***]] [-Smws]** *file ...*

The command **lp** spools text for printing. If you name no *file* on its command line, **lp** spools what it receives from the standard input.

lp prefaces the spooled text with a header that describes, among other things, on what device you want to print the text; then it copies the text into directory **/usr/spool/mlp/queue**, where it remains until it is removed by the printer daemon **lpsched**. The spooled text, which may comprise multiple files, plus its header is called a *job*.

The following describes the header with which **lp** prefaces each file:

Offset	*Length*	*Description*
0	14	User who spooled the file
14	14	Name of the printer on which to print file
28	10	Type of file (application specific)
38	3	Length of output page (default, 66 lines)
41	4	Number of pages (maximum, 9,999)
45	2	Number of copies to print (default, one; maximum, 99)
47	1	Set life expectancy of job (see below)
48	1	If 'M', send user mail after printing
49	1	If 'W', write user after printing
50	14	Name of data base (application specific)
64	14	Name of program (application specific)
78	10	Date/time stamp (no. of seconds since 1/1/1970)
88	60	Description or title

Note that the fields marked "application specific" are not use by **lp** or **lpsched**. Rather, they are available to applications, such as filters, that may be used with **lp** to print files.

The "life expectancy" byte of the header defines how long the job remains alive in **/usr/spool/mlp/queue**. Jobs labeled **T** (temporary) live for 30 minutes after being spooled; those labeled **S** (short-term) live for 24 hours; and those labeled **L** (long-term) live for 72 hours. Once a job's life expectancy has expired, the printer daemon **lpsched** removes it. The default is life expectancy is **S**. To change the life expectancy of a job, use the command **chreq**. You can also change the above default "lifetimes" by editing the file **/usr/spool/mlp/controls**.

When **lp** creates a job, it gives the job a seven-character name. The name's first character gives the status of the job: **R** indicates that the file is being printed or is pending printing, whereas **r** indicates that the job has already been printed. The second character gives the job's priority status, from 0 through 9: zero gives highest priority, nine the lowest. The default priority is 2. The last five characters of the name give a zero-padded sequence number. To change a job's status or priority, use the command **chreq**; or the system administrator can alter either simply by renaming the file.

lp recognizes the following options:

-R *request* Print a job beginning from the first *page* and continuing either to the second *page* or to the end of the document (if no second *page* is specified). Note that the printer daemon **lpsched** identifies pages by counting lines of input, so this feature works only with straight text. It does *not* work correctly with "cooked" input, such as files of PostScript or PCL.

-S9 Shut down the spooler daemon **lpsched**.

-d*printer* Print the job on *printer*.

-m Send mail to the user once the spooled job has been printed.

-n*copies* Print *copies* copies of the job.

-s Silent — do not acknowlege submissions. Normally, **lp** writes on the standard output the sequence number of the job you just spooled. You can use that number to remove or abort a job, or otherwise manipulate it.

-t *title* Give this job *title*. This is the title that appears in the queue displayed by the command **lpstat**.

-w Write a message on the user's screen once the job has been printed.

lp sends you mail if one of your print jobs failed due to an error.

For more information on **lp** and its related commands, see the Lexicon entry **printer**.

See Also

chreq, commands, controls, lp [device driver], lpadmin, lpsched, pclfont, printer

Notes

Because most users find banners annoying rather than helpful, **lp** does not print banners. It ignores the option **-b**, which under orthodox implementations of **lp** prints a banner page. Applications that desire a banner page should make provision for it in the individual printer's control file. For details, see the Lexicon entry for the command **lpadmin**.

If you wish to use **lp** to download a PCL bitmapped font to your PCL printer, you must first process the font with the command **pclfont**. For details, see its Lexicon entry.

lp — Device Driver

Driver for parallel ports
/dev/lpt*N*

The device driver **lp** drives the parallel ports. It has major number 3.

This driver follows the IBM PC standard in that it can only send data out the port — it cannot receive data from the port.

The following script lets you install or de-install the parallel-port driver: To install or de-install a parallel printer, log in as the superuser **root**; then execute the following script:

```
cd /etc/conf
lp/mkdev
/conf/mlpconfig
bin/idmkcoh -o /kernel_name
```

kernel_name should name the new kernel to build. Then reboot to invoke the newly built *kernel_name*.

See Also

device drivers, printer

lpadmin — Command

Administer the lp print-spooler system
lpadmin [-d*printer***]**
lpadmin [-p*printer***] [-v***device***] [-m***backend***]**
lpadmin [-x*printer***]**

The command **lpadmin** administers the **lp** print-spooling system.

Under the **lp** spooler, the system administrator gives each printer a name. She also establishes a script for each class of printers; for example, she would prepare one script for all Epson printers, and another for all PostScript printers. The script lists commands that must be executed to print the text properly, such as setting the port into the correct mode or post-processing the text; Finally, she inserts into file **/usr/spool/mlp/controls** an entry that links a printer by name with its device and its script. When a user spools a job for printing, she selects the printer by name. (She can also use the command **route** establish a default printer for herself.) The print spooler **lpsched** reads the information established by the administrator to ensure that printing is managed correctly.

The command **lpadmin** is designed to make it easy for you to perform these tasks of administration. With **lpadmin**, you can add a new printer to your COHERENT system and link it to a device and a description script. You can also add or modify a description script, or drop a printer. **lpadmin** recognizes the following options:

-d*printer* Make *printer* the default printer for your system. This is the printer that is used when a user names no printer on the **lp** command line and has set no default printer for herself.

-m*script* Use *script* to preprocess all text sent to a given printer. *script* is stored in directory **/usr/spool/mlp/backend**. This option is always used with option **-p**.

-p*printer* Select *printer* for definition or change. This option is used with the options **-m** and **-v**.

-v*device* Associate *device* (a serial or parallel port) with the printer named in the option **-p**.

-x*printer* Remove *printer* from the system.

For detailed examples of how to modify the file **controls** and how to build a control script for a printer, see the Lexicon entry for **controls**.

See Also

commands, controls, lp, printer

lpd — System Administration

Spooler daemon for line printer
/usr/lib/lpd

lpd is the daemon that prints listings queued by the command **lpr**. All jobs are printed on the printer that is accessed through device **/dev/lp**. For information on this device, and on printer management in general, see the Lexicon entry **printer**.

lpr invokes **lpd** automatically. If there is no printing to do, or if another daemon is already running (indicated by the file **dpid**), **lpd** exits immediately. Otherwise, it searches the spool directory for control files of listings to print. A control file contains 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. Priority is not given to any file, either by size or by requester.

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

Administering COHERENT, init, lpr, lpskip, printer

Notes

Beginning with release 4.2, COHERENT also includes the printer daemon **despooler**, which prints files spooled with the command **lp**. For details on how COHERENT manages printing, see the Lexicon entry for **printer**.

lpioctl.h — Header File

Definitions for line-printer I/O control
#include <sys/lpioctl.h>

lpioctl.h defines constants used by routines that control I/O on the line printer.

See Also

header files

lpr — Command

Spool a job for printing on the line printer
lpr [-cmnr] [-b *banner***] [***file ...***]**

The command **lpr** spools each *file* for printing on the line printer. If no *file* is named on the command line, **lpr** spools what it reads from the standard input.

lpr recognizes the following options:

-B Suppress printing of a banner.

-b *banner* Print *banner* on the banner page. The default banner is the user's login name.

-c Copy each *file* into the spooling directory, instead of reading the file from its home directory. This option lets you change a *file* before it has finished printing.

-m Write a message on the user's terminal when printing completes.

-n Do not send a message (default).

-r Remove the files when they have been spooled.

The command **lpskip** aborts or restarts printing of the file that is currently being printed. The command **epson** converts the output of **nroff** into a form usable by Epson-compatible dot-matrix printers.

Files

/dev/lp — Line printer
/usr/lib/lpd — Line printer daemon
/usr/spool/lpd — Spool directory
/usr/spool/lpd/dpid — Daemon lockfile

See Also

commands, hpr, lp, lpd, lpskip, printer

Notes

Beginning with release 4.2, COHERENT also includes the **lp** print spooler. **lp** offers a more sophisticated way to manage printers, especially on machines that support multiple printers of the same type. For details, see the Lexicon entries for **printer** and **lp**.

lpsched — Command

Print jobs spooled with command lp; turn on printer daemon
lpsched

The daemon **lpsched** prints jobs spooled with the command **lp**.

Typing the command **lpsched** by itself launches the daemon. The rest of this article describes how **lpsched**

manages printing.

Each file in directory **/usr/spooler/mlp/queue** is a print job spooled by the command **lp**. When **lp** spools a job, it copies the input (usually a file) into the spool directory and appends to the beginning of each job a 192-byte header that indicates how the job is to be printed. This header includes such information as the name of the printer on which to print the job and the date and time the job was spooled. For a detailed layout of this header, see the Lexicon entry for **lp**.

lp also assigned each job a seven-character name. The first character is **R** or **r**: the former indicates that the job is either being printed or is awaiting printing; whereas the latter indicates that the job has been printed. The second character is a digit, from zero to nine, that sets the job's priority: zero gives highest priority, nine lowest. (The default priority is **2**.) The last five characters give a zero-padded identification number.

lpsched awakens every 30 seconds or whenever the command **lp** spools a job for printing. **lpsched** then processes each file in **/usr/spooler/mlp/queue**. It reads each job each that is awaiting printing, the order being dictated first by the priority code and then by the identification number (which indicates the order in which the jobs were spooled).

When **lpsched** actually prints a job, it performs the following tasks:

- First, it opens the file that contains the job to be printed, and reads its header. The header gives the number of the job; the name of the user who spooled the job; the name of the printer device upon which the job is to be printed; the number of copies to print; and the title of the job, as set with the **lp** option **-t**. (NB, do not confuse an MLP "device," which is set in the file **/usr/spool/mlp/controls**, with the physical device into which the printer is plugged.)
- **lpsched** then finds the entry in file **/usr/spool/mlp/controls** that describes the printer device the user requested. An entry in **controls** is of the form

  ```
  printer = banner, /dev/hp, make_banner
  ```

  In this case, the MLP device is named **banner**; the output is to be printed on physical device **/dev/hp**; and the output is to be filtered through backend script **make_banner**, which is a script kept in directory **/usr/spool/mlp/backend**. (For details on how to describe an MLP printer device, see the Lexicon entry for **controls**).
- If the entry for this device does not name a backend script, **lpsched** copies the body of the job (that is, the text that you had spooled) without modification to the device by which the printer is accessed.
- If the entry for this device does name a backend script, **lpsched** invokes the script and redirects its output to the physical device.

When **lpsched** invokes a backend script, it passes it four arguments: (1) the number of the job to be printed, (2) the login identifier of the user who spooled the job, (3) the number of copies to be printed, and (4) the title of the job. The script can ignore these arguments, or use them in its filtration process; for example, it can use the fourth argument to construct a banner page that is printed before the job. For examples of backend script that perform various types of sophisticated processing, see the Lexicon entry for **controls**.

lpsched uses a system of lock files to ensure that each device is accessed in a disciplined manner. For details on COHERENT's system of building lock files, see the Lexicon entries for **UUCP** and **libmisc**.

To abort the printing of a job, invoke the command **cancel**. Note that this only affects jobs that are being spooled or waiting to be spooled. If a job has been downloaded to a printer, the only way to abort printing is to manipulate the printer itself through its front panel and switches.

When a job has printed successfully, **lpsched** changes the status character in its name to **r**. A file remains in the spool directory until its "lifetime" has expired. You can reprint a quiescent file by invoking the command **reprint**. To change a job's target printer, priority, or lifetime, use the command **chreq**. For details on these commands, see their Lexicon entries.

lpsched awakens whenever you use the command **lp** to spool a job for printing. It also awakens every five minutes, whether or not a job has been spooled, to see if anything needs to be printed and check quiescent files.

After it has processed every job that awaits printing, **lpsched** reads the header of every quiescent file. If a file's "lifetime" has expired, **lpsched** removes it. A file with a *temporary* lifetime survives 30 minutes after spooling; one with a *short-term* lifetime survives 24 hours; and one with a *long-term* lifetime survives 72 hours. You can change these defaults by editing **controls**; for details, see its entry in the Lexicon. By default, a job is given a short-term life expectancy. To change a job's life expectancy, use the command **chreq**.

LEXICON

The command **lpsched** turns on **lpsched**; and command **lpshut** turns it off. Jobs spooled while **lpsched** is turned off remain spooled until **lpsched** is reawakened.

See Also

Administering COHERENT, commands, lp, lpshut, printer

lpshut — Command

Turn off the printer daemon despooler
lpshut [-d]

The command **lpshut** turns off the printer daemon **lpsched**.

The option **-d** tells **lpshut** to finish the jobs that are currently printing before it shuts down the daemon. If jobs are interrupted while printing, the printer daemon **lpsched** reprints them when it restarts.

See Also

commands, lp, lpsched, printer

lpskip — Command

Abort/restart current job on line printer
lpskip [-r]

The command **lpskip** aborts or restarts the file being printed on the printer plugged into device **/dev/lp**. By default, it aborts the job and prints a message on the user's terminal.

When invoked with the **-r** option, **lpskip** restarts the printing of the current job. This is useful when a printing is spoiled due to, say, a paper jam.

lpskip works only with files that have been spooled with the command **lpr**.

Files

/usr/lib/lpd — Line printer daemon
/usr/spool/lpd — Spool directory
/usr/spool/lpd/dpid Daemon lockfile

See Also

commands, lpd, lpr, hpskip

Notes

To cancel jobs spooled with the command **hpr**, use the command **hpskip**. To cancel or reprint jobs spooled with the command **lp**, use the commands **cancel** and **reprint**. See the Lexicon entry **printer** for details.

lpstat — Command

Give status of printer or job
lpstat [-p*printer***] [-drqstv]**

The command **lpstat** gives information about the operation of the **lp** print-spooling mechanism. It recognizes the following options:

-p *printer*
Give the status of *printer*.

-d Name the system's default printer.

-r Give the status of the daemon **lpsched**.

-q Give a detailed report of jobs in the queue. The jobs are displayed in two groups, quiescent and active, with each group ordered by their priority — which, given **lp**'s conventions for naming jobs, is identical with their alphabetical order.

-s Summarize status of each request and status of each printer.

-t Like option **-s**, but in somewhat more detail.

-v List all available printers and the devices associated with them.

See Also

commands, lp, printer

lr — Command

List subdirectories' contents in columnar format
lr [*file ...***]**

lr is a link to the command **ls -CR**. It prints each *file* in columnar format, like the command **lc**. If a *file* is a directory, **lr** also prints its contents and that of each of its subdirectories. If no *file* is named, it lists the contents of the current directory by default.

See Also

commands, l, lc, lf, ls, lx

lrand48() — Random-Number Function (libc)

Return a 48-bit pseudo-random number as a non-negative long integer
long lrand48();

Function **lrand48()** generates a 48-bit random number, then returns its high 31 bits in the form of a non-negative **long**. The value returned is (or should be) uniformly distributed throughout the range of zero through 2^31.

See Also

libc, srand48()

ls — Command

List directory's contents
ls [-abCcdFfgilmnopqRrstux] [*file ...* **]**

The command **ls** prints information about each *file*. Normally, **ls** sorts its output by file name and prints only the name of each *file*. If a directory name is given as an argument, **ls** sorts and lists its contents, not including '.' and '..'. If no *file* is named, **ls** lists the contents of the current directory.

The following options control how **ls** sorts and displays its output:

-a Print all directory entries, including '.', '..', any hidden files, and volume ID's.

-b Print non-graphic characters in octal.

-C Print the output in multi-column format, sorted down the columns.

-c Print the time the files' attributes were last changed.

-d Treat directories as if they were files.

-F Print a '/' after the name of each directory, and print an '*' after each executable file.

-f Force each argument to be treated as a directory. This disables the **-lrst** options and sorting, and enables the **-a** option.

-i Print the i-number of each file.

-l Print information in long format. The fields give mode bits, link count, owner uid, owner gid, size in bytes, date, and file name. For special files, major and minor device numbers replace the size field.

-m "Stream" the output horizontally across the screen, with each file name separated by a comma.

-n Same as **-l**, except the group identifiers and user identifiers are numbers rather than names.

-o Same as **-l**, except that the group id is not printed.

-p Print a '/' after each directory name.

-q Print non-graphics characters as '?'.

-r Reverse the sense of the sort.

-R Recursively print directories.

-s Print the size in blocks of each file.

-t Sort by time, newest first.

-u Sort by the *access* time.

-x Print multicolumn output, sorted across the columns. This resembles the output of the command **lc**.

The date **ls** prints with the **-l** and **-t** options is the *modification* time, unless the **-c** or **-u** option is used as well.

The mode field in the long list format consists of ten characters. The first character will be one of the following:

- Regular file
b Block special file
c Character special file
d Directory
p Pipe
x Bad entry (remove it immediately!)

The remaining nine characters are permission bits, in three sets of three characters each. The first set pertains to the owner of the file, the second to users from the owner's group, and the third to users from other groups. Each set may contain three characters from the following.

r The file can be read
s Set effective user ID or group ID on execution
t Shared text is sticky
w The file can be written
x The file is executable
- No permission is given

Links

COHERENT includes several commands that are links to **ls** and its options, to make it easier for you to use the various features of **ls**. The following table gives each command and the form of **ls** to which it is linked:

l **ls -l**
lf **ls -CF**
lr **ls -CR**
lx **ls -x**

See Also

chmod, commands, l, lc, lf, lr, lx, stat

lseek() — System Call (libc)

Set read/write position
#include <unistd.h>
long lseek(*fd***,** *where***,** *how***)**
int *fd***,** *how***; long** *where***;**

lseek() changes the *seek position*, or the point within a file where the next read or write operation is performed. *fd* is the file's file descriptor, which is returned by **open()**.

where and *how* describe the new seek position. *where* gives the number of bytes that you wish to move the seek position. It is measured from the beginning of the file if *how* equals **SEEK_SET** (zero), from the current seek position if *how* equals **SEEK_CUR** (one), and from the end of the file if *how* equals **SEEK_END** (two). A successful call to **lseek()** returns the new seek position. For example,

```
position = lseek(fd, 100L, SEEK_SET);
```

moves the seek position 100 bytes past the beginning of the file; whereas

```
position = lseek(fd, 0L, SEEK_CUR);
```

returns the current seek position and does not change the seek position at all.

You can create a *sparse file* by seeking beyond the current size of the file and writing. The "hole" between the end of the file and where the write occurs is read as zero and will occupy no disk space. For example, if you **lseek()** 10,000 bytes past the current end of file and write a string, the data will be written 10,000 bytes past the old end of file and all intervening matter will be considered part of the file.

lseek() differs from its cousin **fseek()** in that **lseek()** is a system call and uses a file descriptor, whereas **fseek()** is a C function and uses a **FILE** pointer.

If all goes well, **lseek()** returns the new seek position. If an error occurs, such as seeking to a negative position, **lseek()** returns -1L and sets **errno** to an appropriate value.

See Also

libc, unistd.h
POSIX Standard, §6.5.3

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.

ltol3() — General Function (libc)

Convert long integer to file system block number
ltol3(*l3p, lp, n***)**
char **l3p*;*
long **lp*;*
unsigned *n*;

To conserve space inside i-nodes in COHERENT file systems, the system stores block addresses in three bytes. Programs that reference or maintain file systems use the functions **l3tol()** and **ltol3()** to convert between the three byte representation and **long** numbers.

ltol3() converts *n* **long** integers at address *lp* to the more compact form at address *l3p*.

See Also

libc

lvalue — Definition

An **lvalue** is an expression that designates a region of storage. The name comes from the assignment expression **e1=e2;**, in which the left operand must be an lvalue.

An identifier has both an *lvalue* (its address) and an *rvalue* (its contents). Some C operators require lvalue operands; for example, the left operand of an assignment statement must be an lvalue. Some operators give lvalue results; for example, if *e* is a pointer expression, ***e* is an lvalue that designates the object to which *e* points.

A *variable* can be used as an lvalue, whereas a constant cannot. For example, you cannot say

```
6 = (foo+bar);
```

A pointer is a variable, and can be manipulated within limits. An array name, however, is a constant and cannot be altered legally. Thus, the code

```
int foo[10];
int *bar;
foo = bar;
```

will generate an error message when you attempt to compile it, whereas

```
int foo[10];
int *bar;
bar = foo;
```

will not.

The following example shows the use of both an lvalue and a rvalue:

```
int i, *ip;

ip = &i;        /* ip is an lvalue, i and &i are rvalues */
i = 3;          /* i is an lvalue, 3 is an rvalue */
*ip = 4;        /* *ip is an lvalue, 4 is an rvalue */
```

See Also

Programming COHERENT, rvalue
ANSI Standard, §6.2.2.1

lx — Command

List directory's contents in columnar format
lx [*file ...***]**

lx is a link to the command **ls -x**. It prints each *file* in columnar format, like the command **lc**, except that directories and file names are printed together in one listing. If a *file* is a directory, **lx** lists its contents. If no *file* is named, **lx** lists the contents of the current directory by default.

See Also

commands, l, lc, lf, lr, ls