The XENIX™

Development System

Programmer's Reference

for the Apple Lisa 2™

The Santa Cruz Operation, Inc.

Contents

Appendix B XENIX System Calls

Chapter 1
Introduction

1.1 Overview

This manual explains how to use the functions given in the C language libraries of the XENIX system. In particular, it describes the functions of two C language libraries: the standard C library, and the screen updating and cursor movement library *curses*.

The C library functions may be called by any program that needs the resources of the XENIX system to perform a task. The functions let programs read and write to files in the XENIX file system, read and write to devices such as terminals and lineprinters, load and execute other programs, receive and process signals, communicate with other programs through pipes, share system resources, and process errors.

1.2 Using the C Library Functions

To use the C library functions you must include the proper function call and definitions in the program and specify the corresponding library is given when the program is compiled. The standard C library, contained in the file *libc.a*, is automatically specified when you compile a C language program. Other libraries, including the screen updating and cursor movement library contained in the file *libcurses.a*, must be explicitly specified when you compile a program with the –l option of the cc command (see Chapter 2, "Cc: a C Compiler" in the XENIX *Programmer's Guide*).

1.3 Using This Manual

This manual is intended to be used in conjunction with section S of the XENIX *Reference Manual*. If you have never used the C library functions before, read this manual first, then refer to the *Reference Manual* to learn about other functions. If you are familiar with the library functions, turn to the *Reference Manual* to see how these functions may differ from the ones you already know, then return to this manual for examples of the functions.

Chapter 1 introduces the C language libraries.

Chapter 2 describes the standard input and output functions. These function let a program read and write to the files of a XENIX file system.

Chapter 3 describes the screen processing functions. These functions let a program use the screen processing facilities of a user's terminal.

Chapter 4 describes the character and string processing functions. These functions let a program assign, manipulate, and compare characters and strings.

Chapter 2
Using the Standard I/O Functions

2.1 Introduction

Nearly all programs use some form of input and output. Some programs read from or write to files stored on disk. Others write to devices such as line printers. Many programs read from and write to the user's terminal. For this reason, the standard C library provides several predefined input and output functions that a programmer can use in programs.

This chapter explains how to use the I/O functions in the standard C library. In particular, it describes:

- Command line arguments
- Standard input and output files
- Stream functions for ordinary files
- Low-level functions for ordinary files
- Random access functions

2.1.1 Preparing for the I/O Functions

To use the standard I/O functions a program must include the file *stdio.h*, which defines the needed macros and variables. To include this file, place the following line at the beginning of the program.

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

The actual functions are contained in the library file *libc.a*. This file is automatically read whenever you compile a program, so no special argument is needed when you invoke the compiler.

2.1.2 Special Names

The standard I/O library uses many names for special purposes. In general, these names can be used in any program that has included the *stdio.h* file.

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

at the beginning of the main program function. When a program begins execution, "argc" contains the count, and each element in "argv" contains a pointer to one argument.

An argument is stored as a null-terminated string (i.e., a string ending with a null character, \0). The first string (at "argv[0]") is the program name. The argument count is never less than 1, since the program name is always considered the first argument.

In the following example, command line arguments are read and then echoed on the terminal screen. This program is similar to the XENIX echo command.

```
main(argc, argv)  /* echo arguments */
int argc;
char *argv[];
{
        int i;

        for (i = 1; i < argc; i++)
        printf("%s%c", argv[i], (i<argc-1) ? ' ' : '\n');
}
```

In the example above, an extra space character is added at the end of each argument to separate it from the next argument. This is required, since the system automatically removes leading and trailing whitespace characters (i.e., spaces and tabs) when it reads the arguments from the command line. Adding a newline character to the last argument is for convenience only; it causes the shell prompt to appear on the next line after the program terminates.

When typing arguments on a command line, make sure each argument is separated from the others by one or more whitespace characters. If an argument must contain whitespace characters, enclose that argument in double quotation marks. For example, in the command line

```
display 3 4 "echo hello"
```

the string "echo hello" is treated as a single argument Also enclose in double quotation marks any argument that contains characters recognized by the shell (e.g., <, >, |, and ^).

You should not change the values of the "argc" and "argv" variables. If necessary, assign the argument value to another variable and change that variable instead. You can give other functions in the program access to the arguments by assigning their values to external variables.

```
readn (p, cnt)
char p[];
int cnt;
{
        int i,c;

        i = 0;
        while ( i<cnt )
                if ( p[i++] = getchar()) != EOF ) {
                        p[i] = 0;
                        return(EOF);
                }
        return(0);
}
```

Note that if *getchar* is reading from the keyboard, it waits for characters to be typed before returning.

The *gets* function reads a string of characters from the standard input and copies the string to a given memory location. The function call has the form:

```
gets(s)
```

where *s* is a pointer to the location to receive the string. The function reads characters until it finds a newline character, then replaces the newline character with a null character (\0) and copies the resulting string to memory. The function returns the null pointer value NULL if the end of the file or an error is encountered. Otherwise, it returns the value of *s*.

The function is typically used to read a full line from the standard input. For example, the following program fragment reads a line from the standard input, stores it in the character array "cmdln" and calls a function (called *parse*) if no error occurs.

```
char cmdln[SIZE];

if ( gets(cmdln) != NULL )
        parse();
```

In this case, the length of the string is assumed to be less than "SIZE".

Note that *gets* cannot check the length of the string it reads, so overflow can occur.

The *scanf* function reads one or more values from the standard input where a value may be a character string or a decimal, octal, or hexadecimal number. The function call has the form:

```
scanf (format, argptr ...)
```

You may use the *getchar*, *gets*, and *scanf* functions in a single program. Just remember that each function reads the next available character, making that character unavailable to the other functions.

Note that when the standard input is the terminal keyboard, the *getchar*, *gets*, and *scanf* functions usually do not return a value until at least one newline character has been typed. This is true even if only one character is desired. If you wish to have immediate input on a single keystroke, see the example in the section "Using the *system* Call" in Chapter 3.

2.3.2 Writing to the Standard Output

You can write to the standard output with the *putchar*, *puts*, and *printf* functions.

The *putchar* function writes a single character to the output buffer. The function call has the form:

```
putchar (c)
```

where *c* is the character to be written. The function normally returns the same character it wrote, but will return the value EOF if an error is encountered.

The function is typically used in a conditional loop to write a string of characters to the standard output. For example, the function

```
writen (p,cnt)
char p[];
int cnt;
{
        int i;

        for (i=0; i<=cnt; i++)
                putchar( (i != cnt) ? p[i] : '\n');
}
```

writes "cnt" number of characters plus a newline character to the standard output.

The *puts* function copies the string found at a given memory location to the standard output. The function call has the form:

```
puts(s)
```

where *s* is a pointer to the location containing the string. The string may be any number of characters, but must end with a null character (\0). The function writes each character in the string to the standard output and replaces the null character at the end of the string with a newline character.

You may use the *putchar*, *puts*, and *printf* functions in a single program. Just remember that the output appears in the same order as it is written to the standard output.

2.3.3 Redirecting the Standard Input

You can change the standard input from the terminal keyboard to an ordinary file by using the normal shell redirection symbol, <. This symbol directs the shell to open for reading the file whose name immediately follows the symbol. For example, the following command line opens the file *phonelist* as the standard input to the program *dial*.

```
dial <phonelist
```

The *dial* program may then use the *getchar*, *gets*, and *scanf* functions to read characters and values from this file. Note that if the file does not exist, the shell displays an error message and stops the program.

Whenever *getchar*, *gets*, or *scanf* are used to read from an ordinary file, they return the value EOF if the end of the file or an error is encountered. It is useful to check for this value to make sure you do not continue to read characters after an error has occurred.

2.3.4 Redirecting the Standard Output

You can change the standard output of a program from the terminal screen to an ordinary file by using the shell redirection symbol, >. The symbol directs the shell to open for writing the file whose name immediately follows the symbol. For example, the command line

```
dial >savephone
```

opens the file *savephone* as the standard output of the program *dial* and not the terminal screen. You may use the *putchar*, *puts*, and *printf* functions to write to the file.

If the file does not exist, the shell automatically creates it. If the file exists, but the program does not have permission to change or alter the file, the shell displays an error message and does not execute the program.

2.3.5 Piping the Standard Input and Output

Another way to redefine the standard input and output is to create a pipe. A pipe simply connects the standard output of one program to the standard input of another. The programs may then use the standard input and output to pass information from one to the other. You can create a pipe by using the standard shell pipe symbol, |.

```
cat file1 file2 file3 | ccstrip
```

If you wish to save the stripped files, you can redirect the standard output of *ccstrip*. For example, this command line writes the stripped files to the file *clean*.

```
cat file1 file2 file3 | ccstrip >clean
```

Note that the *exit* function is used at the end of the program to ensure that any program which executes the *ccstrip* program will receive a normal termination status (typically 0) from the program when it completes. An explanation of the *exit* function and how to execute one program under control of another is given in Chapter 5.

2.4 Using the Stream Functions

The functions described so far have all read from the standard input and written to the standard output. The next step is to show functions that access files not already connected to the program. One set of standard I/O functions allows a program to open and access ordinary files as if they were a "stream" of characters. For this reason, the functions are called the stream functions.

Unlike the standard input and output files, a file to be accessed by a stream function must be explicitly opened with the *fopen* function. The function can open a file for reading, writing, or appending. A program can read from a file with the *getc*, *fgetc*, *fgets*, *fgetw*, *fread*, and *fscanf* functions. It can write to a file with the *putc*, *fputc*, *fputs*, *fputw*, *fwrite*, and *fprintf* functions. A program can test for the end of the file or for an error with the *feof* and *ferror* functions. A program can close a file with the *fclose* function.

2.4.1 Using File Pointers

Every file opened for access by the stream functions has a unique pointer associated with it called a file pointer. This pointer, defined with the predefined type FILE found in the *stdio.h* file, points to a structure that contains information about the file, such as the location of the buffer (the intermediate storage area between the actual file and the program), the current character position in the buffer, and whether the file is being read or written. The pointer can be given a valid pointer value with the *fopen* function as described in the next section. (The NULL value, like FILE, is defined in the *stdio.h* file.) Thereafter, the file pointer may be used to refer to that file until the file is explicitly closed with the *fclose* function.

Typically, a file pointer is defined with the statement:

```
FILE  *infile;
```

2.4.3 Reading a Single Character

The *getc* and *fgetc* functions return a single character read from a given file, and return the value EOF if the end of the file or an error is encountered. The function calls have the form:

c = getc (*stream*)

and

c = fgetc (*stream*)

where *stream* is the file pointer to the file to be read and *c* is the variable to receive the character. The return value is always an integer.

The functions are typically used in conditional loops to read a string of characters from a file. For example, the following program fragment continues to read characters from the file given to it by "infile" until the end of the file or an error is encountered.

```
int i;
char buf[MAX];
FILE *infile;

while ((c=getc(infile)) != EOF)
        buf[i++]=c;
```

The only difference between the functions is that *getc* is defined as a macro, and *fgetc* as a true function. This means that, unlike *getc*, *fgetc* may be passed as an argument in another function, used as a target for a breakpoint when debugging, or used to avoid any side effects of macro processing.

2.4.4 Reading a String from a File

The *fgets* function reads a string of characters a file and copies the string to a given memory location. The function call has the form:

fgets (*s*, *n*, *stream*)

where *s* is be a pointer to the location to receive the string, *n* is a count of the maximum number of characters to be in the string, and *stream* is the file pointer of the file to be read. The function reads *n*–*1* characters or upto to the first newline character, whichever occurs first. The function appends a null character (\0) to the last character read and then stores the string at the specified location. The function returns the null pointer value NULL if the end of the file or an error is encountered. Otherwise, it returns the pointer *s*.

function reads from the standard input. The function call has the form:

fscanf (*stream, format, argptr ...*)

where *stream* is the file pointer of the file to be read, *format* is a pointer to the string that defines the format of the input to be read, and *argptr* is one or more pointers to the variables that are to receive the formatted input. There must be one *argptr* for each format given in the *format* string. The format may be "%s" for a string, "%c" for a character, and "%d", "%o", or "%x" for a decimal, octal, or hexadecimal number, respectively. (Other formats are described in *scanf*(S) in the XENIX *Reference Manual*.) The function normally returns the number of arguments it read, but will return the value EOF if the end of the file or an error is encountered.

The function is typically used to read files that contain both numbers and text. For example, this program fragment reads a name and a decimal number from the file given by "file".

```
FILE *file;
int pay;
char name[20];

fscanf(file,"%s %d\n", name, &pay);
```

This program fragment copies the name to the character array "name" and the number to the integer variable "pay".

2.4.7 Writing a Single Character

The *putc* and *fputc* functions write single characters to a given file. The function calls have the forms:

putc (*c,stream*)

and

fputc (*c,stream*)

where *c* is the character to be written and *stream* is the file pointer to the file to receive the character. The function normally returns the character written, but will return the value EOF if an error is encountered.

The function is defined as a macro and may have undesirable side effects resulting from argument processing. In such cases, the equivalent function *fputc* should be used.

These functions are typically used in conditional loops to write a string of characters to a file. For example, this following program fragment writes characters from the array "name" to the file given by "out".

2.4.9 Writing Formatted Output

The *fprintf* function writes formatted output to a given file, just as the *printf* function writes to the standard output. The function call has the form:

fprintf (*stream, format* [, *arg*]...)

where *stream* is the file pointer of the file to be written to, *format* is a pointer to a string which defines the format of the output, and *arg* is one or more arguments to be written. There must be one *arg* for each format in the *format* string. The formats may be "%s" for a string, "%c" for a character, and "%d", "%o", or "%x" for a decimal, octal, or hexadecimal number, respectively. (Other formats are described in *printf*(S) in the XENIX *Reference Manual.*) If a string is requested, the corresponding *arg* must be a pointer, otherwise, the actual variable must be used. The function normally returns zero, but will return a nonzero number if an error is encountered.

The function is typically used to write output that contains both numbers and text. For example, to write a name and a decimal number to the file given by "outfile" use the following program fragment.

```
FILE *outfile;
int pay;
char name[20];

fprintf(outfile,"%s %d\n", name, pay);
```

The name is copied from the character array "name" and the number from the integer variable "pay".

2.4.10 Writing Records to a File

The *fwrite* function writes one or more records to a given file. The function call has the form:

fwrite (*ptr, size, nitems, stream*)

where *ptr* is a pointer to the first record to be written, *size* is the size (in bytes) of each record, *nitems* is the number of records to be written, and *stream* is the file pointer of the file. The *ptr* may point to a variable of any type (from a single character to a structure). The *size* should give the number of bytes in each item to be written. One way to ensure this is to use the *sizeof* function (see the example below). The function always returns the number of items actually written to the file whether or not the end of the file or an error is encountered.

The function is typically used to write binary data to a file. For example, the following program fragment writes two records to the file given by "database".

The function is typically used to test for errors before perform a subsequent read or write to the file. For example, in the following program fragment *ferror* tests the file given by "stream".

```
char *buf;
char x[5];

while ( !ferror(stream) )
        fread(buf, sizeof(x), 10, stream);
```

If it returns zero, the next item in the file given by "stream" is copied to "buf". Otherwise, execution passes to the next statement.

Further use of a file after a error is detected may cause undesirable results.

2.4.13 Closing a File

The *fclose* function closes a file by breaking the connection between the file pointer and the structure created by *fopen*. Closing a file empties the contents of the corresponding buffer and frees the file pointer for use by another file. The function call has the form:

fclose (*stream*)

where *stream* is the file pointer of the file to close. The function normally returns 0, but will return –1 if an error is encountered.

The *fclose* function is typically used to free file pointers when they are no longer needed. This is important because usually no more than 20 files can be open at the same time. For example, the following program fragment closes the file given by "infile" when the file has reached its end.

```
FILE *infile;

if ( feof(infile) )
        fclose( infile );
```

Note that whenever a program terminates normally, the *fclose* function is automatically called for each open file, so no explicit call is required unless the program must close a file before its end. Also, the function automatically calls *fflush* to ensure that everything written to the file's buffer actually gets to the file.

2.4.14 Program Example

This section shows how you may use the stream functions you have seen so far to perform useful tasks. The following program, which counts the characters, words, and lines found in one or more files, uses the *fopen*, *fprintf*, *getc*, and

```
#include <stdio.h>

main(argc, argv)  /* wc: count lines, words, chars */
int argc;
char *argv[];
{
        int c, i, inword;
        FILE *fp, *fopen();
        long linect, wordct, charct;
        long tlinect = 0, twordct = 0, tcharct = 0;

        i = 1;
        fp = stdin;
        do
        {
                if (argc > 1 &&
                        (fp=fopen(argv[i], "r")) == NULL) {
                        fprintf (stderr, "wc: can't open %s\n",
                                argv[i]);
                        continue;
                }
                linect = wordct = charct = inword = 0;
                while ((c = getc(fp)) != EOF) {
                        charct++;
                        if (c == '\n')
                                linect++;
                        if (c == ' ' || c == '\t' || c == '\n')
                                inword = 0;
                        else if (inword == 0) {
                                inword = 1;
                                wordct++;
                        }
                }
                printf("%7ld %7ld %7ld", linect, wordct, charct);
                printf(argc > 1 ? " %s\n" : "\n", argv[i]);
                fclose(fp);
                tlinect += linect;
                twordct += wordct;
                tcharct += charct;
        } while (++i < argc);
        if (argc > 2)
                printf("%7ld %7ld %7ld total\n", tlinect,
                        twordct, tcharct);
        exit(0);
}
```

The program uses "fp" as the pointer to receive the current file pointer. Initially this is set to "stdin" in case no filenames are present in the command line. If a filename is present, the program calls *fopen* and assigns the file pointer to "fp". If the file cannot be opened (in which case *fopen* returns NULL), the

and the *fflush* function lets a program flush the buffer before it is full.

2.5.2 Reopening a File

The *freopen* closes the file associated with a given file pointer, then opens a new file and gives it the same file pointer as the old file. The function call has the form:

freopen (*newfile*, *type*, *stream*)

where *newfile* is a pointer to the name of the new file, *type* is a pointer to the string that defines how the file is to be opened ("r" for read, "w" for writing, and "a" for appending), and *stream* is the file pointer of the old file. The function returns the file pointer *stream* if the new file is opened. Otherwise, it returns the null pointer value NULL.

The *freopen* function is used chiefly to attach the predefined file pointers "stdin", "stdout", and "stderr" to other files. For example, the following program fragment opens the file named by "newfile" as the new standard output file.

```
char *newfile;
FILE *nfile;

nfile = freopen(newfile,"r",stdout);
```

This has the same effect as using the redirection symbols in the command line of the program.

2.5.3 Setting the Buffer

The *setbuf* function changes the buffer associated with a given file to the program's own buffer. It can also change the access to the file to no buffering. The function call has the form:

setbuf (*stream*, *buf*)

where *stream* is a file descriptor and *buf* is a pointer to the new buffer, or is the null pointer value NULL if no buffering is desired. If a buffer is given, it must be BSIZE bytes in length, where BSIZE is a manifest constant found in *stdio.h*.

The function is typically used to to create a buffer for the standard output when it is assigned to the user's terminal, improving execution time by eliminating the need to write one character to the screen at a time. For example, the following program fragment changes the buffer of the standard output the location pointed at by "p".

Note that the value EOF must never be put back in the buffer.

2.5.5 Flushing a File Buffer

The *fflush* function empties the buffer of a give file by immediately writing the buffer contents to the file. The function call has the form:

```
fflush (stream)
```

where *stream* is the file pointer of the file. The function normally returns zero, but will return the value EOF if an error is encountered.

The function is typically used to guarantee that the contents of a partially filled buffer are written to the file. For example, the following program fragment empties the buffer for the file given by "outtty" if the error condition given by "errflag" is 0.

```
FILE *outtty;
int errflag;

if (errflag == 0)
        fflush( outtty );
```

Note that *fflush* is automatically called by the *fclose* function to empty the buffer before closing the file. This means that no explicit call to *fflush* is required if the file is also being closed.

The function ignores any attempt to empty the buffer of a file opened for reading.

2.6 Using the Low-Level Functions

The low-level functions provide direct access to files and peripheral devices. They are actually direct calls to the routines used in the XENIX operating system to read from and write to files and peripheral devices. The low-level functions give a program the same control over a file or device as the system, letting it access the file or device in ways that the stream functions do not. However, low-level functions, unlike stream functions, do not provide buffering or any other useful services of the stream functions. This means that any program that uses the low-level functions has the complete burden of handling input and output.

The low-level functions, like the stream functions, cannot be used to read from or write to a file until the file has been opened. A program may use the *open* function to open an existing or a new file. A file can be opened for reading, writing, or appending.

```
int in, out;

in = open( "/usr/accounts", O_RDONLY );
out = open( "/usr/tmp/scratch", O_WRONLY | O_CREAT, 0754 );
```

In the XENIX system, each file has 9 bits of protection information which control read, write, and execute permission for the owner of the file, for the owner's group, and for all others. A three-digit octal number is the most convenient way to specify the permissions. For example, in the example above the octal number "0755" specifies read, write, and execute permission for the owner, read and execute permission for the group, and read everyone else.

Note that if O_CREAT is given and the file already exists, the function destroys the file's old contents.

2.6.3 Reading Bytes From a File

The *read* function reads one or more bytes of data from a given file and copies them to a given memory location. The function call has the form:

n_read = read(*fd*, *buf*, *n*);

where *n_read* is the variable to receive the count of bytes actually read, *fd* is the file descriptor of the file, *buf* is a pointer to the memory location to receive the bytes read, and *n* is a count of the desired number of bytes to be read. The function normally returns the same number of bytes as requested, but will return fewer if the file does not have that many bytes left to be read. The function returns 0 if the file has reached its end, or –1 if an error is encountered.

When the file is a terminal, *read* normally reads only up to the next newline.

The number of bytes to be read is arbitrary. The two most common values are 1, which means one character at a time, and 1024, which corresponds to the physical block size on many peripheral devices.

2.6.4 Writing Bytes to a File

The *write* function writes one or more bytes from a given memory location to a given file. The function call has the form:

n_written = write(*fd*, *buf*, *n*);

where *n_written* is the variable to receive a count of bytes actually written, *fd* is the file descriptor of the file, *buf* is the name of the buffer containing the bytes to be written, and *n* is the number of bytes to be written.

The function always returns the number of bytes actually written. It is considered an error if the return value is not equal to the number of bytes

```
#define BUFSIZE         BSIZE

main()  /* copy input to output */
{
        char    buf[ BUFSIZE ];
        int     n;

        while ((n = read( 0, buf, BUFSIZE )) > 0)
                write(1, buf, n);
        exit(0);
}
```

The program uses the *read* function to read BUFSIZE bytes from the standard input (file descriptor 0). It then uses *write* to write the same number of bytes it read to the standard output (file descriptor 1). If the standard input file size is not a multiple of BUFSIZE, the last *read* returns a smaller number of bytes to be written by *write*, and the next call to *read* returns zero.

This program can be used like a copy command to copy the content of one file to another. You can do this by redirecting the standard input and output files.

The second example shows how the *read* and *write* functions can be used to construct higher level functions like *getchar* and *putchar*. For example, the following is a version of *getchar* which performs unbuffered input:

```
#define CMASK 0377      /* for making chars > 0 */

getchar()/* unbuffered single character input */
{
        char c;
        return((read(0, &c, 1) > 0) ? c & CMASK : EOF);
}
```

The variable "c" must be declared **char**, because *read* accepts a character pointer. In this case, the character being returned must be masked with octal 0377 to ensure that it is positive; otherwise sign extension may make it negative.

The second version of *getchar* reads input in large blocks, but hands out the characters one at a time:

```
error(s1, s2)       /* print error message and die */
char *s1, *s2;
{
        printf(s1, s2);
        printf("\n");
        exit(1);
}
```

There is a limit (usually 20) to the number of files that a program may have open simultaneously. Therefore, any program which intends to process many files must be prepared to reuse file descriptors by closing unneeded files.

2.6.7 Using Random Access I/O

Input and output operations on any file are normally sequential. This means each read or write takes place at the character position immediately after the last character read or written. The standard library, however, provides a number of stream and low-level functions that allow a program to access a file randomly, that is, to exactly specify the position it wishes to read from or write to next.

The functions that provide random access operate on a file's "character pointer". Every open file has a character pointer that points to the next character to be read from that file, or the next place in the file to receive a character. Normally, the character pointer is maintained and controlled by the system, but the random access functions let a program move the pointer to any position in the file.

2.6.8 Moving the Character Pointer

The *lseek* function, a low-level function, moves the character pointer in a file opened for low-level access to a given position. The function call has the form:

lseek(*fd*, *offset*, *origin*);

where *fd* is the file descriptor of the file, *offset* is the number of bytes to move the character pointer, and *origin* is the number that gives the starting point for the move. It may be 0 for the beginning of the file, 1 for the current position, and 2 for the end.

For example, this call forces the current position in the file whose descriptor is 3 to move to the 512th byte from the beginning of the file.

lseek(3, (long)512, 0)

Subsequent reading or writing will begin at that position. Note that *offset* must be a long integer and *fd* and *origin* must be integers.

The function may be used on either buffered or unbuffered files.

2.6.10 Rewinding a File

The *rewind* function, a stream function, moves the character pointer to the beginning of a given file. The function call has the form:

```
rewind (stream)
```

where *stream* is the file pointer of the file. The function is equivalent to the following function call

```
fseek (stream,0L,0);
```

It is chiefly used as a more readable version of the call.

2.6.11 Getting the Current Character Position

The *ftell* function, a stream function, returns the current position of the character pointer in the given file. The returned position is always relative to the beginning of the file. The function call has the form:

```
p = ftell (stream)
```

where *stream* is the file pointer of the file and *p* is the variable to receive the position. The return value is always a long integer. The function returns the value -1 if an error is encountered.

The function is typically used to save the current location in the file so that the program can later return to that position. For example, the following program fragment first saves the current character position in "oldp", then restores the file to this position if the current character position is greater than "800".

```
FILE *outfile;
long oldp;

oldp = ftell( outfile );

if ((ftell( outfile )) > 800)
        fseek(outfile, oldp, 0);
```

The *ftell* is identical to the function call

```
lseek( fd, (long)0, 1)
```

where *fd* is the file descriptor of the given stream file.

Chapter 3
Screen Processing

3.1 Introduction

This chapter explains how to use the screen updating and cursor movement library named *curses*. The library provides functions to create and update screen windows, get input from the terminal in a screen-oriented way, and optimize the motion of the cursor on the screen.

3.1.1 Screen Processing Overview

Screen processing gives a program a simple and efficient way to use the capabilities of the terminal attached to the program's standard input and output files. Screen processing does not rely on the terminal's type. Instead the screen processing functions use the XENIX terminal capability file */etc/termcap* to tailor their actions for any given terminal. This makes a screen processing program terminal-independent. The program can be run with any terminal as long as that terminal is described in the */etc/termcap* file.

The screen processing functions access a terminal screen by working through intermediate "screens" and "windows" in memory. A screen is a representation of what the entire terminal screen should look like. A window is a representation of what some portion of the terminal screen should look like. A screen can be made up of one or more windows. A window can be as small as a single character or as large as an entire screen.

Before a screen or window can be used, it must be created by using the *newwin* or *subwin* functions. These functions define the size of the screen or window in terms of lines and columns. Each position in a screen or window represents a place for a single character and corresponds to a similar place on the terminal screen. Positions are numbered according to line and column. For example, the position in the upper left corner of a screen or window is numbered (0,0) and the position immediately to its right is (0,1). A typical screen has 24 lines and 80 columns. Its upper left corner corresponds to the upper left corner of the terminal screen. A window, on the other hand, may be any size (within the limits of the actual screen). Its upper left corner can correspond to any position on the terminal screen. For convenience, the *initscr* function which initializes a program for screen processing also creates a default screen , *stdscr* (for "standard screen"). The *stdscr* may be used without first creating it. The function also creates *curscr* (for "current screen") which contains a copy of what is currently on the terminal screen.

To display characters at the terminal screen, a program must write these characters to a screen or window using screen processing functions such as *addch* and *waddch*. If necessary, a program can move to the desired position in the screen or window by using the *move* and *wmove* functions. Once characters are added to a screen or window, the program can copy the characters to the terminal screen by using the *refresh* or *wrefresh* function. These functions update the terminal screen according to what has changed in the given screen or window. Since the terminal screen is not changed until a program calls

Variables

Type	Name	Description
WINDOW*	curscr	A pointer to the current version of the terminal screen.
WINDOW*	stdscr	A pointer to the default screen used for updating when no explicit screen is defined.
char	Def_term	A pointer to the default terminal type if the type cannot be determined.
bool	My_term	The terminal type flag. If set, it causes the terminal specification in "Def_term" to be used, regardless of the real terminal type.
char	ttytype	A pointer to the full name of the current terminal.
int	LINES	The number of lines on the terminal.
int	COLS	The number of columns on the terminal.
int	ERR	The error flag. Returned by functions on an error.
int	OK	The okay flag. Returned by functions on successful operation.

3.2.2 Using Terminal Capability and Type

The *initscr* function uses the terminal capability descriptions given in the XENIX system's */etc/termcap* file to prepare the screen processing functions for creating and updating terminal screens. The descriptions define the character sequences required to perform a given operation on a given terminal. These sequences are used by the screen processing functions to add, insert, delete, and move characters on the screen. The descriptions are automatically read from the file when screen processing is initialized, so direct access by a program is not required.

The *initscr* function uses the shell's "TERM" variable to determine which terminal capability description to use. The "TERM" variable is usually assigned an identifier when a user logs in. This identifier defines the terminal type and is associated with a terminal capability description in the */etc/termcap* file.

If the "TERM" variable has no value, the functions use the default terminal type in the library's predefined variable "Def_term". This variable initially has the value "dumb" (for "dumb terminal"), but the user may change it to any desired value. This must be done before calling the *initscr* function.

In some cases, it is desirable to force the screen processing functions to use the default terminal type. This can be done by setting the library's predefined variable "My_term" to the value 1. The full name of the current terminal is stored in the predefined variable "ttytype".

Terminal capabilities, types, and identifiers are described in detail in *termcap*(F) in the XENIX *Reference Manual*.

3.2.3 Using Default Terminal Modes

The *initscr* function automatically sets a terminal to default operation modes. These modes define how the terminal displays characters sent to the screen and how it responds to characters typed at the keyboard. The *initscr* function sets the terminal to ECHO mode which causes characters typed at the keyboard to be displayed at the screen, and RAW mode which causes characters to be used as direct input (no editing or signal processing is done).

The default terminal modes can be changed by using the appropriate functions described in the section "Setting a Terminal Mode" in this chapter. If the modes are changed, they must be changed immediately after calling *initscr*. Terminal modes are described in detail in *tty*(M) in the XENIX *Reference Manual*.

```
#include <curses.h>

main ()
{

initscr();
        /* Program body. */
endwin();
}
```

Note that *endwin* must not be called if *initscr* has not been called. Also, *endwin* should be called before any call to the *exit* function. The *endwin* function must also be called if the *gettmode* and *setterm* functions have been called even if *initscr* has not.

3.3 Using the Standard Screen

The following sections explain how to use the standard screen to display and edit characters on the terminal screen.

3.3.1 Adding a Character

The *addch* function adds a given character to the standard screen and moves the character pointer one position to the right. The function call has the form:

```
addch( ch )
```

where *ch* gives the character to be added and must have **char** type. For example, if the current position is (0, 0), the function call

```
addch('A')
```

places the letter "A" at this position and moves the pointer to (0, 1).

If a newline ('\n') character is given, the function deletes all characters from the current position to the end of the line and moves the pointer one line down. If the **newline** flag is set, the function deletes the characters and moves the pointer to the beginning of the next line. If a return ('\r') is given, the function moves the pointer to the beginning of the current line. If a tab ('\t') is given, the function moves the pointer to the next tab stop, adding enough spaces to fill the gap between the current position and the stop. Tab stops are placed at every eight character positions.

The function returns ERR if it encounters an error, such as illegal scrolling.

prints the number "15" immediately after the name.

The function returns ERR if it encounters an error such as illegal scrolling.

3.3.4 Reading a Character From the Keyboard

The *getch* function reads a single character from the terminal keyboard and returns the character as a value. The function call has the form:

```
c = getch()
```

where *c* is the variable to receive the character.

The function is typically used to read a series of individual characters. For example, in the following program fragment, characters are read and stored until a newline or the end of the file is encountered, or until the buffer size has been reached.

```
char c, p[MAX];
int i;

i = 0;
while ((c=getch()) != '\n' && c != EOF && i <MAX )
        p[i++] = c;
```

If the terminal is set to ECHO mode, *getch* copies the character to the standard screen; otherwise, the screen remains unchanged. If the terminal is not set to RAW or NOECHO mode, *getch* automatically sets the terminal to CBREAK mode, then restores the previous mode after reading the character. Terminal modes are described later in the chapter.

The function returns ERR if it encounters an error such as illegal scrolling.

3.3.5 Reading a String From the Keyboard

The *getstr* function reads a string of characters from the terminal keyboard and copies the string to a given location. The function call has the form:

```
getstr( str )
```

where *str* is a character pointer to the variable or location to receive the string. When typed at the keyboard, the string must end with a newline character or with the end-of-file character. The extra character is replaced by a null character when the string is stored. It is the programmer's responsibility to ensure that *str* has adequate space to store the typed string.

The function is typically used to read names and other text from the keyboard. For example, in the following program fragment, reads a filename from the

3.3.7 Moving the Current Position

The *move* function moves the pointer to the given position. The function call has the form:

```
move (y, x)
```

where *y* is an integer value giving the new row position, and *x* is an integer value giving the new column position. For example, if the current position is (0,0), the function call

```
move(5,4)
```

moves the pointer to line 5, column 4.

The function returns ERR if it encounters an error such as illegal scrolling.

3.3.8 Inserting a Character

The *insch* function inserts a character at the current position and shifts the existing character (and all characters to its right) one position to the right. The function call has the form:

```
insch ( c )
```

where *c* is the character to be inserted.

The function is typically used to insert a series of characters into an existing line. For example, in the following program fragment *insch* is used to insert the number of characters given by "cnt" into the standard screen a the current position.

```
int cnt;
char *string;

while ( cnt != 0 ) {
        insch(string[cnt]);
        cnt--;
        }
```

The function returns ERR if it encounters an error such as illegal scrolling.

3.3.9 Inserting a Line

The *insertln* function inserts a blank line at the current position and moves the existing line (and all lines below it) down one line, causing the last line to move off the bottom of the screen. The function call has the form:

The *deleteln* function is used to delete existing lines from the standard screen. For example, in the following program fragment *deleteln* is used to delete a line from the standard screen if the count in "cnt" is 79.

```
int cnt;

if ( cnt == 79 )
        deleteln();
```

3.3.12 Clearing the Screen

The *clear* and *erase* functions clear all characters from the standard screen by replacing them with spaces. The functions are typically used to prepare the screen for new text.

The *clear* function clears all characters from the standard screen, moves the pointer to (0,0), and sets the standard screen's clear flag. The flag causes the next call to the *refresh* function to clear all characters from the terminal screen.

The *erase* function clears the standard screen, but does not set the clear flag. For example, in the following program fragment *clear* clears the screen if the input value is 12.

```
char c;

if ((c=getch()) == 12)
        clear();
```

3.3.13 Clearing a Part of the Screen

The *clrtobot* and *clrtoeol* functions clear one or more characters from the standard screen by replacing the characters with spaces. The functions are typically used to prepare a part of the standard screen for new characters.

The *clrtobot* function clears the screen from the current position to the bottom of the screen. For example, if the current position is (10,0), the function call

```
clrtobot();
```

clears all characters from line 10 and all lines below line 10.

The *clrtoeol* function clears the standard screen from the current position to the end of the current line. For example, if the current position is (10,10), the function call

```
clrtoeol();
```

where *win* is the pointer variable to receive the return value, *lines* and *cols* are integer values that give the total number of lines and columns, respectively, in the window, and *begin_y* and *begin_x* are integer values that give the line and column positions, respectively, of the upper left corner of the window when displayed on the terminal screen. The *win* variable must have type WINDOW*.

The function is typically used in programs that maintain a set of windows, displaying different windows at different times or alternating between window as needed. For example, in the following program fragment *newwin* creates a new window and assigns the pointer to this window to the variable *midscreen*.

```
WINDOW *midscreen;

midscreen = newwin(5, 10, 9, 35);
```

The window has 5 lines and 10 columns. The upper left corner of the window is placed at the position (9,35) on the terminal screen.

If either *lines* or *cols* is zero, the function automatically creates a window that has "LINES - *begin_y*" lines or "COLS - *begin_x*" columns, where "LINES" and "COLS" are the predefined constants giving the total number of lines and columns on the terminal screen. For example, the function call

```
newwin(0, 0, 0, 0)
```

creates a new window whose upper left corner is at position (0,0) and that has "LINES" lines and "COLS" columns.

Note

You must not create windows that exceed the dimensions of the actual screen.

The *newwin* function returns the value (WINDOW*) ERR on an error, such as insufficient memory for the new window.

3.4.2 Creating a Subwindow

The *subwin* function creates a subwindow and returns a pointer to the new window. A subwindow is a window which shares all or part of the character space of another window and provides an alternate way to access the characters in that space. The function call has the form:

pointer to the given string. For example, if the current position is (0,0), the function call

```
waddstr(midscreen, "line");
```

places the beginning of the string "line" at this position and moves the pointer to (0,4).

The *wprintw* function prints one or more values on the given window, where a value may be a string, a character, or a decimal, octal, or hexadecimal number. The function call has the form:

```
wprintw( win, fmt [, arg ] ...)
```

where *win* is a pointer to the window to receive the values, *fmt* is a pointer to a string that defines the format of the values, and *arg* is a value to be printed. If more than one *arg* is given, each must be separated from the preceding with a comma (,). For each *arg* given, there must be a corresponding format given in *fmt*. A format may be "%s" for string, "%c" for character, and "%d", "%o", or "%x" for a decimal, octal, or hexadecimal number, respectively. (Other formats are described in *printf*(S) in the XENIX *Reference Manual.*) If "%s" is given, the corresponding *arg* must be a character pointer. For other formats, the actual value or a variable containing the value may be given.

The function is typically used to copy both numbers and strings to the standard screen at the same time. For example, in the following program fragment *wprintw* prints a name and then the number "15" at the current position in the window "midscreen".

```
char *name;

wprintw(midscreen, "%s %d", name, 15);
```

Note that when a newline, return, or tab character is given to a *waddch*, *waddstr*, or *wprintw* function, the functions perform the same actions as described for the *addch* function. The functions return ERR if they encounter errors such as illegal scrolling.

3.4.4 Reading and Scanning for Input

The *wgetch*, *wgetstr*, and *wscanw* functions read characters, strings, and numbers from the standard input file and usually echo the values by copying them to the given window.

The *wgetch* function reads a single character from the standard input file and returns the character as a value. The function call has the form:

```
c = wgetch( win )
```

The function is typically used to read a combination of strings and numbers from the keyboard. For example, in the following program fragment *wscanw* reads a name and a number from the keyboard.

```
char name[20];
int id;

wscanw(midscreen, "%s %d", name, &id);
```

In this example, the name is stored in the character array "name" and the number in the integer variable "id".

If the terminal is set to ECHO mode, the function copies the string to the given window. If the terminal is not set to RAW or NOECHO mode, the function automatically sets the terminal to CBREAK mode, then restores the previous mode after reading the character.

The functions return ERR if they encounter errors such as illegal scrolling.

3.4.5 Moving a the Current Position in a Window

The *wmove* function moves the current position in a given window. The function call has the form:

wmove (*win*, *y*, *x*)

where *win* is a pointer to a window, *y* is an integer value giving the new line position, and *x* is an integer value giving the new column position. For example, the function call

```
wmove(midscreen, 4, 4)
```

moves the current position in the window "midscreen" to (4,4).

The function returns ERR if it encounters an error such as illegal scrolling.

3.4.6 Inserting Characters

The *winsch* and *winsertln* functions insert characters and lines into a given window.

The *winsch* function inserts a character at the current position and shifts the existing character (and all characters to its right) one position to the right. The function call has the form:

winsch (*win*, *c*)

where *win* is a pointer to a window, and *c* is the character to be inserted.

The *wdeleteln* function deletes the current line and shifts the line below the deleted line (and all lines below it) one line up, leaving the last line in the screen blank. The function call has the form:

```
wdeleteln( win )
```

where *win* is a pointer to a window.

The function is typically used to delete existing lines from a given window. For example, in the following program fragment *wdeleteln* deletes the lines in "midscreen" until "cnt" is equal to zero.

```
int cnt;

while ( cnt != 0 ) {
        wdeleteln(midscreen);
        cnt--;
        }
```

3.4.8 Clearing the Screen

The *wclear*, *werase*, *wclrtobot*, and *wclrtoeol* functions clear all or part of the characters from the given window by replacing them with spaces. The functions are typically used to prepare the window for new text.

The *wclear* function clears all characters from the window, moves the pointer to (0,0), and sets the standard screen's clear flag. The flag causes the next *refresh* function call to clear all characters from the terminal screen. The function call has the form:

```
wclear( win )
```

where *win* is the window to be cleared.

The *werase* function clears the given window, moves the pointer to (0,0), but does not set the clear flag. It is used whenever the contents of the terminal screen must be preserved. The function call has the form:

```
werase( win )
```

where *win* is a pointer to the window to be cleared.

The *wclrtobot* function clears the window from the current position to the bottom of the screen. The function call has the form:

```
wclrtobot( win )
```

where *win* is a pointer to the window to be cleared. For example, if the current

---

Note

If *curscr* is given with *wrefresh*, the function restores the actual screen to its most recent contents. This is useful for implementing a "redraw" feature for screens that become cluttered with unwanted output.

The function returns ERR if it encounters an error such as illegal scrolling. If an error is encountered, the function attempts to update as much of the screen as possible without causing the scroll.

3.4.10 Overlaying Windows

The *overlay* function copies all characters, except spaces, from one window to another, moving characters from their original positions in the first window to identical positions in the second. The function effectively lays the first window over the second, letting characters in the second window that would otherwise be covered by spaces remain unchanged. The function call has the form:

overlay(*win1*, *win2*)

where *win1* is a pointer to the window to be copied, and *win2* is a pointer to the window to receive the copied text. The starting positions of *win1* and *win2* must match, otherwise an error occurs. If *win1* is larger than *win2*, the function copies only those lines and columns in *win1* that fit in *win2*.

The function is typically used to build a composite screen from overlapping windows. For example, in the following program fragment *overlay* is used to build the standard screen from two different windows.

```
WINDOW *info, *cmdmenu;

overlay(info, stdscr);
overlay(cmdmenu, stdscr);
refresh();
```

3.4.11 Overwriting a Screen

The *overwrite* function copies all characters, including spaces, from one window to another, moving characters from their positions in the first window to identical positions in the second. The function effectively writes the contents of the first window over the second, destroying the previous contents of the second window. The function call has the form:

```
c = inch()
```

where *c* is the character variable to receive the character read.

The *winch* function reads a character from a given window or screen. The function call has the form:

```
c = winch( win )
```

where *win* is the pointer to the window containing the character to be read.

The functions are typically used to compare the actual contents of a window with what is assumed to be there. For example, in the following program fragment *inch* and *winch* are used to compare the characters at position (0,0) in the standard screen and in the window named "altscreen".

```
char c1, c2;

c1 = inch();
c2 = winch(altscreen);
if (c1 != c2 )
        error();
```

Note that reading a character from a window does not alter the contents of the window.

3.4.14 Touching a Window

The *touchwin* function makes the entire contents of a given window appear to be modified, causing a subsequent *refresh* call to copy all characters in the window to the terminal screen. The function call has the form:

```
touchwin( win )
```

where *win* is a pointer to the window to be touched.

The function is typically used when two or more overlapping windows make up the terminal screen. For example, the function call

```
touchwin(leftscreen);
```

is used to touch the window named "leftscreen". A subsequent *refresh* copies all characters in "leftscreen" to the terminal screen.

3.4.15 Deleting a Window

The *delwin* function deletes a given window from memory, freeing the space previously occupied by the window for other windows or for dynamically

The *standout* function sets the standout attribute for characters added to the standard screen. The function call has the form:

```
standout()
```

No arguments are required.

The *wstandout* function sets the standout attribute of characters added to the given window or screen. The function call has the form:

```
wstandout( win )
```

where *win* is a pointer to a window.

The functions are typically used to make error messages or instructions clearly visible when displayed at the terminal screen. For example, in the following program fragment *standout* sets the standout character attribute before adding an error message to the standard screen.

```
if ( code == 5 ) {
        standout();
        addstr("Illegal character.\n");
        }
```

Note that the actual appearence of characters with the standout attribute depends on the given terminal. This attribute is defined by the SO and SE (or US and UE) sequences given in the terminal's *termcap* entry (see *termcap*(M) in the XENIX *Reference Manual*).

3.5.3 Restoring Normal Characters

The *standend* and *wstandend* functions restore the normal character attribute, causing characters subsequently added to a given window or screen to be displayed as normal characters.

The *standend* function restores the normal attribute for the standard screen. The function call has the form:

```
standend()
```

No arguments are required.

The *wstandend* function restores the normal attribute for a given window or screen. The function call has the form:

```
wstandend( win )
```

where *win* is a pointer to a window.

leaveok(*win*, *state*)

where *win* is a pointer to the window containing the flag to be set, and *state* is a Boolean value defining the state of the flag. If *state* is TRUE the flag is set; if FALSE, the flag is cleared. For example, the function call

leaveok(stdscr, TRUE);

sets the cursor flag.

The *scrollok* function sets or clears the scroll flag for the given window. If the flag is set, scrolling through the window is allowed. If the flag is clear, then no scrolling is allowed. The function call has the form:

scrollok(*win*, *state*)

where *win* is a pointer to a window, and *state* is a Boolean value defining how the flag is to be set. If *state* is TRUE, the flag is set; if FALSE, the flag is cleared. The flag is initially clear, making scrolling illegal.

The *clearok* function sets and clears the clear flag for a given screen. The function call has the form:

clearok(*win*, *state*)

where *win* is a pointer to the desired screen, and *state* is a Boolean value. The function sets the flag if *state* is TRUE, and clears the flag if FALSE. For example, the function call

clearok(stdscr, TRUE)

sets the clear flag for the standard screen.

When the clear flag is set, each *refresh* call to the given screen automatically clears the screen by passing a clear-screen sequence to the terminal. This sequence affects the terminal only; it does not change the contents of the screen.

If *clearok* is used to set the clear flag for the current screen "curscr", each call to *refresh* automatically clears the screen, regardless of which window is given in the call.

3.5.6 Scrolling a Window

The *scroll* function scrolls the contents of a given window upward by one line. The function call has the form:

scroll(*win*)

where *win* is a pointer to the window to be scrolled. The function should be used

The *echo* function sets the ECHO mode for the terminal, causing each character typed at the keyboard to be displayed at the terminal screen. The function call has the form:

```
echo()
```

No arguments are required.

The *nl* function sets a terminal to NEWLINE mode, causing all newline characters to be mapped to a corresponding newline and return character combination. The function call has the form:

```
nl()
```

No arguments are required.

The *raw* function sets the RAW mode for the terminal, causing each character typed at the keyboard to be sent as direct input. The RAW mode disables the function of the editing and signal keys and disables the mapping of newline characters into newline and return combinations. The function call has the form:

```
raw()
```

No arguments are required.

3.7.2 Clearing a Terminal Mode

The *nocrmode*, *noecho*, *nonl*, and *noraw* functions clear the current terminal mode, allowing input to be processed according to a previous mode.

The *nocrmode* function clears a terminal from the CBREAK mode. The function call has the form:

```
nocrmode()
```

No arguments are required.

The *noecho* function clears a terminal from the ECHO mode. This mode prevents characters typed at the keyboard from being displayed on the terminal screen. The function call has the form:

```
noecho()
```

No arguments are required.

The *nonl* function clears a terminal from NEWLINE mode, causing newline characters to be mapped into themselves. This allows the screen processing functions to perform better optimization. The function call has the form:

The function is normally called by the *initscr* function.

3.7.5 Saving and Restoring the Terminal Flags

The *savetty* function saves the current terminal flags, and the *resetty* function restores the flags previously saved by the *savetty* function. These functions are performed automatically by *initscr* and *endwin* functions. They are not required when performing ordinary screen processing.

3.7.6 Setting a Terminal Type

The *stterm* function sets the terminal type to the given type. The function call has the form:

setterm(*name*)

where *name* is a pointer to a string containing the terminal type identifier. The function is normally called by the *initscr* function, but may be used in special cases.

3.7.7 Reading the Terminal Name

The *longname* function converts a given *termcap* identifier into the full name of the corresponding terminal. The function call has the form:

longname(*termbuf*, *name*)

where *termbuf* is a pointer to the string containing the terminal type identifier, and *name* is a character pointer to the location to receive the long name. The terminal type identifier must exist in the */etc/termcap* file.

The function is typically used to get the full name of the terminal currently being used. Note that the current terminal's identifier is stored in the variable "ttytype", which may be used to receive a new name.

Chapter 4
Character and String Processing

4.1 Introduction

Character and string processing is an important part of many programs. Programs regularly assign, manipulate, and compare characters and strings in order to complete their tasks. For this reason, the standard library provides a variety of character and string processing functions. These functions give a convenient way to test, translate, assign, and compare characters and strings.

To use the character functions in a program the file, *ctype.h*, which provides the definitions for special character macros, must be included in the program. The line

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

must appear at the beginning of the program.

To use the string functions, no special action is required. These functions are defined in the standard C library and are read whenever you compile a C program.

4.2 Using the Character Functions

The character functions test and convert characters. Many character functions are defined as macros, and as such cannot be redefined or used as a target for a breakpoint when debugging.

4.2.1 Testing for an ASCII Character

The *isascii* function tests for characters in the ASCII character set, i.e., characters whose values range from 0 to 127. The function call has the form:

```
isascii (c)
```

where *c* is the character to be tested. The function returns a nonzero (true) value if the character is ASCII, otherwise it returns zero (false). For example, in the following program fragment *isascii* determines whether or not the value in "c" read from the file given by "data" is in the acceptable ASCII range.

```
FILE *data;
int c;

c = fgetc(data);
if (!isascii(c))
        notext();
```

In this example, a function named *notext* is called if the character is not in range.

4.2.4 Testing for a Letter

The *isalpha* function tests for uppercase or lowercase letters, i.e., alphabetic characters. The function call has the form:

isalpha (*c*)

where *c* is the character to be tested. The function returns a nonzero (true) value if the character is a letter, otherwise it returns zero. For example, the function call

isalpha('a')

returns a nonzero value, but the call

isalpha('1')

returns zero.

4.2.5 Testing for Control Characters

The *iscntrl* function test for control characters, i.e., characters whose ASCII values are in the range 0 to 31 or is 127. The function call has the form:

iscntrl (*c*)

where *c* is the character to be tested. The function returns a nonzero (true) value if the character is a control character, otherwise it returns zero (false). For example, in the program following fragment *iscntrl* determines whether or not the character in "c" read from the file given by "infile" is a control character.

```
FILE *infile, *outfile;
int c;

c = fgetc(infile);
if ( !iscntrl(c) )
        fputc( c, outfile );
```

The *fputc* function is ignored if the character is a control character.

4.2.6 Testing for a Decimal Digit

The *isdigit* function tests for decimal digits. The function call has the form:

isdigit (*c*)

neither control characters nor alphanumeric characters. The function call has the form:

ispunct (*c*)

where *c* is the character to be tested. The function returns a nonzero function if the character is a punctuation character, otherwise it returns zero.

4.2.10 Testing for Whitespace

The *isspace* function tests for whitespace characters, i.e, the space, horizontal tab, vertical tab, carriage return, formfeed, and newline characters. The function call has the form:

isspace (*c*)

where *c* is the character to be tested. The function returns a nonzero value if the character is a whitespace character, otherwise it returns zero.

4.2.11 Testing for Case in Letters

The *isupper* and *islower* functions test for uppercase and lowercase letters, respectively. The function calls have the form:

isupper (*c*)

and

islower (c)

where *c* is the character to be tested. The function returns a nonzero value if the character is the proper case, otherwise it returns zero. For example, the function call

isupper('b')

returns zero (false), but the call

islower('b')

returns a nonzero (true) value.

4.2.12 Converting the Case of a Letter

The *tolower* and *toupper* functions convert the case of a given letter. The function calls have the form:

strcat (*dst*, *src*)

where *dst* is a pointer to the string to receive the new characters, and *src* is a pointer to the string containing the new characters. The function appends the new characters in the same order as they appear in *src*, then appends a null character (\0) to the last character in the new string. The function always returns the pointer *dst*.

The function is typically used to build a string such as a full pathname from two smaller strings. For example, in the following program fragment *strcat* concatenates the string "temp" to the contents of the character array "dir".

```
char dir[MAX] = "/usr/";

strcat(dir, "temp");
```

4.3.2 Comparing Strings

The *strcmp* function compares the characters in one string to those in another and returns an integer value showing the result of the comparison. The function call has the form:

strcmp (*s1*, *s2*)

where *s1* and *s2* are the pointers to the strings to be compared. The function returns zero if the strings are equal (i.e., have the same characters in the same order). If the strings are not equal, the function returns the difference between the ASCII values of the first unequal pair of characters. The value of the second string character is always subtracted from the first. For example, the function call

```
strcmp("Character A", "Character A");
```

returns zero since the strings are identical in every way, but the function call

```
strcmp("Character A", "Character B");
```

returns –1 since the ASCII value of "B" is one greater than "A".

Note that the *strcmp* function continues to compare characters until a mismatch is found. If one string is shorter than the other, the function usually stops at the end of the shorter string. For example, the function call

```
strcmp("Character A", "Character ")
```

returns 65, that is, the difference between the null character at the end of the second string and the "A" in the first string.

strncat (*dst*, *src*, *n*)

where *dst* is a pointer to the string to receive the new characters, *src* is a pointer to the string containing the new characters, and *n* is an integer value giving the number of characters to be concatenated. The function appends the given number of characters to the end of the *dst* string, then returns the pointer *dst*.

In the following program fragment, *strncat* copies the first three characters in "letter" to the end of "cover".

```
char cover[] = "cover";
char letter[] = "letter";

strncat( cover, letter, 3);
```

This example creates the new string "coverlet" in "cover".

4.3.6 Comparing Characters in Strings

The *strncmp* function compares one or more pairs of characters in two given strings and returns an integer value which gives the result of the comparison. The function call has the form:

strncmp (*s1*, *s2*, *n*)

where *s1* and *s2* are pointers to the strings to be compared, and *n* is an integer value giving the number of characters to compare. The function returns zero if the first *n* characters are identical. Otherwise, the function returns the difference between the ASCII values of the first unequal pair of characters. The function generates the difference by subtracting the second string character from the first.

For example, the function call

strncmp("Character A", "Character B", 5)

returns zero because the first five characters are identical, but the function call

strncmp("Character A", "Character B", 11)

returns –1 because the value of "B" is one greater than "A".

Note that the function continues to compare characters until a mismatch or the end of a string is found.

```
char datestr[] = {"THU MAR 29 11:04:40 EST 1983"};
char month[4];
char year[5];

sscanf(datestr,"%*3s%3s%*2s%*8s%*3s%4s",month,year);
printf("%s, %s\n",month,year);
```

The first value (a three-character string) is stored at the location pointed to by "month", the second value (a four-character string) is stored at the location pointed to by "year".

4.3.9 Writing Values to a String

The *sprintf* function writes one or more values to a given string. The function call has the form:

sprintf (*s*, *format* [, *arg*] ...)

where *s* is a pointer to the string to receive the value, *format* is a pointer to a string which defines the format of the values to be written, and *arg* is the variable or value to be written. If more than one *arg* is given, they must be separated by commas (,). The *format* string may contain the same formats as given for *printf* (see *printf*(S) in the XENIX *Reference Manual*). After all values are written to the string, the function adds a null character (\0) to the end of the string. The function normally returns zero, but will return a nonzero value if an error is encountered.

The function is typically used to build a large string from several values of different format. For example, in the following program fragment *sprintf* writes three values to the string pointed to by "cmd".

```
char cmd[100];
char *doc = "/usr/src/cmd/cp.c"
int width = 50;
int length = 60;

sprintf(cmd,"pr -w%d -l%d %s\n",width,length,doc);
system(cmd);
```

In this example, the string created by *sprintf* is used in a call to the *system* function. The first two values are the decimal numbers given by "width" and "length". The last value is a string (a filename) and is pointed to by *doc*. The final string has the form:

```
pr -w50 -l60 /usr/src/cmd/cp.c
```

Note that the string to receive the values must have sufficient length to store those values. The function cannot check for overflow.

Chapter 5
Using Process Control

5.1 Introduction

This chapter describes the process control functions of the standard C library. The functions let a program call other programs, using a method similar to calling functions.

There are a variety of process control functions. The *system* and *exit* functions provide the highest level of execution control and are used by most programs that need a straightforward way to call another program or terminate the current one. The *execl*, *execv*, *fork*, and *wait* functions provide low-level control of execution and are for those programs which must have very fine control over their own execution and the execution of other programs. Other process control functions such as *abort* and *exec* are described in detail in section S of the XENIX *Reference Manual*.

The process control functions are a part of the standard C library. Since this library is automatically read when compiling a C program, no special library argument is required when invoking the compiler.

5.2 Using Processes

"Process" is the term used to describe a program executed by the XENIX system. A process consists of instructions and data, and a table of information about the program, such as its allocated memory, open files, and current execution status.

You create a process whenever you invoke a program through a shell. The system assigns a unique process ID to a program when it is invoked, and uses this ID to control and manage the program. The unique IDs are needed in a system running several processes at the same time.

You can also create a process by directing a program to call another program. This causes the system to perform the same functions as when it invokes a program through a shell. In fact, these two methods are actually the same method; invoking a program through a shell is nothing more than directing a program (the shell) to call another program.

The system handles all processes in essentially the same way, so the sections that follow should give you valuable information for writing your own programs and an insight into the XENIX system itself.

5.3 Calling a Program

The *system* function calls the given program, executes it, and then returns control to the original program. The function call has the form:

exit (*status*)

where *status* is the integer value to be sent to the system as the termination status.

The function is typically used to terminate a program before its normal end, such as after a serious error. For example, in the following program fragment *exit* stops the program and sends the integer value "2" to the system if the *fopen* function returns the null pointer value NULL.

```
FILE *ttyout;

if ( fopen(ttyout,"r") == NULL )
        exit(2);
```

Note that the *exit* function automatically closes each open file in the program before returning to the system. This means no explicit calls to the *fclose* or *close* functions are required before an exit.

5.5 Starting a New Program

The *execl* and *execv* functions cause the system to overlay the calling program with the given one, allowing the calling program to terminate while the new program continues execution.

The *execl* function call has the form:

execl (*pathname*, *command-name*, *argptr* ...)

where *pathname* is a pointer to a string containing the full pathname of the command you want to execute, *command-name* is a pointer to a string containing the name of the program you want to execute, and *argptr* is one or more pointers to strings which contain the program arguments. Each *argptr* must be separated from any other argument by a comma. The last *argptr* in the list must be the null pointer value NULL. For example, in the call

```
execl("/bin/date", "date", NULL);
```

the date command, whose full pathname is "/bin/date", takes no arguments, and in the call

```
execl("/bin/cat", "cat", file1, file2, NULL);
```

the cat command, whose full pathname is "/bin/cat", takes the pointers "file1" and "file2" as arguments.

The *execv* function call has the form:

If the program *display* is not found or lacks the necessary permissions, the original program resumes control and displays an error message.

Note that the *execl* and *execv* functions will not expand metacharacters (e.g., <, >, *, ?, and [ ]) given in the argument list. If a program needs these features, it can use *execl* or *execv* to call a shell as described in the next section.

5.6 Executing a Program Through a Shell

One drawback of the *execl* and *execv* functions is that they do not provide the metacharacter features of a shell. One way to overcome this problem is to use *execl* to execute a shell and let the shell execute the command you want.

The function call has the form:

```
execl ("/bin/sh","sh","-c", command-line, NULL);
```

where *command-line* is a pointer to the string containing the command line needed to execute the program. The string must be exactly as it would appear if typed at the terminal.

For example, a program can execute the command

```
cat *.c
```

(which contains the metacharacter *) with the call

```
execl("/bin/sh", "sh", "-c", "cat *.c", NULL);
```

In this example, the full pathname */bin/sh* and command name *sh* start the shell. The argument "-c" causes the shell to treat the argument "cat *.c" as a whole command line. The shell expands the metacharacter and displays all files which end with *.c*, something that the cat command cannot do by itself.

5.7 Duplicating a Process

The *fork* function splits an executing program into two independent and fully-functioning processes. The function call has the form:

```
fork ()
```

No arguments are required.

The function is typically used to make multiple copies of any program that must take divergent actions as a part of its normal operation, e.g., a program that must use the *execl* function yet still continue to execute. The original program, called the "parent" process, continues to execute normally, just as it would after any other function call. The new process, called the "child"

```
int status;
char *pathname;
char *cmd[ ];

if (fork() == 0)
        execv(pathname, cmd);
wait(&status);
```

The *wait* function always copies a status value to its argument. The status value is actually two 8-bit values combined into one. The low-order 8 bits is the termination status of the child as defined by the system. This status is zero for normal termination and nonzero for other kinds of termination, such as termination by an interrupt, quit, or hangup signal (see *signal*(S) in the XENIX *Reference Manual* for a description of the various kinds of termination). The next 8 bits is the termination status of the child as defined by its own call to *exit*. If the child did not explicitly call the function, the status is zero.

5.9 Inheriting Open Files

Any program called by another program or created as a child process to aprogram automatically inherits the original program's open files and standard input, output, and error files. This means if the file was open in the original program, it will be open in the new program or process.

A new program also inherits the contents of the input and output buffers used by the open files of the original program. To prevent a new program or process from reading or writing data that is not intended for its use, these buffers should be flushed before calling the program or creating the new process. A program can flush an output buffer with the *fflush* function, and an input buffer with *setbuf*.

5.10 Program Example

This section shows how to use the process control functions to control a simple process. The following program starts a shell on the terminal given in the command line. The terminal is assumed to be connected to the system through a line that has not been enabled for multiuser operation.

Chapter 6
Creating and Using Pipes

6.1 Introduction

A pipe is an artifical file that a program may create and use to pass information to other programs. A pipe is similar to a file in that it has a file pointer and/or a file descriptor and can be read from or written to using the input and output functions of the standard library. Unlike a file, a pipe does not represent a specific file or device. Instead a pipe represents temporary storage in memory that is independent of the program's own memory and is controlled entirely by the system.

Pipes are chiefly used to pass information between programs, just as the shell pipe symbol (|), is used to pass the output of one program to the input of another. This eliminates the need to create temporary files to pass information to other programs. A pipe can also be used as a temporary storage place for a single program. A program can write to the pipe, then read that information back at a later time.

The standard library provides several pipe functions. The *popen* and *pclose* functions control both a pipe and a process. The *popen* function opens a pipe and creates a new process at the same time, making the new pipe the standard input or output of the new process. The *pclose* function closes the pipe and waits for termination of the corresponding process. The *pipe* function, on the other hand, gives low-level access to a pipe. The function is similar to the *open* function, but opens the pipe for both reading and writing, returning two file descriptors instead of one. The program can either use both sides of the pipe or close the one it does not need. The low-level input and output functions *read* and *write* can be used to read from and write to a pipe. Pipe file descriptors are used in the same way as other file descriptors.

6.2 Opening a Pipe to a New Process

The *popen* function creates a new process and then opens a pipe to the standard input or output file of that new process. The function call has the form:

popen (*command, type*)

where *command* is a pointer to a string that contains a shell command line, and *type* is a pointer to the string which defines whether the pipe is to be opened for reading or writing by the original process. It may be "r" for reading or "w" for writing. The function normally returns the file pointer to the open pipe, but will return the null pointer value NULL if an error is encountered.

The function is typically used in programs that need to call another program and pass substantial amounts of data to that program. For example, in the following program fragment *popen* creates a new process for the cat command and opens a pipe for writing.

where *stream* is the file pointer of the pipe to be closed. The function normally returns the exit status of the command that was issued as the first argument of its corresponding *popen*, but will return the value –1 if the pipe was not opened by *popen*.

For example, in the following program fragment *pclose* closes the pipe given by "pstrm" if the end-of-file value EOF has been found in the pipe.

```
FILE *pstrm;

if (feof(pstrm))
        pclose (pstrm);
```

6.5 Opening a Low-Level Pipe

The *pipe* function opens a pipe for both reading and writing. The function call has the form:

```
pipe (fd)
```

where *fd* is a pointer to a two-element array. It must have **int** type. Each element receives one file descriptor. The first element receives the file descriptor for the reading side of the pipe, and the other element receives the file descriptor for the writing side. The function normally returns 0, but will return the value –1 if an error is encountered. For example, in the following program fragment *pipe* creates two file descriptors if no error is encountered.

```
int chan[2];

if (pipe(chan) == -1)
        exit(2);
```

The array element "chan[0]" receives the file descriptor for the reading side of the pipe, and "chan[1]" receives it for the writing side.

The function is typically used to open a pipe in preparation for linking it to a child process. For example, in the following program fragment *pipe* causes the program to create a child process if it successfully creates a pipe.

```
int  fd[2];

if (pipe(fd) != -1)
        if ( fork() == 0 )
                close(fd[1]);
```

Note that the child process closes the writing side of the pipe. The parent can now pass data to the child by writing to the pipe; the child can retrieve the data by reading the pipe.

The system copies the end-of-file value EOF to a pipe when the process that made the original pipe and every process created or called by that process has closed the writing side of the pipe. This means, for example, that if a parent process is sending data to a child process through a pipe and closes the pipe to signal the end of the file, the child process will not receive the end-of-file value unless it has already closed its own write side of the pipe.

6.8 Program Examples

This section shows how to use the process control functions with the low-level *pipe* function to create functions similar to the *popen* and *pclose* functions.

The first example is a modified version of the *popen* function. The modified function identifies the new pipe with a file descriptor rather than a file pointer. It also requires a "mode" argument rather than a "type" argument, where the mode is 0 for reading or 1 for writing.

```
#include <stdio.h>

#define  READ   0
#define  WRITE  1
#define  tst(a, b) (mode == READ ? (b) : (a))
static   int       popen_pid;

popen(cmd, mode)
char     *cmd;
int      mode;
{
         int p[2];

         if (pipe(p) < 0)
                  return(NULL);

         if ((popen_pid = fork()) == 0) {
                  close(tst(p[WRITE], p[READ]));
                  close(tst(0, 1));
                  dup(tst(p[READ], p[WRITE]));
                  close(tst(p[READ], p[WRITE]));
                  execl("/bin/sh", "sh", "-c", cmd, 0);
                  exit(1);    /* sh cannot be found */
         }
         if (popen_pid == -1)
                  return(NULL);

         close(tst(p[READ], p[WRITE]));
         return(tst(p[WRITE], p[READ]));
}
```

The function creates a pipe with the *pipe* function first. It then uses the *fork*

```
#include <signal.h>

pclose(fd)          /* close pipe fd */
int fd;
{
        int r, status;
        int (*hstat)(), (*istat)(), (*qstat)();
        extern int popen_pid;

        close(fd);

        istat = signal(SIGINT, SIG_IGN);
        qstat = signal(SIGQUIT, SIG_IGN);
        hstat = signal(SIGHUP, SIG_IGN);

        while ((r = wait(&status)) != popen_pid && r != -1)
                ;
        if (r == -1)
                status = -1;

        signal(SIGINT, istat);
        signal(SIGQUIT, qstat);
        signal(SIGHUP, hstat);

        return(status);
}
```

The function closes the pipe first. It then uses a **while** statement to wait for the child process given by "popen_pid". If other child processes terminate while it waits, it ignores them and continues to wait for the given process. It stops waiting as soon as the given process terminates or if no child process exists. The function returns the termination status of the child, or the value -1 if there was an error.

The *signal* function calls used in this example ensure that no interrupts interfere with the waiting process. The first set of functions causes the process to ignore the interrupt, quit, and hang up signals. The last set restores the signals to their original status. The *signal* function is described in detail in Chapter 7, "Using Signals".

Note that both example functions use the external variable "popen_pid" to store the process ID of the child process. If more than one pipe is to be opened, the "popen_pid" value must be saved in another variable before each call to *popen*, and this value must be restored before calling *pclose* to close the pipe. The functions can be modified to support more than one pipe by changing the "popen_pid" variable to an array indexed by file descriptor.

Chapter 7
Using Signals

7.1 Introduction

This chapter explains how to use C library functions to process signals sent to a program by the XENIX system. A signal is the system's response to an unusual condition that occurs during execution of a program such as a user pressing the INTERRUPT key or the system detecting an illegal operation. A signal interrupts normal execution of the program and initiates an action such as terminating the program or displaying an error message.

The *signal* function of the standard C library lets a program define the action of a signal. The function can be used to disable a signal to prevent it from affecting the program. It can also be used to give a signal a user-defined action.

The *signal* function is often used with the *setjmp* and *longjmp* functions to redefine and reshape the action of a signal. These functions allow programs to save and restore the execution state of a program, giving a program a means to jump from one state of execution to another without a complex assembly language interface.

To use the *signal* function, you must add the line

```
#include <signal.h>
```

to the beginning of the program. The *signal.h* file defines the various manifest constants used as arguments by the function. To use the *setjmp* and *longjmp* functions you must add the line

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

to the beginning of the program. The *setjmp.h* file contains the declaration for the type **jmp_buf**, a template for saving a program's current execution state.

7.2 Using the *signal* Function

The *signal* function changes the action of a signal from its current action to a given action. The function has the form

signal (*sigtype*, *ptr*)

where *sigtype* is an integer or a mainfest constant that defines the signal to be changed, and *ptr* is a pointer to the function defining the new action or a manifest constant giving a predefined action. The function always returns a pointer value. This pointer defines the signal's previous action and may be used in subsequent calls to restore the signal to its previous value.

The *ptr* may be "SIG_IGN" to indicate no action (ignore the signal) or "SIG_DFL" to indicate the default action. The *sigtype* may be "SIGINT" for interrupt signal, caused by pressing the INTERRUPT key, "SIGQUIT" for quit

```
#include <signal.h>

main ()
{

        if ( fork() == 0) {
                signal(SIGINT, SIG_IGN);
                /* Child process. */

        }

/* Parent process. */

}
```

This call does not affect the parent process which continues to receive interrupts as before. Note that if the parent process is interrupted, the child process continues to execute until it reaches its normal end.

7.2.2 Restoring a Signal's Default Action

You can restore a signal to its default action by using the "SIG_DFL" constant with *signal*. The function call has the form

signal (*sigtype*, SIGDFL)

where *sigtype* is the manifest constant defining the signal you wish to restore. For example, the function call

signal (SIGINT, SIG_DFL)

restores the interrupt signal to its default action.

The function call is typically used to restore a signal after it has been temporarily disabled to keep it from interrupting critical operations. For example, in the following program fragment the second call to *signal* restores the signal to its default action.

```
#include <signal.h>

main ()
{
        int catch ();

        printf("Press INTERRUPT key to stop.0);
        signal (SIGINT, catch);
        while () {
                /* Body */
        }
}

catch ()
{
        printf("Program terminated.\n");
        exit(1);
}
```

The *catch* function prints the message "Program terminated" before stopping the program with the *exit* function.

A program may redefine the action of a signal at any time. Thus, many programs define different actions for different conditions. For example, in the following program fragment the action of the interrupt signal depends on the return value of a function named *keytest*.

```
#include <signal.h>

main ()
{
        int catch1 (), catch2 ();

        if (keytest() == 1)
                signal(SIGINT, catch1);
        else
                signal(SIGINT, catch2);

}
```

Later the program may change the signal to the other action or even a third action.

When using a function pointer in the *signal* call, you must make sure that the function name is defined before the call. In the program fragment shown above, *catch1* and *catch2* are explicitly declared at the beginning of the main program function. Their formal definitions are assumed to appear after the *signal* call.

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

system(s)           /* run command string s */
char *s;
{
        int status, pid, w;
        register int (*istat)(), (*qstat)();

        if ((pid = fork()) == 0) {
                execl("/bin/sh", "sh", "-c", s, NULL);
                exit(127);
        }
        istat = signal(SIGINT, SIG_IGN);
        qstat = signal(SIGQUIT, SIG_IGN);
        while ((w = wait(&status)) != pid && w != -1)
                ;
        if (w == -1)
                status = -1;
        signal(SIGINT, istat);
        signal(SIGQUIT, qstat);
        return(status);
}
```

Note that the parent uses the **while** statement to wait until the child's process ID "pid" is returned by *wait*. If *wait* returns the error code "–1" no more child processes are left, so the parent returns the error code as its own status.

7.3 Controlling Execution With Signals

Signals do not need to be used solely as a means of immediately terminating a program. Many signals can be redefined to delay their actions or even cause actions that terminate a portion of a program without terminating the entire program. The following sections describe ways that signals can be caught and used to provide control of a program.

7.3.1 Delaying a Signal's Action

You can delay the action of a signal by catching the signal and redefining its action to be nothing more than setting a globally-defined flag. Such a signal does nothing to the current execution of the program. Instead, the program continues uninterrupted until it can test the flag to see if a signal has been received. It can then respond according to the value of the flag.

The key to a delayed signal is that all functions return execution the exact point at which the program was interrupted. If the function returns normally the program continues execution just as if no signal occurred.

receives a signal when reading the terminal, all characters read before the interruption are lost, making it appear as though no characters were typed.

Whenever a program intends to use delayed signals during calls to system functions, the program should include a check of the function return values to ensure that an error was not caused by an interruption. In the following program fragment, the program checks the current value of the interrupt flag "intflag" to make sure that the value EOF returned by *getchar* actually indicates the end of the file.

```
if (getchar() == EOF)
        if (intflag)
                /* EOF caused by interrupt */
        else
                /* true end-of-file */
```

7.3.3 Using Signals in Interactive Programs

Signals can be used in interactive programs to control the execution of the program's various commands and operations. For example, a signal may be used in a text editor to interrupt the current operation (e.g., displaying a file) and return the program to a previous operation (e.g., waiting for a command).

To provide this control, the function that redefines the signal's action must be able to return execution of the program to a meaningful location, not just the point of interruption. The standard C library provides two functions to do this: *setjmp* and *longjmp*. The *setjmp* function saves a copy of a program's execution state. The *longjmp* function changes the current execution state to a previously saved state. The functions cause a program to continue execution at an old location with old register values and status as if no operations had been performed between the time the state was saved and the time it was restored.

The *setjmp* function has the form

```
setjmp (buffer)
```

where *buffer* is the variable to receive the execution state. It must be explicitly declared with type **jmpbuf** before it is used in the call. For example, in the following program fragment *setbuf* copies the execution of the program to the variable "oldstate" defined with type **jmpbuf**.

```
jmpbuf oldstate;

setbuf(oldstate);
```

Note that after a *setbuf* call, the *buffer* variable contains values for the program counter, the data and address registers, and the process status. These values must not be modified in any way.

7.4.1 Protecting Background Processes

Any program that has been invoked using the shell's background symbol (&) is executed as a background process. Such programs usually do not use the terminal for input or output, and complete their tasks silently. Since these programs do not need additional input, the shell automatically disables the signals before executing the program. This means signals generated at the terminal do not affect execution of the program. This is how the shell protects the program from signals intended for other programs invoked from the same terminal.

In some cases, a program that has been invoked as a background process may also attempt to catch its own signals. If it succeeds, the protection from interruption given to it by the shell is defeated, and signals intended for other programs will interrupt the program. To prevent this, any program which is intended to be executed as a background process, should test the current state of a signal before redefining its action. A program should redefine a signal only if the signal has not been disabled. For example, in the following program fragment the action of the interrupt signal is changed only if the signal is not currently being ignored.

```
#include <signal.h>

main()
{
        int catch();

        if (signal(SIGINT, SIG_IGN) != SIG_IGN)
                signal(SIGINT, catch);

        /* Program body. */
}
```

This step lets a program continue to ignore signals if it is already doing so, and change the signal if it is not.

Chapter 8
Using System Resources

8.1 Introduction

This chapter describes the standard C library functions that let programs share the resources of the XENIX system. The functions give a program the means to queue for the use and control of a given resource and to synchronize its use with use by other programs.

In particular, this chapter explains how to

- Allocate memory for dynamically required storage
- Lock a file to ensure exclusive use by a program
- Use semaphores to control access to a resource
- Share data space to allow interaction between programs

8.2 Allocating Space

Some programs require significant changes to the size of their allocated memory space during different phases of their execution. The memory allocation functions of the standard C library let programs allocate space dynamically. This means a program can request a given number of bytes of storage for its exclusive use at the moment it needs the space, then free this space after it has finished using it.

There are four memory allocation functions: *malloc*, *calloc*, *ralloc*, and *free*. The *malloc* and *calloc* functions are used to allocate space for the first time. The functions allocate a given number of bytes and return a pointer to the new space. The *realloc* function reallocates an existing space, allowing it to be used in a different way. The *free* function returns allocated space to the system.

8.2.1 Allocating Space for a Variable

The *malloc* function allocates space for a variable containing a given number of bytes. The function call has the form:

malloc (*size*)

where *size* is an unsigned number which gives the number of bytes to be allocated. For example, the function call

table = malloc (4)

allocates four bytes or storage. The function normally returns a pointer to the starting address of the allocated space, but will return the null pointer value if there is not enough space to allocate.

8.2.3 Reallocating Space

The *realloc* function reallocates the space at a given address without changing the contents of the memory space. The function call has the form:

realloc (*ptr*, *size*)

where *ptr* is a pointer to the starting address of the space to be reallocated, and *size* is an unsigned number giving the new size in bytes of the reallocated space. The function normally returns a pointer to the starting address of the allocated space, but will return a null pointer value if there is not enough space to allocate.

This function is typically used to keep storage as compact as possible. For example, in the following program fragment *realloc* is used to remove table entries.

```
main ()
{
char *table;
int i;
unsigned inum;

for (i=inum; i>-1; i--) {
        printf("%d0, strings[i]);
        strings = realloc(strings, i*4);
        }
```

In this example, an entry is removed after it has been printed at the standard output, by reducing the size of the allocated space from its current length to the length given by "i*4".

8.2.4 Freeing Unused Space

The *free* function frees unused memory space that had been previously allocated by a *malloc*, *calloc*, or *realloc* function call. The function call has the form:

free (*ptr*)

where *ptr* is the pointer to the starting address of the space to be freed. This pointer must be the return value of a *malloc*, *calloc*, or *realloc* function.

The function is used exclusively to free space which is no longer used or to free space to be used for other purposes. For example, in the following program fragment *free* frees the allocated space pointed to by "strings" if the first element is equal to zero.

8.3.2 Locking a File

The *locking* function locks one or more bytes of a given file. The function call has the form:

locking (*filedes*, *mode*, *size*)

where *filedes* is the file descriptor of the file to be locked, *mode* is an integer value which defines the type of lock to be applied to the file, *size* is a long integer value giving the size in bytes of the portion of the file section to be locked or unlocked. The *mode* may be "LOCK" for locking the given bytes, or "UNLOCK" for unlocking them. For example, in the following program fragment *locking* locks 100 bytes at the current character pointer position in the file given by "fd".

```
#include <sys/locking.h>

main ()
{
int fd,

fd = open("data", 2);
locking(fd, LOCK, 100);
```

The function normally returns the number of bytes locked, but will return –1 if it encounters an error.

8.3.3 Program Example

This section shows how to lock and unlock a small section in a file using the *locking* function. In the following program, the function locks 100 bytes in the file *data* which is opened for reading and writing. The locked portion of the file is accessed, then *locking* is used again to unlock the file.

8.4.1 Creating a Semaphore

The *creatsem* function creates a semaphore, returning a semaphore number which may be used in subsequent semaphore functions. The function call has the form:

creatsem (*sem_name*, *mode*)

where *sem_name* is a character pointer to the name of the semaphore, and *mode* is an integer value which defines the access mode of the semaphore. Semaphore names have the same syntax as regular file names. The names must be unique. The function normally returns an integer semaphore number which may be used in subsequent semaphore functions to refer to the semaphore. The function returns –1 if it encounters an error, such as creating a semaphore that already exists, or using the name of an existing regular file.

The function is typically used at the beginning of one process to clearly define the semaphores it intends to share with other processes. For example, in the following program fragment *creatsem* creates a semaphore named "tty1" before preceding with its tasks.

```
main ()
{
int tty1;
FILE ftty1;

tty1 = creatsem("tty1", 0777);
ftty1 = fopen("/dev/tty01", "w");
        /* Program body. */
}
```

Note that *fopen* is used immediately after *creatsem* to open the file */dev/tty01* for writing. This is one way to make the association between a semaphore and a device clear.

The mode "0777" defines the semaphore's access permissions. The permissions are similar to the permissions of a regular file. A semaphore may have read permission for the owner, for users in the same group as the owner, and for all other users. The write and execution permissions have no meaning. Thus, "0777" means read permission for all users.

No more than one process ever need create a given semaphore; all other processes simply open the semaphore with the *opensem* function. Once created or opened, a semaphore may be accessed only by using the *waitsem*, *nbwaitsem*, or *sigsem* functions. The *creatsem* function may be used more than once during execution of a process. In particular, it can be used to reset a semaphore if a process fails to relinquish control before terminating.

semaphore that does not exist or requesting a semaphore that is locked to a dead process.

The function is used whenever a given process wishes to access the device or system resource associated with the semaphore. For example, in the following program fragment *waitsem* signals the intention to write to the file given by "tty1".

```
main ()
{
int tty1;
FILE ftty1;

waitsem( tty1 );
fprintf( ftty1, "Changing tty driver\n");
```

The function waits until current controlling process relinquishes control of the semaphore before returning to the next statement.

8.4.4 Checking the Status of a Semaphore

The *nbwaitsem* function checks the current status of a semaphore. If the semaphore is not available, the function returns an error value. Otherwise, it gives immediate control of the semaphore to the calling process. The function call has the form:

nbwaitsem (*sem_num*)

where *sem_num* is the semaphore number of the semaphore to be checked. The function returns –1 if it encounters an error such as requesting a semaphore that does not exist. The function also returns –1 if the process controlling the requested semaphore terminates without relinquishing control of the semaphore.

The function is typically used in place of *waitsem* to take control of a semaphore.

8.4.5 Relinquishing Control of a Semaphore

The *sigsem* function causes a process to relinquish control of a given semaphore and to signal this fact to all processes waiting for the semaphore. The function call has the form:

sigsem (*sem_num*)

where *sem_num* is the semaphore number of the semaphore to relinquish. The semaphore must have been previously created or opened by the process. Furthermore, the process must have been previously taken control of the

```
#define NPROC 5

char    semf[] = "_kesemfXXXXXX";
int     sem_num;
int     holdsem = 5;

main()
{
        register i, chid;

        mktemp(semf);
        if ((sem_num = creatsem(semf, 0777)) < 0)
                err("creatsem");
        for (i = 1; i < NPROC; ++i) {
                if((chid = fork()) < 0)
                        err("No fork");
                else if(chid == 0) {
                        if((sem_num = opensem(semf)) < 0)
                                err("opensem");
                        doit(i);
                        exit(0);
                }
        }
        doit(0);
        for (i = 1; i < NPROC; ++i)
                while(wait((int *)0) < 0)
                        ;
        unlink(semf);
}

doit(id)
{
        while(holdsem--) {
                if(waitsem(sem_num) < 0)
                        err("waitsem");
                printf("%d\n", id);
                sleep(1);
                if(sigsem(sem_num) < 0)
                        err("sigsem");
        }
}

err(s)
char *s;
{
        perror(s);
        exit(1);
}
```

8.5.1 Creating a Shared Data Segment

The *sdget* function creates a shared data segment for the current process, or if the segment already exists, attaches the segment to the data space of the current process. The function call has the form:

sdget (*path*, *flag* [, *size*, *mode*])

where *path* is a character pointer to a valid pathname, *flag* is an integer value which defines how the segment should be created or attached, *size* is an integer value which defines the size in bytes of the segment to be created, and *mode* is an integer value which defines the access permissions to be given to the segment if created. The *size* and *mode* values are used only when creating a segment. The *flag* may be SD_RDONLY for attaching the segment for reading only, SD_WRITE for attaching the segment for reading and writing, SD_CREAT for creating the segment given by *path* if it does not already exist, or SD_UNLOCK for allowing simultaneous access by multiple processes. The values can be combined by logically ORing them. The SD_UNLOCK value is used only if the segment is created. The function returns the address of the segment if it has been successfully created or attached. Otherwise, the function returns -1 if it encounters an error.

The function is most often used to create a segment to be shared by another process. The function may then be used in the other process to attach the segment to its data space. For example, in the following program fragment *sdget* creates a segment and assigns the address of the segment to "shared".

```
#include <sd.h>

main ()
{
char *shared, *spath;

shared = sdget( spath, SD_CREAT, 512, 0777 );
}
```

When the segment is created, the size "512" and the mode "0777" are used to define the segment's size in bytes and access permissions. Access permissions are similar to permissions given to regular files. A segment may have read or write permission for the owner of the process, for users belonging to the same group as the owner, and for all other users. Execute permission for a segment has no meaning. For example, the mode "0777" means read and write permission for everyone, but "0660" means read and write permissions for the owner and group processes only. When first created, a segment is filled with zeroes.

Note that the SD_UNLOCK flag used on systems without hardware support for shared data may severely degrade the execution performance of the program.

```
#include <sd.h>

main ()
{
char *shared;

while ( *x++ != 0 ) {
        sdenter(shared);
        /* write to segment */
        sdleave(shared);
        }
}
```

8.5.4 Getting the Current Version Number

The *sdgetv* function returns the current version number of the given data segment. The function call has the form:

```
sdgetv (addr)
```

where *addr* is a character pointer to the desired segment. A segment's version number is initially zero, but it is incremented by one whenever a process leaves the segment using the *sdleave* function. Thus, the version number is a record of the number of times the segment has been accessed. The function's return value is always an integer. It returns -1 if it encounters an error.

The function is typically used to choose an action based on the current version number of the segment. For example, in the following program fragment *sdgetv* determines whether or not *sdenter* should be used to enter the segment given by "shared".

```
#$include <sd.h>

main ()
{
char *shared;

if (sdgetv(shared) > 10)
        sdenter(shared);
```

In this example, the segment is entered if the current version number of the segment is greater than "10".

8.5.5 Waiting for a Version Number

The *sdwaitv* function causes a process to wait until the version number for the given segment is no longer equal to a given version number. The function call

Chapter 9
Error Processing

9.1 Introduction

The XENIX system automatically detects and reports errors that occur when using standard C library functions. Errors range from problems with accessing files to allocating memory. In most cases, the system simply reports the error and lets the program decide how to respond. The XENIX system terminates a program only if a serious error has occurred, such as a violation of memory space.

This chapter explains how to process errors, and describes the functions and variables a program may use respond to errors.

9.2 Using the Standard Error File

The standard error file is a special output file that can be used by a program to display error messages. The standard error file is one of three standard files (standard input, output, and error) automatically created for the program when it is invoked.

The standard error file, like the standard output, is normally assigned to the user's terminal screen. Thus, error messages written to the file are displayed at the screen. The file can also be redirected by using the shell's redirection symbol (>) For example, the following command redirects the standard error file to the file *errorlist*.

```
dial 2>errorlist
```

In this case, subsequent error messages are written to the given file.

The standard error file, like the standard input and standard output, has predefined file pointer and file descriptor values. The file pointer **stderr** may be used in stream functions to copy data to the error file. The file descriptor **2** may be used in low-level functions to copy data to the file. For example, in the following program fragment **stderr** is used to write the message "Unexpected end of file" to the standard error file.

```
if ( (c=getchar()) == EOF)
        fprintf(stderr, "Unexpected end of file.\n");
```

The standard error file is not affected by the shell's pipe symbol (|). This means that even if the standard output of a program is piped to another program, errors generated by the program will still appear at the terminal screen (or in the appropriate file if the standard error is redirected).

9.3 Using the errno Variable

The errno variable is a predefined external variable which contains the error

accounts: Permission denied.

if **errno** is equal to the constant EACCES.

```
if ( errno == EACCES ) {
        perror("accounts");
        fd = open ("/usr/tmp/accounts", O_RDONLY);
}
```

All error messages displayed by *perror* are stored in an array named sys_errno, an external array of character strings. The *perror* function uses the variable **errno** as the index to the array element containing the desired message.

9.5 Using Error Signals

Some program errors cause the XENIX system to generate error signals. These signals are passed back to the program that caused the error and normally terminate the program. The most common error signals are SIGBUS, the bus error signal, SIGFPE, the floating point exception signal, SIGSEGV, the segment violation signal, SIGSYS, the system call error signal, and SIGPIPE, the pipe error signal. Other signals are described in *signal*(S) in the XENIX *Reference Manual*.

A program can, if necessary, catch an error signal and perform its own error processing by using the *signal* function. This function, as described in Chapter 7, "Using Signals" can set the action of a signal to a user-defined action. For example, the function call

```
signal(SIGBUS, fixbus);
```

sets the action of the bus error signal to the action defined by the user-supplied function *fixbus*. Such a function usually attempts to remedy the problem, or at least display detailed information about the problem before terminating the program.

For details about how to catch, redefine, and restore these signals, see Chapter 7.

9.6 Encountering System Errors

Programs that encounter serious errors, such as hardware failures or internal errors, generally do not receive detailed reports on the cause of the errors. Instead, the XENIX system treats these errors as "system errors", and reports them by displaying a system error message on the system console. This section briefly describes some aspects of XENIX system errors and how they relate to user programs. For a complete list and description of XENIX system errors, see *messages*(M) in the XENIX *Reference Manual*.

Appendix A
Assembly Language Interface

A.1 Introduction

When mixing MC68000 assembly language routines and compiled C routines, there are several things to be aware of:

- Registers and Return Values
- Calling Sequence
- Stack Probes

With an understanding of these three topics, you should be able to write both C programs that call MC68000 assembly language routines and assembly language routines that call compiled C routines.

A.1.1 Registers and Return Values

Function return values are passed in registers if possible. The set of machine registers used is called the *save set*, and includes the registers from *d2 – d7* and *a2 – a7* that are modified by a routine. The compiler assumes that these registers are preserved by the callee, and saves them itself when it is generating code for the callee (when a C compatible routine is called by another routine, we refer to the calling routine as the *caller*. We refer to the called routine as the *callee*.) Note that *a6* and *a7* are in effect saved by a link instruction at procedure entry.

The function return value is in *d0*. The current floating point implementation returns the high order 32 bits of doubles in *d1*, and the low order 32 bits in *d0*. Functions that return structure values (not pointers to the values) do so by loading *d0* with a pointer to a static buffer containing the structure value.

This makes the following two functions equivalent:

```
struct foo proc (){
        struct foo this;
        ...
        return (this);
}

struct foo *proc (){
        struct foo this;
        static struct foo temp;
        ...
        temp = this;
        return (&temp);
}
```

This implementation allows recursive reentrancy (as long as the explicit form is not used, since the first sequence is indivisible but not the second). However, this implementation does *not* permit multitasking reentrancy. Note that the latter includes the XENIX *signal*(3) call.

Setjmp(3) and *longjmp*(3) can not be implemented as they are on the PDP–11, because each procedure saves only the registers from the save set that it will modify. This makes it difficult to get back the current values of the register variables of the

bytes pushed as temporaries, save areas, and arguments by the whole procedure. The 8 bytes are the space for the return address and frame pointer save (by the link instruction) of a nested call. The *slop* is tolerance so that extremely short runtimes that use little stack do not need to perform a stack probe. The tolerance is intentionally kept small to conserve memory, so make sure you understand what you are doing before you consider leaving out a stack probe in your assembly procedures.

In most cases, unless you are pushing huge structures or doing tricks with the stack within your procedure, you can use the following instruction for your stack probe:

```
tstb    -100(sp)
```

This makes sure that enough space has been allocated for most of the usual things you might do with the stack and is enough for the XENIX runtimes that do not perform stack probes. Note that you do not need to consider space allocated by the link instruction in this stack probe, since it is already added by indexing off the stack pointer.

Appendix B
XENIX System Calls

B.1 Introduction

This appendix lists some of the differences between XENIX 2.3, XENIX 3.0, UNIX V7, and UNIX System 3.0. It is intended to aid users who wish to convert system calls in existing application programs for use on other systems.

B.2 Executable File Format

Both XENIX 3.0 and UNIX System 3.0 execute only those programs with the *x.out* executable file format. The format is similar to the old *a.out* format, but contains additional information about the executable file, such as text and data relocation bases, target machine identification, word and byte ordering, and symbol table and relocation table format. The *x.out* file also contains the revision number of the kernel which is used during execution to control access to system functions. To execute existing programs in *a.out* format, you must first convert to the *x.out* format. The format is described in detail in *a.out*(F) in the XENIX *Reference Manual*.

B.3 Revised System Calls

Some system calls in XENIX 3.0 and UNIX System 3.0 have been revised and do not perform the same tasks as the corresponding calls in previous systems. To provide compatibilty for old programs, XENIX 3.0 and UNIX System 3.0 maintain both the new and the old system calls and automatically check the revision information in the *x.out* header to determine which version of a system call should be made. The following table lists the revised system calls and their previous versions.

System Call #	XENIX 2.3 function	System 3 function
35	ftime	unused
38	unused	clocal
39	unused	setpgrp
40	unused	cxenix
57	unused	utssys
62	clocal	fcntl
63	cxenix	ulimit

The *cxenix* function provides access to system calls unique to XENIX System 3.0. The *clocal* function provides access to all calls unique to an OEM.

B.4 Version 7 Additions

XENIX 3.0 maintains a number of UNIX V7 features that were dropped from UNIX System 3.0. In particular, XENIX 3.0 continues to support the *dup2* and

B.9 Separate Version Libraries

XENIX 3.0 and UNIX System 3.0 support the construction of XENIX 2.3 executable files. These systems maintain both the new and old versions of system calls in separate libraries and include files.

Index

CONTENTS

Programming Commands (CP)

Index

Name

intro – Introduces XENIX Software Development commands.

Description

This section describes use of the individual commands available in the XENIX Software Development System. Each individual command is labeled with the letters CP to distinguish it from commands available in the XENIX Timesharing and Text Processing Systems. These letters are used for easy reference from other documentation. For example, the reference *cc*(CP) indicates a reference to a discussion of the cc command in this section, where the letter "C" stands for "command" and the letter "P" stands for "Programming".

Syntax

Unless otherwise noted, commands described in this section accept options and other arguments according to the following syntax:

name [*options*] [*cmdarg*]

where:

name The filename or pathname of an executable file

option A single letter representing a command option By convention, most options are preceded with a dash. Option letters can sometimes be grouped together as in – abcd or alternatively they are specified individually as in – a – b – c – d . The method of specifying options depends on the syntax of the individual command. In the latter method of specifying options, arguments can be given to the options. For example, the – f option for many commands often takes a following filename argument.

cmdarg A pathname or other command argument *not* beginning with a dash. It may also be a dash alone by itself indicating the standard input.

See Also

getopt(C), getopt(S)

Diagnostics

Upon termination, each command returns 2 bytes of status, one supplied by the system and giving the cause for termination, and (in the

Name

adb – debugger

Syntax

adb [**–w**] [objfil [corfil]]

Description

Adb is a general purpose debugging program. It may be used to examine files and to provide a controlled environment for the execution of XENIX programs.

Objfil is normally an executable program file, preferably containing a symbol table; if not then the symbolic features of *adb* cannot be used although the file can still be examined. The default for *objfil* is **a.out.** *Corfil* is assumed to be a core image file produced after executing *objfil*; the default for *corfil* is **core.**

Requests to *adb* are read from the standard input and responses are to the standard output. If the **–w** flag is present then both *objfil* and *corfil* are created if necessary and opened for reading and writing so that files can be modified using *adb*. *Adb* ignores QUIT; INTERRUPT causes return to the next *adb* command.

In general requests to *adb* are of the form:

[*address*] [, *count*] [*command*] [;]

If *address* is present then *dot* is set to *address*. Initially *dot* is set to 0. For most commands *count* specifies how many times the command will be executed. The default *count* is 1. *Address* and *count* are expressions.

The interpretation of an address depends on the context it is used in. If a subprocess is being debugged then addresses are interpreted in the usual way in the address space of the subprocess. For further details of address mapping see ADDRESSES.

EXPRESSIONS

.	The value of *dot*.
+	The value of *dot* incremented by the current increment.
^	The value of *dot* decremented by the current increment.
"	The last *address* typed.
integer	If the integer begins with 0 it is an an octal number. It is a hexadecimal number if preceded by 0x or 0X. It is a decimal number when preceded by 0d, 0D, 0t, or 0T; otherwise the current input radix (default decimal).
'cccc'	The ASCII value of up to 4 characters. \ may be used to

to the format *f*.

/f Locations starting at *address* in *corfil* are printed according to the format *f*.

=f The value of *address* itself is printed in the styles indicated by the format *f*.

A *format* consists of one or more characters that specify a style of printing. Each format character may be preceded by a decimal integer that is a repeat count for the format character. While stepping through a format *dot* is incremented temporarily by the amount given for each format letter. If no format is given then the last format is used. The format letters available are as follows.

o 2 Print 2 bytes in octal. All octal numbers output by *adb* are preceded by 0.
O 4 Print 4 bytes in octal.
q 2 Print in signed octal.
Q 4 Print long signed octal.
d 2 Print in decimal.
D 4 Print long decimal.
x 2 Print 2 bytes in hexadecimal. All hexadecimal numbers output by *adb* are preceded by 0x.
X 4 Print 4 bytes in hexadecimal.
u 2 Print as an unsigned decimal number.
U 4 Print long unsigned decimal.
b 1 Print the addressed byte in octal.
c 1 Print the addressed character.
C 1 Print the addressed character using the following escape convention. Character values 000 to 040 are printed as @ followed by the corresponding character in the range 0100 to 0140. The character @ is printed as @@.
s *n* Print the addressed characters until a zero character is reached.
S *n* Print a string using the @ escape convention. *n* is the length of the string including its zero terminator.
Y 4 Print 4 bytes in date format (see *ctime*(S)).
i n Print as MC68000 instructions. *n* is the number of bytes occupied by the instruction. This style of printing causes variables 1 and 2 to be set to the offset parts of the source and destination respectively.
a 0 Print the value of *dot* in symbolic form. Symbols

! A shell is called to read the rest of the line following '!'.

$modifier

Miscellaneous commands. The available *modifiers* are:

- *<f* Read commands from the file *f* and return.
- *>f* Send output to the file *f*, which is created if it does not exist; > ends the output diversion.
- **r** Print the general registers and the instruction addressed by **pc**. *Dot* is set to **pc**.
- **b** Print all breakpoints and their associated counts and commands.
- **c** C stack backtrace. If *address* is given then it is taken as the address of the current frame. (instead of a6). If *count* is given then only the first *count* frames are printed.
- **e** The names and values of external variables are printed.
- **w** Set the page width for output to *address* (default 80).
- **s** Set the limit for symbol matches to *address* (default 255).
- **o** Set the current input radix to octal.
- **d** Set the current input radix to decimal. EXPRES– SIONS.
- **x** Set the current input radix to hexadecimal.
- **q** Exit from *adb*.
- **v** Print all non zero variables in hexadecimal.
- **m** Print the address map.

:modifier

Manage a subprocess. Available modifiers are:

- **bc** Set breakpoint at *address*. The breakpoint is executed *count*−1 times before causing a stop. Each time the breakpoint is encountered the command *c* is executed. If this command sets *dot* to zero then the breakpoint causes a stop.
- **d** Delete breakpoint at *address*.
- **r** Run *objfil* as a subprocess. If *address* is given explicitly then the program is entered at this point; otherwise the program is entered at its standard entry point. *count* specifies how many breakpoints are to be ignored before stopping. Arguments to the subprocess may be supplied on

b2≤*address*<*e2* => *file address*=*address*+*f2*−*b2*,

otherwise, the requested *address* is not legal. In some cases (e.g., for programs with separated I and D space) the two segments for a file may overlap. If a ? or / is followed by an * then only the second triple is used.

The initial setting of both mappings is suitable for normal **a.out** and **core** files. If either file is not of the kind expected then, for that file, *b1* is set to 0, *e1* is set to the maximum file size and *f1* is set to 0; in this way the whole file can be examined with no address translation.

So that *adb* may be used on large files all appropriate values are kept as signed 32 bit integers.

Files

/dev/mem
/dev/swap
a.out
core

See Also

ptrace(S), a.out(F), core(F)

DIAGNOSTICS

The message 'adb' when there is no current command or format.
Comments about inaccessible files, syntax errors, abnormal termination of commands, etc.
Exit status is 0, unless last command failed or returned nonzero status.

Notes

A breakpoint set at the entry point is not effective on initial entry to the program.
When single stepping, system calls do not count as an executed instruction.
Local variables whose names are the same as an external variable may foul up the accessing of the external.

– r*rel* The *rel*ease into which the initial delta is inserted. This option may be used only if the – i option is also used. If the – r option is not used, the initial delta is inserted into release 1. The level of the initial delta is always 1 (by default initial deltas are named 1.1).

– t[*name*] The *name* of a file from which descriptive text for the SCCS file is to be taken. If the – t option is used and *admin* is creating a new SCCS file (the – n and/or – i options also used), the descriptive text filename must also be supplied. In the case of existing SCCS files: a – t option without a filename causes removal of descriptive text (if any) currently in the SCCS file, and a – t option with a filename causes text (if any) in the named file to replace the descriptive text (if any) currently in the SCCS file.

– f*flag* This option specifies a *flag*, and possibly a value for the *flag*, to be placed in the SCCS file. Several f options may be supplied on a single *admin* command line. The allowable *flags* and their values are:

b Allows use of the – b option on a *get*(CP) command to create branch deltas.

c*ceil* The highest release (i.e., "ceiling"), a number less than or equal to 9999, which may be retrieved by a *get*(CP) command for editing. The default value for an unspecified c flag is 9999.

f*floor* The lowest release (i.e., "floor"), a number greater than 0 but less than 9999, which may be retrieved by a *get*(CP) command for editing. The default value for an unspecified f flag is 1.

d*SID* The default delta number (SID) to be used by a *get*(CP) command.

i Causes the "No id keywords (ge6)" message issued by *get*(CP) or *delta*(CP) to be treated as a fatal error. In the absence of this flag, the message is only a warning. The message is issued if no SCCS identification keywords (see *get*(CP)) are found in the text retrieved or stored in the SCCS file.

j Allows concurrent *get*(CP) commands for editing on the same SID of an SCCS file. This allows multiple concurrent updates to the same version of the SCCS file.

l*list* A *list* of releases to be "unlocked". See the – f option for a description of the l flag and the syntax of a *list*.

– a*login* A *login* name, or numerical XENIX group ID, to be added to the list of users which may make deltas (changes) to the SCCS file. A group ID is equivalent to specifying all *login* names common to that group ID. Several a options may be used on a single *admin* command line. As many *logins*, or numerical group IDs, as desired may be on the list simultaneously. If the list of users is empty, then anyone may add deltas.

– e*login* A *login* name, or numerical group ID, to be erased from the list of users allowed to make deltas (changes) to the SCCS file. Specifying a group ID is equivalent to specifying all *login* names common to that group ID. Several e options may be used on a single *admin* command line.

– y[*comment*] The *comment* text is inserted into the SCCS file as a comment for the initial delta in a manner identical to that of *delta*(CP). Omission of the – y option results in a default comment line being inserted in the form:

YY/MM/DD HH:MM:SS by *login*

The – y option is valid only if the – i and/or – n options are specified (i.e., a new SCCS file is being created).

– m[*mrlist*] The list of Modification Requests (MR) numbers is inserted into the SCCS file as the reason for creating the initial delta in a manner identical to *delta*(CP). The v flag must be set and the MR numbers are validated if the v flag has a value (the name of an MR number validation program). Diagnostics will occur if the v flag is not set or MR validation fails.

– h Causes *admin* to check the structure of the SCCS file (see *sccsfile*(F)), and to compare a newly computed checksum (the sum of all the characters in the SCCS file except those in the first line) with the checksum that is stored in the first line of the SCCS file. Appropriate error diagnostics are produced.

This option inhibits writing on the file, nullifying the effect of any other options supplied, and is therefore only meaningful when processing existing files.

Name

ar – Maintains archives and libraries.

Syntax

ar key [posname] afile name ...

Description

ar maintains groups of files combined into a single archive file. Its main use is to create and update library files as used by the link editor, though it can be used for any similar purpose.

When *ar* creates an archive, it always creates the header in the format of the local system.

Key is one character from the set drqtpmx, optionally concatenated with one or more of vuaibcl. *afile* is the archive file. The *names* are constituent files in the archive file. The meanings of the *key* characters are:

d Deletes the named files from the archive file.

r Replaces the named files in the archive file. If the optional character u is used with r, then only those files with modified dates later than the archive files are replaced. If an optional positioning character from the set abi is used, then the *posname* argument must be present and specifies that new files are to be placed after (a) or before (b or i) *posname*. Otherwise new files are placed at the end.

q Quickly appends the named files to the end of the archive file. Optional positioning characters are invalid. The command does not check whether the added members are already in the archive. Useful only to avoid quadratic behavior when creating a large archive piece by piece.

t Prints a table of contents of the archive file. If no names are given, all files in the archive are tabled. If names are given, only those files are tabled.

p Prints the named files in the archive.

m Moves the named files to the end of the archive. If a positioning character is present, then the *posname* argument must be present and, as in r, specifies where the files are to be moved.

x Extracts the named files. If no names are given, all files in the archive are extracted. In neither case does x alter the archive

Name

as – assembler

Syntax

as [**-l**] [**-o** objfile] [**-g**] file.s

Description

As assembles the named file. If the argument **-l** is used, an assembly listing is produced and written to *file.L*. This includes the source, the assembled code, and any assembly errors.

The output of the assembly is left on the file *objfile;* if that is omitted, *file.o* is used. If the optional **-g** flag is given, undefined symbols will be treated as externals. Arguments may appear in any order, except that **-o** must immediatly precede *objfile*. The optional flag **-e** (externals only) prevents local symbols from being extended into *objfile's* symbol table.

Files

/tmp/A68tmpr* temporary

See Also

ld(CP), nm(CP), adb(CP), a.out(F)

Name

cc – C compiler

Syntax

cc [option] ... file ...

Description

Cc is the XENIX M68000 C compiler. Arguments whose names end with '.c' are taken to be C source programs; they are compiled, and each object program is left on the file whose name is that of the source with '.o' substituted for '.c'. The '.o' file is normally deleted, however, if a single C program is compiled and loaded all at one go.

In the same way, arguments whose names end with '.s' are taken to be assembly source programs and are assembled, producing a '.o' file.

The following options are interpreted by *cc*. See *ld*(CP) for load–time options.

–c Suppress the loading phase of the compilation, and force an object file to be produced even if only one program is compiled.

–O Invoke an object–code optimizer.

–S Compile the named C programs, and leave the assembler–language output on corresponding files suffixed '.s'.

–o *output*
Name the final output file *output*. If this option is used the file 'a.out' will be left undisturbed.

–D*name*=*def*
–D*name* Define the *name* to the preprocessor, as if by '#define'. If no definition is given, the name is defined as 1.

–U*name* Remove any initial definition of *name*.

–I*dir* '#include' files whose names do not begin with '/' are always sought first in the directory of the *file* argument, then in directories named in **–I** options, then in directories on a standard list.

–t1 replace the compiler phase with a program called **c68** from the current directory.

–t2 replace the object code optimizer phase with a program called **c68o** from the current directory.

Name

cdc – Changes the delta commentary of an SCCS delta.

Syntax

cdc – rSID [– m[mrlist]] [– y[comment]] files

Description

Cdc changes the delta commentary for the *SID* specified by the – r option, of each named SCCS file.

Delta commentary is defined to be the Modification Request (MR) and comment information normally specified via the *delta*(CP) command (– m and – y options).

If a directory is named, *cdc* behaves as though each file in the directory were specified as a named file, except that nonSCCS files (last component of the pathname does not begin with s.) and unreadable files are silently ignored. If a name of – is given, the standard input is read (see Warning); each line of the standard input is taken to be the name of an SCCS file to be processed.

Arguments to *cdc*, which may appear in any order, consist of options and file names.

All the described options apply independently to each named file:

– r*SID* Used to specify the *S*CCS *I*Dentification (*SID*) string of a delta for which the delta commentary is to be changed.

– m[*mrlist*] If the SCCS file has the v flag set (see *admin*(CP)) then a list of MR numbers to be added and/or deleted in the delta commentary of the *SID* specified by the – r option *may* be supplied. A null MR list has no effect.

MR entries are added to the list of MRs in the same manner as that of *delta*(CP). In order to delete an MR, precede the MR number with the character ! (see Examples). If the MR to be deleted is currently in the list of MRs, it is removed and changed into a "comment" line. A list of all deleted MRs is placed in the comment section of the delta commentary and preceded by a comment line stating that they were deleted.

The following interactive sequence does the same thing.

```
cdc – r1.6 s.file
MRs? !bl77-54321 bl78-12345 bl79-00001
comments? trouble
```

Warning

If SCCS file names are supplied to the *cdc* command via the standard input (– on the command line), then the – m and – y options must also be used.

Files

x-file See *delta*(CP)

z-file See *delta*(CP)

See Also

admin(CP), delta(CP), get(CP), help(CP), prs(CP), sccsfile(F)

Diagnostics

Use *help*(CP) for explanations.

– s This argument causes *comb* to generate a shell procedure that will produce a report for each file giving the filename, size (in blocks) after combining, original size (also in blocks), and percentage change computed by:

100 * (original – combined) / original

Before any SCCS files are actually combined, you should use this option to determine exactly how much space is saved by the combining process.

If no options are specified, *comb* will preserve only leaf deltas and the minimal number of ancestors needed to preserve the tree.

Files

comb????? Temporary files

See Also

admin(CP), delta(CP), get(CP), help(CP), prs(CP), sccsfile(F)

Diagnostics

Use *help*(CP) for explanations.

Notes

Comb may rearrange the shape of the tree of deltas. It may not save any space; in fact, it is possible for the reconstructed file to be larger than the original.

There are certain drivers that may be provided with the system, that are actually *pseudo-device* drivers; that is, there is no real hardware associated with the driver. Drivers of this type are identified on their respective manual entries.

Second Part of *dfile*

The second part contains three different types of lines. Note that *all* specifications of this part *are required*, although their order is arbitrary.

1. *Root/pipe device specification*

   Each line has three fields:

   ```
   root    devname      minor
   pipe    devname      minor
   ```

   where *minor* is the minor device number (in octal).

2. *Swap device specification*

   One line that contains five fields as follows:

   ```
   swap   devname      minor  swplo   nswap
   ```

   where *swplo* is the lowest disk block (decimal) in the swap area and *nswap* is the number of disk blocks (decimal) in the swap area.

3. *Parameter specification*

   A number of lines of two fields each as follows (*number* is decimal):

   ```
   buffers      number
   inodes       number
   files        number
   mounts       number
   swapmap      number
   pages        number
   calls        number
   procs        number
   maxproc      number
   texts        number
   clists       number
   locks        number
   timezone     number
   daylight     0 or 1
   ```

Example

Suppose we wish to configure a system with the following devices:
one HD disk drive controller with 1 drive
one FD floppy disk drive controller with 1 driver

Diagnostics

Diagnostics are routed to the standard output and are self-explanatory.

Notes

The – t option does not know about devices that have aliases. However, the major device numbers are always correct.

n Omits column 4 (no context)

o Uses an *only* file (see above)

s Current symbol in column 3 (default)

t User-supplied temporary file

u Prints only symbols that occur exactly once

x Prints only C external symbols

1 Sorts output on column 1 (default)

2 Sorts output on column 2

3 Sorts output on column 3

Files

/usr/lib/cref/* Assembler specific files

See Also

as(CP), cc(CP), sort(C), xref(CP)

Notes

Cref inserts an ASCII DEL character into the intermediate file after the eighth character of each name that is eight or more characters long in the source file.

Credit

This utility was developed at the University of California at Berkeley and is used with permission.

– g*list* Specifies a *list* (see *get*(CP) for the definition of *list*) of deltas which are to be *ignored* when the file is accessed at the change level (SID) created by this delta.

– m[*mrlist*] If the SCCS file has the v flag set (see *admin*(CP)) then a Modification Request (MR) number *must* be supplied as the reason for creating the new delta.

If – m is not used and the standard input is a terminal, the prompt MRs? is issued on the standard output before the standard input is read; if the standard input is not a terminal, no prompt is issued. The MRs? prompt always precedes the comments? prompt (see – y keyletter).

MRs in a list are separated by blanks and/or tab characters. An unescaped newline character terminates the MR list.

Note that if the v flag has a value (see *admin*(CP)), it is taken to be the name of a program (or shell procedure) which will validate the correctness of the MR numbers. If a nonzero exit status is returned from MR number validation program, *delta* terminates (it is assumed that the MR numbers were not all valid).

– y[*comment*] Arbitrary text used to describe the reason for making the delta. A null string is considered a valid *comment*.

If – y is not specified and the standard input is a terminal, the prompt comments? is issued on the standard output before the standard input is read; if the standard input is not a terminal, no prompt is issued. An unescaped newline character terminates the comment text.

– p Causes *delta* to print (on the standard output) the SCCS file differences before and after the delta is applied. Differences are displayed in a *diff*(C) format.

Files

All files of the form *?*-file are explained in Chapter 5, "SCCS: A Source Code Control System" in the *XENIX Programmer's Guide*. The naming convention for these files is also described there.

g-file Existed before the execution of *delta*; removed after completion of *delta*.

Name

get – Gets a version of an SCCS file.

Syntax

get [– rSID] [– ccutoff] [– ilist] [– xlist] [– aseq-no.] [– k] [– e] [– l[p]] [– p] [– m] [– n] [– s] [– b] [– g] [– t] file ...

Description

Get generates an ASCII text file from each named SCCS file according to the specifications given by its options, which begin with – . The arguments may be specified in any order, but all options apply to all named SCCS files. If a directory is named, *get* behaves as though each file in the directory were specified as a named file, except that nonSCCS files (last component of the pathname does not begin with s.) and unreadable files are silently ignored. If a name of – is given, the standard input is read; each line of the standard input is taken to be the name of an SCCS file to be processed. Again, nonSCCS files and unreadable files are silently ignored.

The generated text is normally written into a file called the *g-file* whose name is derived from the SCCS filename by simply removing the leading s.; (see also *FILES*, below).

Each of the options is explained below as though only one SCCS file is to be processed, but the effects of any option apply independently to each named file.

– r*SID* The *S*CCS *I*Dentification string (SID) of the version (delta) of an SCCS file to be retrieved.

– c*cutoff* *Cutoff* date-time, in the form:

YY[MM[DD[HH[MM[SS]]]]]

No changes (deltas) to the SCCS file that were created after the specified *cutoff* date-time are included in the generated ASCII text file. Units omitted from the date-time default to their maximum possible values; that is, – c7502 is equivalent to – c750228235959. Any number of nonnumeric characters may separate the various 2 digit pieces of the *cutoff* date-time. This feature allows you to specify a *cutoff* date in the form: "– c77/2/2 9:22:25".

– e Indicates that the *get* is for the purpose of editing or making a change (delta) to the SCCS file via a subsequent use of *delta*(CP). The – e option used in a *get* for a particular version (SID) of the SCCS file prevents further

– **s** Suppresses all output normally written on the standard output. However, fatal error messages (which always go to file descriptor 2) remain unaffected.

– **m** Causes each text line retrieved from the SCCS file to be preceded by the SID of the delta that inserted the text line in the SCCS file. The format is: SID, followed by a horizontal tab, followed by the text line.

– **n** Causes each generated text line to be preceded with the %M% identification keyword value (see below). The format is: %M% value, followed by a horizontal tab, followed by the text line. When both the – **m** and – **n** options are used, the format is: %M% value, followed by a horizontal tab, followed by the – **m** option generated format.

– **g** Suppresses the actual retrieval of text from the SCCS file. It is primarily used to generate an *l-file*, or to verify the existence of a particular SID.

– **t** Used to access the most recently created (top) delta in a given release (e.g., – **r1**), or release and level (e.g., – **r1.2**).

– **a***seq-no*. The delta sequence number of the SCCS file delta (version) to be retrieved (see *sccsfile*(F)). This option is used by the *comb*(CP) command; it is not particularly useful should be avoided. If both the – **r** and – **a** options are specified, the – **a** option is used. Care should be taken when using the – **a** option in conjunction with the – **e** option, as the SID of the delta to be created may not be what you expect. The – **r** option can be used with the – **a** and – **e** options to control the naming of the SID of the delta to be created.

For each file processed, *get* responds (on the standard output) with the SID being accessed and with the number of lines retrieved from the SCCS file.

If the – **e** option is used, the SID of the delta to be made appears after the SID accessed and before the number of lines generated. If there is more than one named file or if a directory or standard input is named, each filename is printed (preceded by a newline) before it is processed. If the – **i** option is used included deltas are listed following the notation "Included"; if the – **x** option is used, excluded deltas are listed following the notation "Excluded".

Identification Keywords

Identifying information is inserted into the text retrieved from the SCCS file by replacing *identification keywords* with their value

implied, the *g-file*'s mode is 644; otherwise the mode is 444. Only the real user need have write permission in the current directory.

The *l-file* contains a table showing which deltas were applied in generating the retrieved text. The *l-file* is created in the current directory if the – l option is used; its mode is 444 and it is owned by the real user. Only the real user need have write permission in the current directory.

Lines in the *l-file* have the following format:

a. A blank character if the delta was applied;
   * otherwise
b. A blank character if the delta was applied or wasn't applied and ignored;
   * if the delta wasn't applied and wasn't ignored
c. A code indicating a "special" reason why the delta was or was not applied:
   "I": Included
   "X": Excluded
   "C": Cut off (by a – c option)
d. Blank
e. SCCS identification (SID)
f. Tab character
g. Date and time (in the form YY/MM/DD HH:MM:SS) of creation
h. Blank
i. Login name of person who created *delta*

The comments and MR data follow on subsequent lines, indented one horizontal tab character. A blank line terminates each entry.

The *p-file* is used to pass information resulting from a *get* with an – e option along to *delta*. Its contents are also used to prevent a subsequent execution of *get* with an – e option for the same SID until *delta* is executed or the joint edit flag, j, (see *admin*(CP)) is set in the SCCS file. The *p-file* is created in the directory containing the SCCS file and the effective user must have write permission in that directory. Its mode is 644 and it is owned by the effective user. The format of the *p-file* is: the gotten SID, followed by a blank, followed by the SID that the new delta will have when it is made, followed by a blank, followed by the login name of the real user, followed by a blank, followed by the date-time the *get* was executed, followed by a blank and the – i option if it was present, followed by a blank and the – x option if it was present, followed by a newline. There can be an arbitrary number of lines in the *p-file* at any time; no two lines can have the same new delta SID.

The *z-file* serves as a *lock-out* mechanism against simultaneous updates. Its contents are the binary (2 bytes) process ID of the command (i.e., *get*) that created it. The *z-file* is created in the directory containing the SCCS file for the duration of *get*. The same protection restrictions as those for the *p-file* apply for the *z-file*. The *z-file* is

Name

gets – Gets a string from the standard input.

Syntax

gets [string]

Description

Gets can be used with *csh*(CP) to read a string from the standard input. If *string* is given it is used as a default value if an error occurs. The resulting string (either *string* or as read from the standard input) is written to the standard output. If no *string* is given and an error occurs, *gets* exits with exit status 1.

See Also

line(C), csh(CP)

- d Causes the data relocation records to be printed out.
- t Causes the text relocation records to be printed out.
- r Causes both text and data relocation to be printed.
- p Causes seek positions to be printed out as defined by macros in the include file, **<a.out.h>**.
- s Prints the symbol table.
- S Prints the file segment table with a header. (Only applicable to x.out segmented executable files.)

See Also

a.out(F), nm(CP)

Name

ld − link editor

Syntax

ld [option] file ...

Description

Ld combines several object programs into one, resolves external references, and searches libraries. *Ld* combines the given object *files*, producing an object module which can be either executed or become the input for a further *ld* run (in the latter case, the **−r** option must be given to preserve the relocation records). The output of *ld* is left by default in the file **x.out**. This file is made executable only if no errors occurred.

The files given as arguments are concatenated in the order specified. The default entry point of the output is the beginning of the first routine in the first file. The C compiler, *cc* , calls *ld* automatically unless given the **−c** option. The command line that *cc* passes to *ld* is

ld /lib/crt0.o files *cc−options* **−lc**

If any argument is a library, it is searched exactly once at the point it is encountered in the argument list. Only those routines defining an unresolved external reference are loaded. If a routine from a library references another routine in the library, and the library has not been processed by *ranlib*(CP), the referenced routine must appear after the referencing routine in the library. Thus the order of programs within libraries may be important. If the first member of a library is named '__.SYMDEF', then it is understood to be a dictionary for the library

as produced by *ranlib*; the dictionary is searched iteratively to satisfy as many references as possible.

The symbols '_etext', '_edata' and '_end' ('etext', 'edata' and 'end' in C) are reserved, and if referred to, are set to the first location above the program, the first location above initialized data, and the first location above all data, respectively. It is erroneous to define these symbols.

If no errors occur and there are no unresolved external references, then short form relocation information is attached and the file is made executable. This short form relocation information is sufficient to allow the file to be used for another pass of *ld* , to change the text and data base addresses. At the same time, the **−n**

symbol table; only enter external symbols. This option saves some space in the output file.

−X Save local symbols except for those whose names begin with 'L'. This option is used by *cc*(CP) to discard internally generated labels while retaining symbols local to routines.

−r Generate (long form) relocation records in the output file so that the output file can be the subject of another *ld* run. This flag also prevents final definitions from being given to common symbols and suppresses the 'undefined symbol' diagnostics.

−d Force definition of common storage even if the **−r** flag is present.

−nn Arrange that when the output file is executed, the text portion will be read−only and shared among all users executing the file. This involves moving the data areas up to the first possible page boundary following the end of the text. A warning is issued if the current machine does not support this option.

−nr Identical to **−nn** except that the text and data positions are reversed.

−n Identical to whichever of **−nn** and **−nr** is the default for the current machine.

−i When the output file is executed, the program text and data areas are given separate address spaces. The only difference between this option and **−n** is that with **−i** the data may start at a boundary unrelated to the position of the text. A warning is issued if the current machine does not support this option.

−o The *name* argument after **−o** is used as the name of the *ld* output file, instead of **x.out**.

−e The following argument is taken to be the name of the entry point of the loaded program. The base of the text segment is the default.

−D The next argument is a decimal number that sets the size of the data segment.

−N The next argument is taken to be a hexadecimal number that sets the pagesize, or rounding size, for use with the **−n** option. With **−i**, it specifies the base of the data

Name

lex – Generates programs for lexical analysis.

Syntax

lex [– ctvn] [file] ...

Description

Lex generates programs to be used in simple lexical analysis of text.

The input *files* (standard input default) contain strings and expressions to be searched for, and C text to be executed when strings are found.

A file lex.yy.c is generated which, when loaded with the library, copies the input to the output except when a string specified in the file is found; then the corresponding program text is executed. The actual string matched is left in *yytext*, an external character array. Matching is done in order of the strings in the file. The strings may contain square brackets to indicate character classes, as in [abx– z] to indicate a, b, x, y, and z; and the operators *, +, and ? mean respectively any nonnegative number of, any positive number of, and either zero or one occurrences of, the previous character or character class. The character . is the class of all ASCII characters except newline. Parentheses for grouping and vertical bar for alternation are also supported. The notation *r*{*d*,*e*} in a rule indicates between *d* and *e* instances of regular expression *r*. It has higher precedence than |, but lower than *, ?, +, and concatenation. The character ^ at the beginning of an expression permits a successful match only immediately after a newline, and the character $ at the end of an expression requires a trailing newline. The character / in an expression indicates trailing context; only the part of the expression up to the slash is returned in *yytext*, but the remainder of the expression must follow in the input stream. An operator character may be used as an ordinary symbol if it is within " symbols or preceded by \. Thus [a– zA– Z]+ matches a string of letters.

Three subroutines defined as macros are expected: input() to read a character; unput(*c*) to replace a character read; and output(*c*) to place an output character. They are defined in terms of the standard streams, but you can override them. The program generated is named yylex(), and the library contains a main() which calls it. The action REJECT on the right side of the rule causes this match to be rejected and the next suitable match executed; the function yymore() accumulates additional characters into the same *yytext*; and the function yyless(*p*) pushes back the portion of the string matched beginning at *p*, which should be between *yytext* and *yytext*+ *yyleng*. The macros *input* and *output* use files yyin and yyout to read from

%a *n*
number of transitions is *n* (3000)

The use of one or more of the above automatically implies the – v option, unless the – n option is used.

See Also

yacc(CP)
Xenix *Software Development Guide*

The following arguments alter *lint's* behavior:

– n Does not check compatibility against either the standard or the portable lint library.

– p Attempts to check portability to other dialects of C.

– llibname
Checks functions definitions in the specified lint library. For example, – lm causes the library *llibm.ln* to be checked.

The – D, – U, and – I options of *cc*(CP) are also recognized as separate arguments.

Certain conventional comments in the C source will change the behavior of *lint*:

/*NOTREACHED*/
At appropriate points stops comments about unreachable code.

/*VARARGS*n*/
Suppresses the usual checking for variable numbers of arguments in the following function declaration. The data types of the first *n* arguments are checked; a missing *n* is taken to be 0.

/*ARGSUSED*/
Turns on the – v option for the next function.

/*LINTLIBRARY*/
Shuts off complaints about unused functions in this file.

Lint produces its first output on a per source file basis. Complaints regarding included files are collected and printed after all source files have been processed. Finally, information gathered from all input files is collected and checked for consistency. At this point, if it is not clear whether a complaint stems from a given source file or from one of its included files, the source filename will be printed followed by a question mark.

Files

/usr/lib/lint[12] Program files

/usr/lib/llibc.ln, /usr/lib/llibport.ln, /usr/lib/llibm.ln, /usr/lib/llibdbm.ln, /usr/lib/llibtermlib.ln
Standard lint libraries (binary format)

Name

lorder – Finds ordering relation for an object library.

Syntax

lorder file ...

Description

Lorder creates an ordered listing of object filenames, showing which files depend on variables declared in other files. The *file* is one or more object or library archive files (see *ar*(CP)). The standard output is a list of pairs of object filenames. The first file of the pair refers to external identifiers defined in the second. The output may be processed by *tsort*(CP) to find an ordering of a library suitable for one-pass access by *ld*(CP).

Example

The following command builds a new library from existing .o files:

```
ar cr library `lorder *.o |tsort`
```

Files

*symref, *symdef Temp files

See Also

ar(CP), ld(CP), tsort(CP)

Notes

Object files whose names do not end with .o, even when contained in library archives, are overlooked. Their global symbols and references are attributed to some other file.

Macro Calls

Macro calls have the form:

name(arg1,arg2, ..., argn)

The (must immediately follow the name of the macro. If a defined macro name is not followed by a (, it is deemed to have no arguments. Leading unquoted blanks, tabs, and newlines are ignored while collecting arguments. Potential macro names consist of alphabetic letters, digits, and underscore _, where the first character is not a digit.

Left and right single quotation marks are used to quote strings. The value of a quoted string is the string stripped of the quotation marks.

When a macro name is recognized, its arguments are collected by searching for a matching right parenthesis. Macro evaluation proceeds normally during the collection of the arguments, and any commas or right parentheses which happen to turn up within the value of a nested call are as effective as those in the original input text. After argument collection, the value of the macro is pushed back onto the input stream and rescanned.

M4 makes available the following built-in macros. They may be redefined, but once this is done the original meaning is lost. Their values are null unless otherwise stated.

define	The second argument is installed as the value of the macro whose name is the first argument. Each occurrence of **$***n* in the replacement text, where *n* is a digit, is replaced by the *n*-th argument. Argument **0** is the name of the macro; missing arguments are replaced by the null string; **$#** is replaced by the number of arguments; **$*** is replaced by a list of all the arguments separated by commas; **$@** is like **$***, but each argument is quoted (with the current quotation marks).
undefine	Removes the definition of the macro named in its argument.
defn	Returns the quoted definition of its argument(s). It is useful for renaming macros, especially built-ins.
pushdef	Like *define*, but saves any previous definition.
popdef	Removes current definition of its argument(s), exposing the previous one if any.
ifdef	If the first argument is defined, the value is the second argument, otherwise the third. If there is no third argument, the value is null. The word XENIX is predefined in *M4*.

radix for the result; the default is 10. The third argument may be used to specify the minimum number of digits in the result.

len — Returns the number of characters in its argument.

index — Returns the position in its first argument where the second argument begins (zero origin), or – 1 if the second argument does not occur.

substr — Returns a substring of its first argument. The second argument is a zero origin number selecting the first character; the third argument indicates the length of the substring. A missing third argument is taken to be large enough to extend to the end of the first string.

translit — Transliterates the characters in its first argument from the set given by the second argument to the set given by the third. No abbreviations are permitted.

include — Returns the contents of the file named in the argument.

sinclude — Identical to *include*, except that it says nothing if the file is inaccessible.

syscmd — Executes the XENIX command given in the first argument. No value is returned.

sysval — Is the return code from the last call to *syscmd*.

maketemp — Fills in a string of XXXXX in its argument with the current process ID.

m4exit — Causes immediate exit from *m4*. Argument 1, if given, is the exit code; the default is 0.

m4wrap — Argument 1 will be pushed back at final EOF; example: m4wrap(`cleanup( )')

errprint — Prints its argument on the diagnostic output file.

dumpdef — Prints current names and definitions, for the named items, or for all if no arguments are given.

traceon — With no arguments, turns on tracing for all macros (including built-ins). Otherwise, turns on tracing for named macros.

traceoff — Turns off trace globally and for any macros specified. Macros specifically traced by *traceon* can be untraced only by specific calls to *traceoff*.

– q Question. The *make* command returns a zero or nonzero status code depending on whether the target file is or is not up-to-date.

.DEFAULT If a file must be made but there are no explicit commands or relevant built-in rules, the commands associated with the name **.DEFAULT** are used if it exists.

.PRECIOUS Dependents of this target will not be removed when quit or interrupt are hit.

.SILENT Same effect as the – s option.

.IGNORE Same effect as the – i option.

Make executes commands in *makefile* to update one or more target *names*. *Name* is typically a program. If no – f option is present, makefile, Makefile, s.makefile, and s.Makefile are tried in order. If *makefile* is – , the standard input is taken. More than one – f makefile argument pair may appear.

Make updates a target only if it depends on files that are newer than the target. All prerequisite files of a target are added recursively to the list of targets. Missing files are deemed to be out of date.

Makefile contains a sequence of entries that specify dependencies. The first line of an entry is a blank-separated, nonnull list of targets, then a :, then a (possibly null) list of prerequisite files or dependencies. Text following a ; and all following lines that begin with a tab are shell commands to be executed to update the target. The first line that does not begin with a tab or # begins a new dependency or macro definition. Shell commands may be continued across lines with the <backslash><newline> sequence. (#) and newline surround comments.

The following *makefile* says that pgm depends on two files a.o and b.o, and that they in turn depend on their corresponding source files (a.c and b.c) and a common file incl.h:

```
pgm: a.o b.o
    cc a.o b.o – o pgm
a.o: incl.h a.c
    cc – c a.c
b.o: incl.h b.c
    cc – c b.c
```

Command lines are executed one at a time, each by its own shell. A line is printed when it is executed unless the – s option is present, or the entry **.SILENT:** is in *makefile*, or unless the first character of the command is @. The – n option specifies printing without execution; however, if the command line has the string $(MAKE) in it, the

subst2. Strings (for the purposes of this type of substitution) are delimited by blanks, tabs, newline characters, and beginnings of lines. An example of the use of the substitute sequence is shown under *Libraries*.

Internal Macros

There are five internally maintained macros which are useful for writing rules for building targets:

$*** The macro **$* stands for the filename part of the current dependent with the suffix deleted. It is evaluated only for inference rules.

$@** The **$@ macro stands for the full target name of the current target. It is evaluated only for explicitly named dependencies.

$<** The **$< macro is only evaluated for inference rules or the **.DEFAULT** rule. It is the module which is out of date with respect to the target (i.e., the "manufactured" dependent filename). Thus, in the .c.o rule, the **$<** macro would evaluate to the .c file. An example for making optimized .o files from .c files is:

```
.c.o:
    cc - c - O $*.c
```

or:

```
.c.o:
    cc - c - O $<
```

$?** The **$? macro is evaluated when explicit rules from the makefile are evaluated. It is the list of prerequisites that are out of date with respect to the target; essentially, those modules which must be rebuilt.

$% The **$%** macro is only evaluated when the target is an archive library member of the form **lib(file.o)**. In this case, **$@** evaluates to lib and **$%** evaluates to the library member, **file.o**.

Four of the five macros can have alternative forms. When an upper case **D** or **F** is appended to any of the four macros the meaning is changed to "directory part" for **D** and "file part" for **F**. Thus, **$(@D)** refers to the directory part of the string **$@**. If there is no directory part **./** is generated. The only macro excluded from this alternative form is **$?**.

Suffixes

Certain names (for instance, those ending with .o) have default

This is because *make* has a set of internal rules for building files. The user may add rules to this list by simply putting them in the *makefile*.

Certain macros are used by the default inference rules to permit the inclusion of optional matter in any resulting commands. For example, **CFLAGS**, **LFLAGS**, and **YFLAGS** are used for compiler options to *cc*(CP), *lex*(CP), and *yacc*(CP) respectively. Again, the previous method for examining the current rules is recommended.

The inference of prerequisites can be controlled. The rule to create a file with suffix .o from a file with suffix .c is specified as an entry with .c.o: as the target and no dependents. Shell commands associated with the target define the rule for making a .o file from a .c file. Any target that has no slashes in it and starts with a dot is identified as a rule and not as a true target.

Libraries

If a target or dependency name contains parentheses, it is assumed to be an archive library, the string within parentheses referring to a member within the library. Thus **lib(file.o)** and **$(LIB)(file.o)** both refer to an archive library which contains file.o. (This assumes the **LIB** macro has been previously defined.) The expression **$(LIB)(file1.o file2.o)** is not legal. Rules pertaining to archive libraries have the form *.XX*.a where the *XX* is the suffix from which the archive member is to be made. An unfortunate byproduct of the current implementation requires the *XX* to be different from the suffix of the archive member. Thus, one cannot have **lib(file.o)** depend upon file.o explicitly. The most common use of the archive interface follows. Here, we assume the source files are all C type source:

```
lib: lib(file1.o) lib(file2.o) lib(file3.o)
    @echo lib is now up to date
.c.a:
    $(CC) - c $(CFLAGS) $<
    ar rv $@ $*.o
    rm -f $*.o
```

In fact, the .c.a rule listed above is built into *make* and is unnecessary in this example. A more interesting, but more limited example of an archive library maintenance construction follows:

```
lib: lib(file1.o) lib(file2.o) lib(file3.o)
    $(CC) - c $(CFLAGS) $(?:.o=.c)
    ar rv lib $?
    rm $?   @echo lib is now up to date
.c.a:;
```

Here the substitution mode of the macro expansions is used. The **$?** list is defined to be the set of object filenames (inside lib) whose

Name

mkstr – Creates an error message file from C source.

Syntax

mkstr [–] messagefile prefix file ...

Description

Mkstr is used to create files of error messages. Its use can make programs with large numbers of error diagnostics much smaller, and reduce system overhead in running the program as the error messages do not have to be constantly swapped in and out.

Mkstr will process each specified *file*, placing a massaged version of the input file in a file whose name consists of the specified *prefix* and the original name. The optional dash (–) causes the error messages to be placed at the end of the specified message file for recompiling part of a large *mkstr*ed program.

A typical *mkstr* command line is

```
mkstr pistrings xx *.c
```

This command causes all the error messages from the C source files in the current directory to be placed in the file *pistrings* and processed copies of the source for these files to be placed in files whose names are prefixed with *xx*.

To process the error messages in the source to the message file, *mkstr* keys on the string 'error(" ' in the input stream. Each time it occurs, the C string starting at the '"' is placed in the message file followed by a null character and a newline character; the null character terminates the message so it can be easily used when retrieved, the newline character makes it possible to sensibly *cat* the error message file to see its contents. The massaged copy of the input file then contains a *lseek* pointer into the file which can be used to retrieve the message. For example, the command changes

```
error("Error on reading", a2, a3, a4);
```

into

```
error(m, a2, a3, a4);
```

where *m* is the seek position of the string in the resulting error message file. The programmer must create a routine *error* which opens the message file, reads the string, and prints it out. The following example illustrates such a routine.

Name

nm – Prints name list.

Syntax

nm [**–gnoOprucv**] [file ...]

Description

nm prints the name list (symbol table) of each object *file* in the argument list. If an argument is an archive, a listing for each object file in the archive will be produced. If no *file* is given, the symbols in **x.out** are listed.

Each symbol name is preceded by its value in hexadecimal (blanks if undefined) and one of the letters **U** (undefined), **A** (absolute), **T** (text segment symbol), **D** (data segment symbol), **B** (bss segment symbol), or **C** (common symbol). If the symbol is local (non–external) the type letter is in lowercase. The output is sorted alphabetically.

Options are:

–g Print only global (external) symbols.

–n Sort numerically rather than alphabetically.

–o Prepend file or archive element name to each output line rather than only once.

–O Print symbol values in octal.

–p Don't sort; print in symbol–table order.

–r Sort in reverse order.

–u Print only undefined symbols.

–c Print only C program symbols (symbols which begin with '_') as they appeared in the C program.

–v Also describe the object file and symbol table format.

Files

x.out Default input file

See Also

ar(CP), ar(F), x.out(F)

Name

prs – Prints an SCCS file.

Syntax

prs [– d[dataspec]] [– r[SID]] [– e] [– l] [– a] files

Description

Prs prints, on the standard output, all or part of an SCCS file (see *sccsfile*(F)) in a user supplied format. If a directory is named, *prs* behaves as though each file in the directory were specified as a named file, except that nonSCCS files (last component of the pathname does not begin with s.), and unreadable files are silently ignored. If a name of – is given, the standard input is read; each line of the standard input is taken to be the name of an SCCS file or directory to be processed; nonSCCS files and unreadable files are silently ignored.

Arguments to *prs*, which may appear in any order, consist of options, and filenames.

All the described options apply independently to each named file:

– d[*dataspec*] Used to specify the output data specification. The *dataspec* is a string consisting of SCCS file *data keywords* (see *Data Keywords*) interspersed with optional user-supplied text.

– r[*SID*] Used to specify the *S*CCS *ID*entification (SID) string of a delta for which information is desired. If no SID is specified, the SID of the most recently created delta is assumed.

– e Requests information for all deltas created *earlier* than and including the delta designated via the – r option.

– l Requests information for all deltas created *later* than and including the delta designated via the – r option.

– a Requests printing of information for both removed, i.e., delta type = *R*, (see *rmdel*(CP)) and existing, i.e., delta type = *D*, deltas. If the – a option is not specified, information for existing deltas only is provided.

TABLE 1. SCCS Files Data Keywords

Keyword	*Data Item*	*File Section*	*Value*	*Format*
:Dt:	Delta information	Delta Table	See below*	S
:DL:	Delta line statistics	"	:Li:/:Ld:/:Lu:	S
:Li:	Lines inserted by Delta	"	nnnnn	S
:Ld:	Lines deleted by Delta	"	nnnnn	S
:Lu:	Lines unchanged by Delta	"	nnnnn	S
:DT:	Delta type	"	*D* or *R*	S
:I:	SCCS ID string (SID)	"	:R:.:L:.:B:.:S:	S
:R:	Release number	"	nnnn	S
:L:	Level number	"	nnnn	S
:B:	Branch number	"	nnnn	S
:S:	Sequence number	"	nnnn	S
:D:	Date Delta created	"	:Dy:/:Dm:/:Dd:	S
:Dy:	Year Delta created	"	nn	S
:Dm:	Month Delta created	"	nn	S
:Dd:	Day Delta created	"	nn	S
:T:	Time Delta created	"	:Th::Tm::Ts:	S
:Th:	Hour Delta created	"	nn	S
:Tm:	Minutes Delta created	"	nn	S
:Ts:	Seconds Delta created	"	nn	S
:P:	Programmer who created Delta	"	logname	S
:DS:	Delta sequence number	"	nnnn	S
:DP:	Predecessor Delta seq-no.	"	nnnn	S
:DI:	Seq-no. of deltas incl., excl., ignored	"	:Dn:/:Dx:/:Dg:	S
:Dn:	Deltas included (seq #)	"	:DS: :DS:...	S
:Dx:	Deltas excluded (seq #)	"	:DS: :DS:...	S
:Dg:	Deltas ignored (seq #)	"	:DS: :DS:...	S
:MR:	MR numbers for delta	"	text	M
:C:	Comments for delta	"	text	M
:UN:	User names	User Names	text	M
:FL:	Flag list	Flags	text	M
:Y:	Module type flag	"	text	S
:MF:	MR validation flag	"	*yes* or *no*	S
:MP:	MR validation pgm name	"	text	S
:KF:	Keyword error/warning flag	"	*yes* or *no*	S
:BF:	Branch flag	"	*yes* or *no*	S
:J:	Joint edit flag	"	*yes* or *no*	S
:LK:	Locked releases	"	:R:...	S
:Q:	User defined keyword	"	text	S
:M:	Module name	"	text	S
:FB:	Floor boundary	"	:R:	S
:CB:	Ceiling boundary	"	:R:	S
:Ds:	Default SID	"	:I:	S
:ND:	Null delta flag	"	*yes* or *no*	S
:FD:	File descriptive text	Comments	text	M
:BD:	Body	Body	text	M
:GB:	Gotten body	"	text	M
:W:	A form of *what*(C) string	N/A	:Z::M:\t:I:	S
:A:	A form of *what*(C) string	N/A	:Z::Y: :M: :I::Z:	S
:Z:	*what*(C) string delimiter	N/A	@(#)	S
:F:	SCCS filename	N/A	text	S
:PN:	SCCS file pathname	N/A	text	S

* :Dt: = :DT: :I: :D: :T: :P: :DS: :DP:

Name

ranlib – Converts archives to random libraries.

Syntax

ranlib archive ...

Description

Ranlib converts each *archive* to a form that can be loaded more rapidly by the loader, by adding a table of contents named **__.SYMDEF** to the beginning of the archive. It uses *ar*(CP) to reconstruct the archive, so sufficient temporary file space must be available in the file system containing the current directory.

See Also

ld(CP), ar(CP), copy(C), settime(C)

Notes

Because generation of a library by *ar* and randomization by *ranlib* are separate, phase errors are possible. The loader *ld* warns when the modification date of a library is more recent than the creation of its dictionary; but this means you get the warning even if you only copy the library. On XENIX 68K use of *ranlib* is optional.

Include:
include filename

The option – h causes quoted strings to be turned into 27H constructs. – C copies comments to the output, and attempts to format it neatly. Normally, continuation lines are marked with an & in column 1; the option – 6x makes the continuation character x and places it in column 6.

Name

rmdel – Removes a delta from an SCCS file.

Syntax

rmdel – r*SID* files

Description

Rmdel removes the delta specified by the *SID* from each named SCCS file. The delta to be removed must be the newest (most recent) delta in its branch in the delta chain of each named SCCS file. In addition, the SID specified must *not* be that of a version being edited for the purpose of making a delta. That is, if a *p-file* exists for the named SCCS file, the SID specified must *not* appear in any entry of the *p-file*(see *get*(CP)).

If a directory is named, *rmdel* behaves as though each file in the directory were specified as a named file, except that nonSCCS files (last component of the pathname does not begin with s.) and unreadable files are silently ignored. If a name of – is given, the standard input is read; each line of the standard input is taken to be the name of an SCCS file to be processed; nonSCCS files and unreadable files are silently ignored.

Files

x-file See *delta*(CP)

z-file See *delta*(CP)

See Also

delta(CP), get(CP), help(CP), prs(CP), sccsfile(F)

Diagnostics

Use *help*(CP) for explanations.

Name

sccsdiff – Compares two versions of an SCCS file.

Syntax

sccsdiff – rSID1 – rSID2 [– p] [– sn] files

Description

Sccsdiff compares two versions of an SCCS file and generates the differences between the two versions. Any number of SCCS files may be specified, but arguments apply to all files.

– r*SID?* *SID1* and *SID2* specify the deltas of an SCCS file that are to be compared. Versions are passed to *bdiff*(C) in the order given.

– p Pipe output for each file through *pr*(C).

– s*n* *n* is the file segment size that *bdiff* will pass to *diff*(C). This is useful when *diff* fails due to a high system load.

Files

/tmp/get????? Temporary files

See Also

bdiff(C), get(CP), help(CP), pr(C)

Diagnostics

file: *No differences* If the two versions are the same.

Use *help*(CP) for explanations.

Name

spline – Interpolates smooth curve.

Syntax

spline [option] ...

Description

Spline takes pairs of numbers from the standard input as abcissas and ordinates of a function. It produces a similar set, which is approximately equally spaced and includes the input set, on the standard output. The cubic spline output has two continuous derivatives, and enough points to look smooth when plotted.

The following options are recognized, each as a separate argument.

– a Supplies abscissas automatically (they are missing from the input); spacing is given by the next argument, or is assumed to be 1 if next argument is not a number.

– k The constant k used in the boundary value computation

$$y_0'' = ky_1', \ldots, y_n'' = ky_{n-1}'$$

is set by the next argument. By default $k = 0$.

– n Spaces output points so that approximately n intervals occur between the lower and upper x limits. (Default $n = 100$.)

– p Makes output periodic, i.e. matches derivatives at ends. First and last input values should normally agree.

– x Next 1 (or 2) arguments are lower (and upper) x limits. Normally these limits are calculated from the data. Automatic abcissas start at lower limit (default 0).

Diagnostics

When data is not strictly monotone in x, *spline* reproduces the input without interpolating extra points.

Notes

A limit of 1000 input points is silently enforced.

Name

strip – Removes symbols and relocation bits.

Syntax

strip name ...

Description

Strip removes the symbol table and relocation bits ordinarily attached to the output of the assembler and link editor. This is useful for saving space after a program has been debugged.

The effect of *strip* is the same as use of the – s option of *ld*.

If *name* is an archive file, *strip* will remove the local symbols from any *a.out* format files it finds in the archive. Certain libraries, such as those residing in /lib, have no need for local symbols. By deleting them, the size of the archive is decreased and link editing performance is increased.

Files

/tmp/stm*	Temporary file

See Also

ld(CP)

Name

tsort – Sorts a file topologically.

Syntax

tsort [file]

Description

Tsort produces on the standard output a totally ordered list of items consistent with a partial ordering of items mentioned in the input *file*. If no *file* is specified, the standard input is understood.

The input consists of pairs of items (nonempty strings) separated by blanks. Pairs of different items indicate ordering. Pairs of identical items indicate presence, but not ordering.

See Also

lorder(CP)

Diagnostics

Odd data: There is an odd number of fields in the input file.

Notes

The *sort* algorithm is quadratic, which can be slow if you have a large input list.

Name

val – Validates an SCCS file.

Syntax

val –

val [– s] [– rSID] [– mname] [– ytype] files

Description

Val determines if the specified *file* is an SCCS file meeting the characteristics specified by the optional argument list. Arguments to *val* may appear in any order. The arguments consist of options, which begin with a – , and named files.

Val has a special argument, – , which causes reading of the standard input until an end-of-file condition is detected. Each line read is independently processed as if it were a command line argument list.

Val generates diagnostic messages on the standard output for each command line and file processed and also returns a single 8-bit code upon exit as described below.

The options are defined as follows. The effects of any option apply independently to each named file on the command line:

– **s** The presence of this argument silences the diagnostic message normally generated on the standard output for any error that is detected while processing each named file on a given command line.

– r*SID* The argument value *SID* (*S*CCS *ID*entification String) is an SCCS delta number. A check is made to determine if the *SID* is ambiguous (e. g., **r1** is ambiguous because it physically does not exist but implies **1.1**, **1.2**, etc. which may exist) or invalid (e. g., **r1.0** or **r1.1.0** are invalid because neither case can exist as a valid delta number). If the *SID* is valid and not ambiguous, a check is made to determine if it actually exists.

– m*name* The argument value *name* is compared with the SCCS %M% keyword in *file*.

– *ytype* The argument value *type* is compared with the SCCS %Y% keyword in *file*.

Name

xref – Cross-references C programs.

Syntax

xref [file ...]

Description

Xref reads the named *files* or the standard input if no file is specified and prints a cross reference consisting of lines of the form

identifier filename line numbers ...

Function definition is indicated by a plus sign (+) preceding the line number.

See Also

cref(CP)

```
cc – E name.c |xstr – c –
cc – c x.c
mv x.o name.o
```

Xstr does not touch the file *strings* unless new items are added, thus *make* can avoid remaking *xs.o* unless truly necessary.

Files

strings	Data base of strings
x.c	Massaged C source
xs.c	C source for definition of array "xstr"
/tmp/xs*	Temp file when "xstr name" doesn't touch *strings*

See Also

mkstr(CP)

Credit

This utility was developed at the University of California at Berkeley and is used with permission.

Notes

If a string is a suffix of another string in the data base, but the shorter string is seen first by *xstr* , both strings will be placed in the data base when just placing the longer one there will do.

Diagnostics

The number of reduce-reduce and shift-reduce conflicts is reported on the standard output; a more detailed report is found in the **y.output** file. Similarly, if some rules are not reachable from the start symbol, this is also reported.

Notes

Because filenames are fixed, at most one *yacc* process can be active in a given directory at a time.

CONTENTS

System Services (S)

Index

Name

intro - Introduces system services, library routines and error numbers.

Syntax

#include <errno.h>

Description

This section describes all system services. System services include all routines or system calls that are available in the operating system kernel. These routines are available to a C program automatically as part of the standard library libc. Other routines are available in a variety of libraries. On 8086/88 and 286 systems, versions for Small, Middle, and Large model programs are provided (that is, three of each library).

To use routines in a program that are not part of the standard library libc, the appropriate library must be linked. This is done by specifying - l*name* to the compiler or linker, where *name* is the name listed below. For example - lm, and - ltermcap are specifications to the linker to search the named libraries for routines to be linked to the object module. The names of the available libraries are:

c The standard library containing all system call interfaces, Standard I/O routines, and other general purpose services.

m The standard math library.

termcap Routines for accessing the *termcap* data base describing terminal characteristics.

curses Screen and cursor manipulation routines.

dbm Data base management routines.

Most services that are part of the operating system kernel have one or more error returns. An error condition is indicated by an otherwise impossible returned value. This is almost always - 1; the individual descriptions specify the details. An error number is also made available in the external variable *errno*. *Errno* is not cleared on successful calls, so it should be tested only after an error has been indicated.

All of the possible error numbers are not listed in each system call description because many errors are possible for most of the calls. The following is a complete list of the error numbers and their names as defined in <error.h>.

12 ENOMEM Not enough space
During an *exec*, or *sbrk*, a program asks for more space than the system is able to supply. This is not a temporary condition; the maximum space size is a system parameter. The error may also occur if the arrangement of text, data, and stack segments requires too many segmentation registers, or if there is not enough swap space during a *fork*.

13 EACCES Permission denied
An attempt was made to access a file in a way forbidden by the protection system.

14 EFAULT Bad address
The system encountered a hardware fault in attempting to use an argument of a system call.

15 ENOTBLK Block device required
A nonblock file was mentioned where a block device was required, e.g., in *mount*.

16 EBUSY Device busy
An attempt to mount a device that was already mounted or an attempt was made to dismount a device on which there is an active file (open file, current directory, mounted-on file, active text segment). It will also occur if an attempt is made to enable accounting when it is already enabled.

17 EEXIST File exists
An existing file was mentioned in an inappropriate context, e.g., *link*.

18 EXDEV Cross-device link
A link to a file on another device was attempted.

19 ENODEV No such device
An attempt was made to apply an inappropriate system call to a device; e.g., read a write-only device.

20 ENOTDIR Not a directory
A nondirectory was specified where a directory is required, for example in a path prefix or as an argument to *chdir*(S).

21 EISDIR Is a directory
An attempt to write on a directory.

22 EINVAL Invalid argument
Some invalid argument (e.g., dismounting a nonmounted device; mentioning an undefined signal in *signal*, or *kill*; reading or writing a file for which *lseek* has generated a negative pointer). Also set by the math functions described in the (S) entries of this manual.

between processes vying for control of that region.

37 ENOTNAM Not a name file
A *creatsem*(S), *opensem*(S), *waitsem*(S), or *sigsem*(S) was issued using an invalid semaphore identifier.

38 ENAVAIL Not available
An *opensem*(S), *waitsem*(S) or *sigsem*(S) was issued to a semaphore that has not been initialized by a call to *creatsem*(S). A *sigsem* was issued to a semaphore out of sequence; i.e., before the process has issued the corresponding *waitsem* to the semaphore. An *nbwaitsem* was issued to a semaphore guarding a resource that is currently in use by another process. The semaphore on which a process was waiting has been left in an inconsistent state when the process controlling the semaphore exits without relinquishing control properly; i.e., without issuing a *waitsem* on the semaphore.

39 EISNAM A name file
A name file (semaphore, shared data, etc.) was specified when not expected.

Definitions

Process ID

Each active process in the system is uniquely identified by a positive integer called a process ID. The range of this ID is from 0 to 30,000.

Parent Process ID

A new process is created by a currently active process; see *fork*(S). The parent process ID of a process is the process ID of its creator.

Process Group ID

Each active process is a member of a process group that is identified by a positive integer called the process group ID. This ID is the process ID of the group leader. This grouping permits the signaling of related processes; see *kill*(S).

Tty Group ID

Each active process can be a member of a terminal group that is identified by a positive integer called the tty group ID. This grouping is used to terminate a group of related process upon termination of one of the processes in the group; see *exit*(S) and *signal*(S).

Pathname and Path Prefix

A pathname is a null-terminated character string starting with an optional slash (/), followed by zero or more directory names separated by slashes, optionally followed by a filename. A filename is a string of 1 to 14 characters other than the ASCII slash and null, and a directory name is a string of 1 to 14 characters (other than the ASCII slash and null) naming a directory.

If a pathname begins with a slash, the path search begins at the *root* directory. Otherwise, the search begins from the current working directory.

A slash by itself names the root directory.

Unless specifically stated otherwise, the null pathname is treated as if it named a nonexistent file.

Directory

Directory entries are called links. By convention, a directory contains at least two links, . and .., referred to as *dot* and *dot-dot* respectively. Dot refers to the directory itself and dot-dot refers to its parent directory.

Root Directory and Current Working Directory

Each process has associated with it a concept of a root directory and a current working directory for the purpose of resolving pathname searches. A process' root directory need not be the root directory of the root file system. See *chroot*(C) and *chroot*(S).

File Access Permissions

Read, write, and execute/search permissions on a file are granted to a process if one or more of the following are true:

The process' effective user ID is super-user.

The process' effective user ID matches the user ID of the owner of the file and the appropriate access bit of the "owner" portion (0700) of the file mode is set.

The process' effective user ID does not match the user ID of the owner of the file, and the process' group ID matches the group of the file and the appropriate access bit of the "group" portion (070) of the file mode is set.

The process' effective user ID does not match the user ID of the owner of the file, and the process' effective group ID does not

Name

a64l, l64a – Converts between long integer and base 64 ASCII.

Syntax

```
long a64l (s)
char *s;

char *l64a (l)
long l;
```

Description

These routines are used to maintain numbers stored in base 64 ASCII. This is a notation by which long integers can be represented by up to six characters; each character represents a "digit" in a radix 64 notation.

The characters used to represent "digits" are . for 0, / for 1, 0 through 9 for 2 through 11, A through Z for 12 through 37, and a through z for 38 through 63.

A64l takes a pointer to a null-terminated base 64 representation and returns a corresponding long value. *L64a* takes a long argument and returns a pointer to the corresponding base 64 representation.

Notes

The value returned by *l64a* is a pointer into a static buffer, the contents of which are overwritten by each call.

Name

abs – Returns an integer absolute value.

Syntax

```
int abs (i)
int i;
```

Description

Abs returns the absolute value of its integer operand.

See Also

fabs in floor(S)

Notes

If the largest negative integer supported by the hardware is given, the function returns it unchanged.

Return Value

If the requested access is permitted, a value of 0 is returned. Otherwise, a value of – 1 is returned and *errno* is set to indicate the error.

See Also

chmod(S), stat(S)

Notes

The super-user (root) may access any file, regardless of permission settings.

Path points to an illegal address. [EFAULT]

Return Value

Upon successful completion, a value of 0 is returned. Otherwise, a value of – 1 is returned and *errno* is set to indicate the error.

See Also

accton(C), acctcom(C), acct(F)

Name

assert – Helps verify validity of program.

Syntax

#include <assert.h>

assert (expression);

Description

This macro is useful for putting diagnostics into programs under development. When it is executed, if ***expression*** is false, it prints

Assertion failed: file ***name***, line ***nnn***

on the standard error file and exits. *Name* is the source filename and ***nnn*** the source line number of the ***assert*** statement.

Notes

To suppress calls to ***assert***, use the option "– DNDEBUG" when compiling the program; see *cc*(CP)).

Name

bessel, j0, j1, jn, y0, y1, yn – Performs Bessel functions.

Syntax

```
#include <math.h>

double j0 (x)
double x;

double j1 (x)
double x;

double jn (n, x);
double x;

double y0 (x)
double x;

double y1 (x)
double x;

double yn (n, x)
int n;
double x;
```

Description

These functions calculate Bessel functions of the first and second kinds for real arguments and integer orders.

Notes

Negative arguments cause *y0*, *y1*, and *yn* to return a huge negative value.

Name

chdir – Changes the working directory.

Syntax

```
int chdir (path)
char *path;
```

Description

Path points to the pathname of a directory. *Chdir* causes the named directory to become the current working directory, the starting point for path searches for pathnames not beginning with /.

Chdir will fail and the current working directory will be unchanged if one or more of the following are true:

A component of the pathname is not a directory. [ENOTDIR]

The named directory does not exist. [ENOENT]

Search permission is denied for any component of the pathname. [EACCES]

Path points outside the process' allocated address space. [EFAULT]

Return Value

Upon successful completion, a value of 0 is returned. Otherwise, a value of – 1 is returned and *errno* is set to indicate the error.

See Also

chroot(S)

Chmod will fail and the file mode will be unchanged if one or more of the following are true:

A component of the path prefix is not a directory. [ENOTDIR]

The named file does not exist. [ENOENT]

Search permission is denied on a component of the path prefix. [EACCES]

The effective user ID does not match the owner of the file and the effective user ID is not super-user. [EPERM]

The named file resides on a read-only file system. [EROFS]

Path points outside the process' allocated address space. [EFAULT]

Return Value

Upon successful completion, a value of 0 is returned. Otherwise, a value of – 1 is returned and *errno* is set to indicate the error.

See Also

chown(S), mknod(S)

Name

chroot – Changes the root directory.

Syntax

```
int chroot (path)
char *path;
```

Description

Path points to a pathname naming a directory. *Chroot* causes the named directory to become the root directory, the starting point for path searches for pathnames beginning with /.

To change the root directory, the effective user ID of the process must be super-user.

The ".." entry in the root directory is interpreted to mean the root directory itself. Thus, ".." cannot be used to access files outside the root directory.

Chroot will fail and the root directory will remain unchanged if one or more of the following are true:

Any component of the pathname is not a directory. [ENOTDIR]

The named directory does not exist. [ENOENT]

The effective user ID is not super-user. [EPERM]

Path points outside the process' allocated address space. [EFAULT]

Return Value

Upon successful completion, a value of 0 is returned. Otherwise, a value of – 1 is returned and *errno* is set to indicate the error.

See Also

chdir(S), chroot(C)

Name

close – Closes a file descriptor.

Syntax

int close (fildes)
int fildes;

Description

Fildes is a file descriptor obtained from a ***creat***, ***open***, ***dup***, ***fcntl***, or ***pipe*** system call. ***Close*** closes the file descriptor indicated by ***fildes***.

Close will fail if ***fildes*** is not a valid open file descriptor. [EBADF]

Return Value

Upon successful completion, a value of 0 is returned. Otherwise, a value of – 1 is returned and ***errno*** is set to indicate the error.

See Also

creat(S), dup(S), exec(S), fcntl(S), open(S), pipe(S)

Notes

Because *_toupper* and *_tolower* are implemented as macros, they should not be used where unwanted side effects may occur. Removing the *_toupper* and *_tolower* macros with the #undef directive causes the corresponding library functions to be linked instead. This allows any arguments to be used without worry about side effects.

The named file resides or would reside on a read-only file system. [EROFS]

The file is a pure procedure (shared text) file that is being executed. [ETXTBSY]

The file exists and write permission is denied. [EACCES]

The named file is an existing directory. [EISDIR]

Twenty file descriptors are currently open. [EMFILE]

Path points outside the process' allocated address space. [EFAULT]

Return Value

Upon successful completion, a nonnegative integer, namely the file descriptor, is returned. Otherwise, a value of – 1 is returned and *errno* is set to indicate the error.

See Also

close(S), dup(S), lseek(S), open(S), read(S), umask(S), write(S)

Notes

Open(S) is preferred to *creat*.

Diagnostics

Createm returns the value – 1 if an error occurs. If the semaphore named by sem_name is already open for use by other processes, *errno* is set to EEXIST. If the file specified exists but is not a semaphore type, *errno* is set to ENOTNAM. If the semaphore has not been initialized by a call to *createm*, *errno* is set to ENAVAIL.

Notes

After a *createm* you must do a *waitsem* to gain control of a given resource.

Notes

The return value from *crypt* points to static data that is overwritten by each call.

Name

ctime, localtime, gmtime, asctime, tzset – Converts date and time to ASCII.

Syntax

```
char *ctime (clock)
long *clock;

#include <time.h>

struct tm *localtime (clock)
long *clock;

struct tm *gmtime (clock)
long *clock;

char *asctime (tm)
struct tm *tm;

tzset ( )

extern long timezone;
extern int daylight;
extern char tzname;
```

Description

Ctime converts a time pointed to by *clock* (such as returned by *time*(S)) into ASCII and returns a pointer to a 26-character string in the following form:

Sun Sep 16 01:03:52 1973\n\0

If necessary, fields in this string are padded with spaces to keep the string a constant length.

Localtime and *gmtime* return pointers to structures containing the time as a variety of individual quantities. These quantities give the time on a 24-hour clock, day of month (1-31), month of year (0-11), day of week (Sunday = 0), year (since 1900), day of year (0-365), and a flag that is nonzero if daylight saving time is in effect. *Localtime* corrects for the time zone and possible daylight savings time. *Gmtime* converts directly to Greenwich time (GMT), which is the time the XENIX system uses.

Asctime converts the times returned by *localtime* and *gmtime* to a 26-character ASCII string and returns a pointer to this string.

Name

ctype, isalpha, isupper, islower, isdigit, isxdigit, isalnum, isspace, ispunct, isprint, isgraph, iscntrl, isascii – Classifies characters.

Syntax

```
#include <ctype.h>

int isalpha (c)
int c;

. . .
```

Description

These macros classify ASCII-coded integer values by table lookup. Each returns nonzero for true, zero for false. *Isascii* is defined on all integer values; the rest are defined only where *isascii* is true and on the single non-ASCII value EOF (see *stdio*(S)).

isalpha	*c* is a letter
isupper	*c* is an uppercase letter
islower	*c* is a lowercase letter
isdigit	*c* is a digit [0-9]
isxdigit	*c* is a hexidecimal digit [0-9], [A-F] or [a-f]
isalnum	*c* is an alphanumeric
isspace	*c* is a space, tab, carriage return, newline, vertical tab, or form feed
ispunct	*c* is a punctuation character (neither control nor alphanumeric)
isprint	*c* is a printing character, octal 40 (space) through octal 176 (tilde)
isgraph	*c* is a printing character, like *isprint* except false for space
iscntrl	*c* is a delete character (octal 177) or ordinary control character (less than octal 40).
isascii	*c* is an ASCII character, code less than 0200

Name

curses – Performs screen and cursor functions.

Syntax

cc [flags] files – lcurses – ltermlib [libraries]

Description

These routines give the user a method of updating screens with reasonable optimization. They keep an image of the current screen, and the user sets up an image of a new one. Then the *refresh()* tells the routines to make the current screen look like the new one. In order to initialize the routines, the routine *initscr()* must be called before any of the other routines that deal with windows and screens are used.

The routines are linked with the loader option -lcurses.

See Also

termcap(F), stty(S), setenv(S)

Functions

addch(ch)	Adds a character to *stdscr*
addstr(str)	Adds a string to *stdscr*
box(win,vert,hor)	Draws a box around a window
crmode()	Sets cbreak mode
clear()	Clears *stdscr*
clearok(scr,boolf)	Sets clear flag for *scr*
clrtobot()	Clears to bottom on *stdscr*
clrtoeol()	Clears to end of line on *stdscr*
delwin(win)	Delete *win*
echo()	Sets echo mode
erase()	Erase *stdscr*
getch()	Gets a char through *stdscr*
getstr(str)	Gets a string through *stdscr*
gettmode()	Gets tty modes
getyx(win,y,x)	Gets (y,x) coordinates
inch()	Gets char at current (y,x) co-ordinates
initscr()	Initializes screens
leaveok(win,boolf)	Sets leave flag for *win*
longname(termbuf,name)	Gets long name from *termbuf*
move(y,x)	Moves to (y,x) on *stdscr*
mvcur(lasty,lastx,newy,newx)	Actually moves cursor
newwin(lines,cols,begin_y,begin_x)	Creates a new window

Name

cuserid – Gets the login name of the user.

Syntax

```
#include <stdio.h>

char *cuserid (s)
char *s;
```

Description

Cuserid returns a pointer to string which represents the login name of the owner of the current process. If (int) *s* is zero, this representation is generated in an internal static area, the address of which is returned. If (int) *s* is nonzero, *s* is assumed to point to an array of at least L_cuserid characters; the representation is left in this array. The manifest constant L_cuserid is defined in **<stdio.h>**.

Diagnostics

If the login name cannot be found, *cuserid* returns NULL; if *s* is nonzero in this case, \0 will be placed at **s*.

See Also

getlogin(S), *getpwent* in getpwent(S)

Notes

Cuserid uses *getpwnam* (see *getpwent*(S)); thus the results of a user's call to the latter will be obliterated by a subsequent call to the former.

traverse the database:

```
for(key=firstkey(); key.dptr!=NULL; key=nextkey(key))
```

Diagnostics

All functions that return an *int* indicate errors with negative values. A zero return indicates ok. Routines that return a *datum* indicate errors with a null (0) *dptr*.

Notes

The ".pag" file will contain holes so that its apparent size is about four times its actual content. Older XENIX systems may create real file blocks for these holes when touched. These files cannot be copied by normal means (cp, cat, tp, tar, ar) without filling in the holes.

Dptr pointers returned by these subroutines point into static storage that is changed by subsequent calls.

The sum of the sizes of a key/content pair must not exceed the internal block size (currently 512 bytes). Moreover all key/content pairs that hash together must fit on a single block. *Store* will return an error in the event that a disk block fills with inseparable data.

Delete does not physically reclaim file space, although it does make it available for reuse.

The order of keys presented by *firstkey* and *nextkey* depends on a hashing function.

These routines are not reentrant, so they should not be used on more than one database at a time.

Credit

This utility was developed at the University of California at Berkeley and is used with permission.

Name

dup, dup2 – Duplicates an open file descriptor.

Syntax

```
int dup (fildes)
int fildes;

dup2(fildes, fildes2)
int fildes, fildes2;
```

Description

Fildes is a file descriptor obtained from a ***creat***, ***open***, ***dup***, ***fcntl***, or ***pipe*** system call. ***Dup*** returns a new file descriptor having the following in common with the original:

Same open file (or pipe).

Same file pointer (i.e., both file descriptors share one file pointer).

Same access mode (read, write or read/write).

The new file descriptor is set to remain open across ***exec*** system calls. See *fcntl*(S).

Dup returns the lowest available file descriptor. ***Dup2*** causes ***fildes2*** to refer to the same file as ***fildes***. If ***fildes2*** already referred to an open file, it is closed first.

Dup will fail if one or more of the following are true:

Fildes is not a valid open file descriptor. [EBADF]

Twenty file descriptors are currently open. [EMFILE]

Return Value

Upon successful completion a nonnegative integer, namely the file descriptor, is returned. Otherwise, a value of – 1 is returned and ***errno*** is set to indicate the error.

See Also

creat(S), close(S), exec(S), fcntl(S), open(S), pipe(S)

Name

execl, execv, execle, execve, execlp, execvp – Executes a file.

Syntax

```
int execl (path, arg0, arg1, ..., argn, 0)
char *path, *arg0, *arg1, ..., *argn;

int execv (path, argv)
char *path, *argv[ ];

int execle (path, arg0, arg1, ..., argn, 0, envp)
char *path, *arg0, *arg1, ..., *argn, *envp[ ];

int execve (path, argv, envp);
char *path, *argv[ ], *envp[ ];

int execlp (file, arg0, arg1, ..., argn, 0)
char *file, *arg0, *arg1, ..., *argn;

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

Description

Exec in all its forms transforms the calling process into a new process. The new process is constructed from an ordinary, executable file called the "new process file" . There can be no return from a successful *exec* because the calling process is overlaid by the new process.

Path points to a pathname that identifies the new process file.

File points to the new process file. The path prefix for this file is obtained by a search of the directories passed as the *environment* line "PATH =" (see *environ*(M)). The environment is supplied by the shell (see *sh*(C)).

Arg0, *arg1*, ..., *argn* are pointers to null-terminated character strings. These strings constitute the argument list available to the new process. By convention, at least *arg0* must be present, and it must point to a string that is the same as *path* (or its last component).

Argv is an array of character pointers to null-terminated strings. These strings constitute the argument list available to the new process. By convention, *argv* must have at least one member, and it must point to a string that is the same as *path* (or its last component). *Argv* is terminated by a null pointer.

From C, two interfaces are available. *Execl* is useful when a known file with known arguments is being called; the arguments to *execl* are the character strings constituting the file and the arguments. The first argument is conventionally the same as the filename (or its last component). A 0 argument must end the argument list.

The *execv* version is useful when the number of arguments is unknown in advance. The arguments to *execv* are the name of the file to be executed and a vector of strings containing the arguments. The last argument string must be followed by a 0 pointer.

When a C program is executed, it is called as follows:

```
main(argc, argv, envp)
int argc;
char **argv, **envp;
```

where *argc* is the argument count and *argv* is an array of character pointers to the arguments themselves. As indicated, *argc* is conventionally at least one and the first member of the array points to a string containing the name of the file.

Argv is directly usable in another *execv* because *argv*[*argc*] is 0.

Envp is a pointer to an array of strings that constitute the *environment* of the process. Each string consists of a name, an "=", and a null-terminated value. The array of pointers is terminated by a null pointer. The shell *sh*(C) passes an environment entry for each global shell variable defined when the program is called. See *environ*(M) for some conventionally used names. The C run-time start-off routine places a copy of *envp* in the global cell *environ*, which is used by *execv* and *execl* to pass the environment to any subprograms executed by the current program. The *exec* routines use lower-level routines as follows to pass an environment explicitly:

```
execle(file, arg0, arg1, . . . , argn, 0, environ);
execve(file, argv, environ);
```

Execlp and *execvp* are called with the same arguments as *execl* and *execv*, but duplicate the shell's actions in searching for an executable file in a list of directories. The directory list is obtained from the environment.

Exec will fail and return to the calling process if one or more of the following are true:

One or more components of the new process file's pathname do not exist. [ENOENT]

A component of the new process file's path prefix is not a directory. [ENOTDIR]

Name

exit – Terminates a process.

Syntax

exit (status)
int status;

Description

Exit terminates the calling process. All of the file descriptors open in the calling process are closed.

If the parent process of the calling process is executing a *wait*, it is notified of the calling process' termination and the low-order 8 bits (i.e., bits 0377) of *status* are made available to it; see *wait*(S).

If the parent process of the calling process is not executing a *wait*, the calling process is transformed into a "zombie process." A zombie process is a process that only occupies a slot in the process table, it has no other space allocated either in user or kernel space. The process table slot that it occupies is partially overlaid with time accounting information (see <sys/proc.h>) to be used by *times*(S).

The parent process ID of all of the calling process' existing child processes and zombie processes is set to 1. This means the initialization process (see *intro*(S)) inherits each of these processes.

An accounting record is written on the accounting file if the system's accounting routine is enabled; see *acct*(S).

If the process ID, tty group ID, and process group ID of the calling process are equal, the **SIGHUP** signal is sent to each processes that has a process group ID equal to that of the calling process.

See Also

signal(S), wait(S)

Warning

See *Warning* in *signal*(S)

Name

fclose, fflush – Closes or flushes a stream.

Syntax

```
#include <stdio.h>

int fclose (stream)
FILE *stream;

int fflush (stream)
FILE *stream;
```

Description

Fclose causes any buffers for the named *stream* to be emptied, and the file to be closed. Buffers allocated by the standard input/output system are freed.

Fclose is performed automatically upon calling *exit*(S).

Fflush causes any buffered data for the named output *stream* to be written to that file. The stream remains open.

These functions return 0 for success, and EOF if any errors were detected.

See Also

close(S), fopen(S), setbuf(S)

Fcntl fails if one or more of the following is true:

Fildes is not a valid open file descriptor. [EBADF]

Cmd is F_DUPFD and 20 file descriptors are currently open. [EMFILE]

Cmd is F_DUPFD and *arg* is negative or greater than 20. [EINVAL]

Return Value

Upon successful completion, the value returned depends on *cmd* as follows:

F_DUPFD	A new file descriptor
F_GETFD	Value of flag (only the low-order bit is defined)
F_SETFD	Value other than – 1
F_GETFL	Value of file flags
F_SETFL	Value other than – 1

Otherwise, a value of – 1 is returned and *errno* is set to indicate the error.

See Also

close(S), exec(S), open(S)

Name

floor, fabs, ceil, fmod – Performs absolute value, floor, ceiling and remainder functions.

Syntax

```
#include <math.h>

double floor (x)
double x;

double ceil (x)
double x;

double fmod (x, y)
double x, y;

double fabs (x)
double x;
```

Description

Fabs returns $|x|$.

Floor returns the largest integer (as a double precision number) not greater than x.

Ceil returns the smallest integer not less than x.

Fmod returns the number f such that $x = iy + f$, for some integer i, and $0 \leq f < y$.

See Also

abs(S)

When a file is opened for update, both input and output may be done on the resulting stream. However, output may not be directly followed by input without an intervening ***fseek*** or ***rewind***, and input may not be directly followed by output without an intervening ***fseek***, ***rewind***, or an input operation which encounters the end of the file.

See Also

open(S), fclose(S)

Diagnostics

Fopen and ***freopen*** return the pointer NULL if ***filename*** cannot be accessed.

process. Otherwise, a value of – 1 is returned to the parent process, no child process is created, and *errno* is set to indicate the error.

See Also

exec(S), wait(S)

Name

frexp, ldexp, modf - Splits floating-point number into a mantissa and an exponent.

Syntax

```
double frexp (value, eptr)
double value;
int *eptr;

double ldexp (value, exp)
double value;

double modf (value, iptr)
double value, *iptr;
```

Description

Frexp returns the mantissa of a double *value* as a double quantity, *x*, of magnitude less than 1, and stores an integer *n* such that *value* = *x**2***n* indirectly through *eptr*.

Ldexp returns the quantity *value**(2***exp*).

Modf returns the positive fractional part of *value* and stores the integer part indirectly through *iptr*.

Name

gamma – Performs log gamma function.

Syntax

```
#include <math.h>
extern int signgam;

double gamma (x)
double x;
```

Description

Gamma returns $ln|\Gamma(|x|)|$. The sign of $\Gamma(|x|)$ is returned in the external integer *signgam*. The following C program fragment might be used to calculate Γ:

```
y = gamma (x);
if (y > 88.0)
        error ( );
y = exp (y) * signgam;
```

Diagnostics

For negative integer arguments, a huge value is returned, and *errno* is set to EDOM.

Name

getcwd – Gets pathname of current working directory.

Syntax

```
len = getcwd (pnbuf, maxlen);
int len;
char *pnbuf;
int maxlen;
```

Description

Getcwd determines the pathname of the current working directory and places it in *pnbuf*. The length excluding the terminating NULL is returned. *Maxlen* is the length of *pnbuf*. If the length of the (null-terminated) pathname exceeds *maxlen*, it is treated as an error.

Diagnostics

A length <= 0 is returned on error.

Notes

maxlen (and *pnbuf*) must be 1 more than the true maximum length of the pathname.

Name

getgrent, getgrgid, getgrnam, setgrent, endgrent – Get group file entry.

Syntax

```
#include <grp.h>

struct group *getgrent ( );

struct group *getgrgid (gid)
int gid;

struct group *getgrnam (name)
char *name;

int setgrent ( );

int endgrent ( );
```

Description

Getgrent, *getgrgid* and *getgrnam* each return pointers. The format of the structure is defined in **/usr/include/grp.h**.

The members of this structure are:

gr_name	The name of the group.
gr_passwd	The encrypted password of the group.
gr_gid	The numerical group ID.
gr_mem	Null-terminated vector of pointers to the individual member names.

Getgrent reads the next line of the file, so successive calls may be used to search the entire file. *Getgrgid* and *getgrnam* search from the beginning of the file until a matching *gid* or *name* is found, or end-of-file is encountered.

A call to *setgrent* has the effect of rewinding the group file to allow repeated searches. *Endgrent* may be called to close the group file when processing is complete.

Files

/etc/group

Name

getlogin – Gets login name.

Syntax

char *getlogin ( );

Description

Getlogin returns a pointer to the login name as found in /etc/utmp. It may be used in conjunction with *getpwnam* to locate the correct password file entry when the same user ID is shared by several login names.

If *getlogin* is called within a process that is not attached to a terminal device, it returns NULL. The correct procedure for determining the login name is to call *cuserid*, or to call *getlogin* and if it fails, to call *getpwuid*.

Files

/etc/utmp

See Also

cuserid(S), getgrent(S), getpwent(S), utmp(M)

Diagnostics

Returns NULL if name not found.

Notes

The return values point to static data whose content is overwritten by each call.

```
main (argc, argv)
int argc;
char **argv;
{
        int c;
        extern int optind;
        extern char *optarg;
        .
        .
        .
        while ((c = getopt (argc, argv, "abf:o:")) != EOF)
                switch (c) {
                case 'a':
                        if (bflg)
                                errflg++;
                        else
                                aflg++;
                        break;
                case 'b':
                        if (aflg)
                                errflg++;
                        else
                                bproc();
                        break;
                case 'f':
                        ifile = optarg;
                        break;
                case 'o':
                        ofile = optarg;
                        bufsiza = 512;
                        break;
                case '?':
                        errflg++;
                }
        if (errflg) {
                fprintf (stderr, "usage: . . . ");
                exit (S);
        }
        for( ; optind < argc; optind++) {
                if (access (argv[optind], 4)) {
        .
        .
        .
}
```

Name

getpid, getpgrp, getppid - Gets process, process group, and parent process IDs.

Syntax

int getpid ( )

int getpgrp ( )

int getppid ( )

Description

Getpid returns the process ID of the calling process.

Getpgrp returns the process group ID of the calling process.

Getppid returns the parent process ID of the calling process.

See Also

exec(S), fork(S), intro(S), setpgrp(S), signal(S)

Name

getpwent, getpwuid, getpwnam, setpwent, endpwent – Gets password file entry.

Syntax

```
#include <pwd.h>

struct passwd *getpwent ( );

struct passwd *getpwuid (uid)
int uid;

struct passwd *getpwnam (name)
char *name;

int setpwent ( );

int endpwent ( );
```

Description

Getpwent, *getpwuid* and *getpwnam* each returns a pointer to a structure containing the fields of an entry line in the password file. The structure of a password entry is defined in /usr/include/pwd.h.

The fields have meanings described in *passwd*(M). (The *pw_comment* field is unused.)

Getpwent reads the next line in the file, so successive calls can be used to search the entire file. *Getpwuid* and *getpwnam* search from the beginning of the file until a matching *uid* or *name* is found, or EOF is encountered.

A call to *setpwent* has the effect of rewinding the password file to allow repeated searches. *Endpwent* may be called to close the password file when processing is complete.

Files

/etc/passwd

See Also

getlogin(S), getgrent(S), passwd(M)

Name

gets, fgets – Gets a string from a stream.

Syntax

```
#include <stdio.h>

char *gets (s)
char *s;

char *fgets (s, n, stream)
char *s;
int n;
FILE *stream;
```

Description

Gets reads a string into *s* from the standard input stream stdin. The function replaces the newline character at the end of the string with a null character before copying to *s*. *Gets* returns a pointer to *s*.

Fgets reads characters from the *stream* until a newline character is encountered or until *n*– 1 characters have been read. The characters are then copied to the string *s*. A null character is automatically appended to the end of the string before copying. *Fgets* returns a pointer to *s*.

See Also

ferror(S), fopen(S), fread(S), getc(S), puts(S), scanf(S)

Diagnostics

Gets and *fgets* return the constant pointer NULL upon end-of-file or error.

Notes

Gets deletes the newline ending its input, but *fgets* keeps it.

Name

hypot, cabs - Determines Euclidean distance.

Syntax

```
#include <math.h>

double hypot (x, y)
double x, y;

double cabs (z)
struct {double x, y;} z;
```

Description

Hypot and *cabs* return

```
sqrt(x*x + y*y)
```

Both take precautions against unwarranted overflows.

See Also

sqrt in exp(S)

Name

kill – Sends a signal to a process or a group of processes.

Syntax

int kill (pid, sig)
int pid, sig;

Description

Kill sends a signal to a process or a group of processes. The process or group of processes to which the signal is to be sent is specified by *pid*. The signal that is to be sent is specified by *sig* and is either one from the list given in *signal*(S), or 0. If *sig* is 0 (the null signal), error checking is performed but no signal is actually sent. This can be used to check the validity of *pid*.

The effective user ID of the sending process must match the effective user ID of the receiving process unless, the effective user ID of the sending process is super-user, or the process is sending to itself.

The processes with a process ID of 0 and a process ID of 1 are special processes (see *intro*(S)) and will be referred to below as *proc0* and *proc1* respectively.

If *pid* is greater than zero, *sig* will be sent to the process whose process ID is equal to *pid*. *Pid* may equal 1.

If *pid* is 0, *sig* will be sent to all processes excluding *proc0* and *proc1* whose process group ID is equal to the process group ID of the sender.

If *pid* is – 1 and the effective user ID of the sender is not super-user, *sig* will be sent to all processes excluding *proc0* and *proc1* whose real user ID is equal to the effective user ID of the sender.

If *pid* is – 1 and the effective user ID of the sender is super-user, *sig* will be sent to all processes excluding *proc0* and *proc1*.

If *pid* is negative but not – 1, *sig* will be sent to all processes whose process group ID is equal to the absolute value of *pid*.

Kill will fail and no signal will be sent if one or more of the following are true:

Sig is not a valid signal number. [EINVAL]

No process can be found corresponding to that specified by *pid*. [ESRCH]

Name

l3tol, ltol3 – Converts between 3-byte integers and long integers.

Syntax

```
l3tol (lp, cp, n)
long *lp;
char *cp;
int n;

ltol3 (cp, lp, n)
char *cp;
long *lp;
int n;
```

Description

L3tol converts a list of *n* 3-byte integers packed into a character string pointed to by *cp* into a list of long integers pointed to by *lp*.

Ltol3 performs the reverse conversion from long integers (*lp*) to 3-byte integers (*cp*).

These functions are useful for file system maintenance where the block numbers are 3 bytes long.

See Also

filesystem(F)

Return Value

Upon successful completion, a value of 0 is returned. Otherwise, a value of – 1 is returned and *errno* is set to indicate the error.

See Also

ln(C)

Name

locking – Locks or unlocks a file region for reading or writing.

Syntax

```
locking(fildes, mode, size);
int fildes, mode;
long size;
```

Description

Locking allows a specified number of bytes in a file to be controlled by the locking process. Other processes which attempt to read or write a portion of the file containing the locked region may sleep until the area becomes unlocked depending upon the mode in which the file region was locked. A process that attempts to write to or read a file region that has been locked against reading and writing by another process (using the LK_LOCK or LK_NBLCK mode) will sleep until the region of the file has been released by the locking process. A process that attempts to write to a file region that has been locked against writing by another process (using the LK_RLCK or LK_NBRLCK mode) will sleep until the region of the file has been released by the locking process, but a read request for that file region will proceed normally.

A process that attempts to lock a region of a file that contains areas that have been locked by other processes will sleep if it has specified the LK_LOCK or LK_RLCK mode in its lock request, but will return with the error EACCES if it specified LK_NBLCK or LK_NBRLCK.

Fildes is the value returned from a successful *creat, open, dup,* or *pipe* system call.

Mode specifies the type of lock operation to be performed on the file region. The available values for mode are:

LK_UNLCK 0
: Unlocks the specified region. The calling process releases a region of the file it had previously locked.

LK_LOCK 1
: Locks the specified region. The calling process will sleep until the entire region is available if any part of it has been locked by a different process. The region is then locked for the calling process and no other process may read or write in any part of the locked region. (lock against read and write).

If a process has done more than one open on a file, *all* locks put on the file by that process will be released on the first close of the file.

Although no error is returned if locks are applied to special files or pipes, read/write operations on these types of files will ignore the locks. Locks may not be applied to a directory.

See Also

creat(S), open(S), read(S), write(S), dup(S), close(S), lseek(S)

Diagnostics

Locking returns the value (int) -1 if an error occurs. If any portion of the region has been locked by another process for the LK_LOCK and LK_RLCK actions and the lock request is to test only, *errno* is set to EACCES. If the file specified is a directory, *errno* is set to EACCES. If locking the region would cause a deadlock, *errno* is set to EDEADLOCK. If there are no more free internal locks, *errno* is set to EDEADLOCK.

Name

lsearch – Performs linear search and update.

Syntax

```
char *lsearch (key, base, nelp, width, compar)
char *key;
char *base;
int *nelp;
int width;
int (*compar)();
```

Description

Lsearch is a linear search routine generalized from Knuth (6.1) Algorithm Q. It returns a pointer into a table indicating the location at which a datum may be found. If the item does not occur, it is added at the end of the table. The first argument is a pointer to the datum to be located in the table. The second argument is a pointer to the base of the table. The third argument is the address of an integer containing the number of items in the table. It is incremented if the item is added to the table. The fourth argument is the width of an element in bytes. The last argument is the name of the comparison routine. It is called with two arguments which are pointers to the elements being compared. The routine must return zero if the items are equal, and nonzero otherwise.

Notes

Unpredictable events can occur if there is not enough room in the table to add a new item.

See Also

bsearch(S), qsort(S)

See Also

creat(S), dup(S), fcntl(S), open(S)

object.

Diagnostics

Malloc, *realloc* and *calloc* return a null pointer (0) if there is no available memory or if the area has been detectably corrupted by storing outside the bounds of a block. When *realloc* returns 0, the block pointed to by *ptr* may be destroyed.

shared memory file or a semaphore.

Mknod may be invoked only by the super-user for file types other than named pipe special.

Mknod will fail and the new file will not be created if one or more of the following are true:

The process' effective user ID is not super-user. [EPERM]

A component of the path prefix is not a directory. [ENOTDIR]

A component of the path prefix does not exist. [ENOENT]

The directory in which the file is to be created is located on a read-only file system. [EROFS]

The named file exists. [EEXIST]

Path points outside the process' allocated address space. [EFAULT]

Return Value

Upon successful completion a value of 0 is returned. Otherwise, a value of – 1 is returned and *errno* is set to indicate the error.

See Also

mkdir(C), mknod(C), chmod(S), creatsem(S), exec(S), sdget(S), umask(S), filesystem(F)

Notes

Semaphore files should be created with the *creatsem*(S) system call.

Share data files should be created with the *sdget*(S) system call.

Name

monitor – Prepares execution profile.

Syntax

```
monitor (lowpc, highpc, buffer, bufsize, nfunc)
int (*lowpc)( ), (*highpc)( );
short buffer[ ];
int bufsize, nfunc;
```

Description

Monitor is an interface to *profil*(S). *Lowpc* and *highpc* are the addresses of two functions; *buffer* is the address of a user-supplied array of *bufsize* short integers. *Monitor* arranges to record a histogram of periodically sampled values of the program counter, and of counts of calls of certain functions, in the buffer. The lowest address sampled is that of *lowpc* and the highest is just below *highpc*. At most *nfunc* call counts can be kept; only calls of functions compiled with the profiling option – p of *cc*(CP) are recorded. For the results to be significant, especially where there are small, heavily used routines, it is suggested that the buffer be no more than a few times smaller than the range of locations sampled.

To profile the entire program, it is sufficient to use

```
extern etext( );
...
monitor(2, etext, buf, bufsize, nfunc);
```

Etext lies just above all the program text.

To stop execution monitoring and write the results on the file **mon.out**, use

```
monitor(0);
```

prof(CP) can then be used to examine the results.

Files

mon.out

See Also

cc(CP), prof(CP), profil(S)

Name

mount - Mounts a file system.

Syntax

int mount (spec, dir, rwflag)
char *spec, *dir;
int rwflag;

Description

Mount requests that a removable file system contained on the block special file identified by *spec* be mounted on the directory identified by *dir*. *Spec* and *dir* are pointers to pathnames.

Upon successful completion, references to the file *dir* will refer to the root directory on the mounted file system.

The low-order bit of *rwflag* is used to control write permission on the mounted file system; if **1**, writing is forbidden, otherwise writing is permitted according to individual file accessibility.

Mount may be invoked only by the super-user.

Mount will fail if one or more of the following are true:

The effective user ID is not super-user. [EPERM]

Any of the named files does not exist. [ENOENT]

A component of a path prefix is not a directory. [ENOTDIR]

Spec is not a block special device. [ENOTBLK]

The device associated with *spec* does not exist. [ENXIO]

Dir is not a directory. [ENOTDIR]

Spec or *dir* points outside the process' allocated address space. [EFAULT]

Dir is currently mounted on, is someone's current working directory or is otherwise busy. [EBUSY]

The device associated with *spec* is currently mounted. [EBUSY]

Name

nap – Suspends execution for a short interval.

Syntax

```
long nap(period)
long period;
```

Description

The current process is suspended from execution for at least the number of milliseconds specified by *period*, or until a signal is received.

Return Value

On successful completion, a long integer indicating the number of milliseconds actually slept is returned. If the process recieved a signal while napping, the return value will be -1, and *errno* will be set to EINTR.

Notes

This function is driven by the system clock, which in most cases has a granularity of tens of milliseconds.

See Also

sleep(S)

Name

nlist – Gets entries from name list.

Syntax

```
#include <a.out.h>
nlist (filename, nl)
char *filename;
struct nlist nl[ ];
```

Description

Nlist examines the name list in the given executable output file and selectively extracts a list of values. The name list consists of an array of structures containing names, types and values. The list is terminated with a null name. Each name is looked up in the name list of the file. If the name is found, the type and value of the name are inserted in the next two fields. If the name is not found, both entries are set to 0. See *a.out*(F) for a discussion of the symbol table structure.

See Also

a.out(F), xlist(S)

Diagnostics

Nlist return – 1 and sets all type entries to 0 if the file cannot be read, is not an object file, or contains an invalid name list. Otherwise, *nlist* returns 0. A return value of 0 does not indicate that any or all symbols were found.

If O_NDELAY is clear:

The open will block until carrier is present.

O_APPEND If set, the file pointer will be set to the end of the file prior to each write.

O_CREAT If the file exists, this flag has no effect. Otherwise, the file's owner ID is set to the process' effective user ID, the file's group ID is set to the process' effective group ID, and the low-order 12 bits of the file mode are set to the value of *mode* modified as follows (see *creat*(S)):

All bits set in the process' file mode creation mask are cleared. See *umask*(S).

The "save text image after execution bit" of the mode is cleared. See *chmod*(S).

O_TRUNC If the file exists, its length is truncated to 0 and the mode and owner are unchanged.

O_EXCL If O_EXCL and O_CREAT are set, *open* will fail if the file exists.

O_SYNCW Every write to this file descriptor will be synchronous, that is, when the write system call completes data is guaranteed to have been written to disk.

Upon successful completion a nonnegative integer, the file descriptor, is returned.

The file pointer used to mark the current position within the file is set to the beginning of the file.

The new file descriptor is set to remain open across *exec* system calls. See *fcntl*(S).

No process may have more than 20 file descriptors open simultaneously.

The named file is opened unless one or more of the following are true:

A component of the path prefix is not a directory. [ENOTDIR]

O_CREAT is not set and the named file does not exist. [ENOENT]

A component of the path prefix denies search permission. [EACCES]

Name

opensem – Opens a semaphore.

Syntax

```
sem_num = opensem(sem_name);
int sem_num;
char *sem_name;
```

Description

Opensem opens a semaphore named by *sem_name* and returns the unique semaphore identification number *sem_num* used by *waitsem* and *sigsem*. *Createm* should always be called to initialize the semaphore before the first attempt to open it, or to reset the semaphore if it has become inconsistent due to an exiting process neglecting to do a sigsem after issuing a waitsem.

See Also

creatsem(S), waitsem(S), sigsem(S)

Diagnostics

Opensem returns the value – 1 if an error occurs. If the semaphore named does not exist, *errno* is set to ENOENT. If the file specified is not a semaphore file (i.e., a file previously created by a process using a call to *creatsem*), *errno* is set to ENOTNAM. If the semaphore has become invalid due to inappropriate use, *errno* is set to ENOTAVAIL.

Name

perror, sys_errlist, sys_nerr, errno – Sends system error messages.

Syntax

```
perror (s)
char *s;

int sys_nerr;
char *sys_errlist[ ];

int errno;
```

Description

Perror produces a short error message on the standard error, describing the last error encountered during a system call from a C program. First the argument string *s* is printed, then a colon, then the message and a newline. To be of most use, the argument string should be the name of the program that incurred the error. The error number is taken from the external variable *errno*, which is set when errors occur but not cleared when correct calls are made.

To simplify variant formatting of messages, the vector of message strings *sys_errlist* is provided; *errno* can be used as an index in this table to get the message string without the newline. *Sys_nerr* is the number of entries provided for in the table; it should be checked because new error codes may be added to the system before they are added to the table.

See Also

intro(S)

Name

popen, pclose – Initiates I/O to or from a process.

Syntax

```
#include <stdio.h>

FILE *popen (command, type)
char *command, *type;

int pclose (stream)
FILE *stream;
```

Description

The arguments to *popen* are pointers to null-terminated strings containing, respectively, a shell command line and an I/O mode, either "r" for reading or "w" for writing. *Popen* creates a pipe between the calling process and the command to be executed. The value returned is a stream pointer that can be used (as appropriate) to write to the standard input of the command or read from its standard output.

A stream opened by *popen* should be closed by *pclose*, which waits for the associated process to terminate and returns the exit status of the command. Because open files are shared between processes, a type "r" command may be used as an input filter, and a type "w" as an output filter.

See Also

pipe(S), wait(S), fclose(S), fopen(S), system(S)

Diagnostics

Popen returns a null pointer if files or processes cannot be created, or if the shell cannot be accessed.

Pclose returns – 1 if *stream* is not associated with a *popen*ed command.

Notes

Only one stream opened by *popen* can be in use at once. Buffered reading before opening an input filter may leave the standard input of that filter mispositioned. Similar problems with an output filter may be forestalled by careful buffer flushing; see *fclose*(S).

maximum number of significant digits for the g conversion, or the maximum number of characters to be printed from a string in s conversion. The precision takes the form of a period (.) followed by a decimal digit string: a null digit string is treated as zero.

An optional l specifying that a following d, o, u, x, or X conversion character applies to a long integer *arg*.

A character that indicates the type of conversion to be applied.

A field width or precision may be indicated by an asterisk (*) instead of a digit string. In this case, an integer *arg* supplies the field width or precision. The *arg* that is actually converted is not fetched until the conversion letter is seen, so the *args* specifying field width or precision must appear *before* the *arg* (if any) to be converted.

The flag characters and their meanings are:

- The result of the conversion will be left-justified within the field.

+ The result of a signed conversion will always begin with a sign (+ or -).

blank If the first character of a signed conversion is not a sign, a blank will be prepended to the result. This implies that if the blank and + flags both appear, the blank flag will be ignored.

This flag specifies that the value is to be converted to an "alternate form." For c, d, s, and u conversions, the flag has no effect. For o conversion, it increases the precision to force the first digit of the result to be a zero. For x (X) conversion, a nonzero result will have 0x (0X) prepended to it. For e, E, f, g, and G conversions, the result will always contain a decimal point, even if no digits follow the point (normally, a decimal point appears in the result of these conversions only if a digit follows it). For g and G conversions, trailing zeroes will *not* be removed from the result (which they normally are).

The conversion characters and their meanings are:

d,o,u,x,X The integer *arg* is converted to signed decimal, unsigned octal, decimal, or hexadecimal notation (x and X), respectively; the letters abcdef are used for x conversion and the letters ABCDEF for X conversion. The precision specifies the minimum number of digits to appear; if the value being converted can be represented in fewer digits, it will be expanded with leading zeroes. The default precision is 1. The result of converting a zero value with a precision of zero is a null string (unless the conversion is

Examples

To print a date and time in the form "Sunday, July 3, 10:02", where *weekday* and *month* are pointers to null-terminated strings:

```
printf("%s, %s %d, %.2d:%.2d", weekday, month, day, hour,
min);
```

To print π to five decimal places:

```
printf("pi = %.5f", 4*atan(1.0));
```

See Also

ecvt(S), putc(S), scanf(S)

Name

ptrace – Traces a process.

Syntax

```
int ptrace (request, pid, addr, data);
int request, pid, data;
```

Description

Ptrace provides a means by which a parent process may control the execution of a child process. Its primary use is in the implementation of breakpoint debugging; see *adb*(CP). The child process behaves normally until it encounters a signal (see *signal*(S) for the list), at which time it enters a stopped state and its parent is notified via *wait*(S). When the child is in the stopped state, its parent can examine and modify its "memory image" using *ptrace*. Also, the parent can cause the child either to terminate or continue, with the possibility of ignoring the signal that caused it to stop.

The *addr* argument is dependant on the underlying machine type, specifically the process memory model. On systems where the memory management mechanism provides a uniform and linear address space to user processes, the argument is declared as:

```
int *addr;
```

which is sufficient to address any location in the process' memory. On machines where the user address space is segmented (even if the particular program being traced has only one segment allocated), the form of the *addr* argument is:

```
struct {
        int offset;
        int segment;
} *addr;
```

which allows the caller to specify segment and offset in the process address space.

The *request* argument determines the precise action to be taken by *ptrace* and is one of the following:

0 This request must be issued by the child process if it is to be traced by its parent. It turns on the child's trace flag that stipulates that the child should be left in a stopped state upon receipt of a signal rather than the state specified by *func*; see *signal*(S). The *pid*, *addr*, and *data* arguments are ignored, and a return value is

7 This request causes the child to resume execution. If the *data* argument is 0, all pending signals including the one that caused the child to stop are canceled before it resumes execution. If the *data* argument is a valid signal number, the child resumes execution as if it had incurred that signal and any other pending signals are canceled. In a linear address space memory model, the value of *addr* must be (int *)1, or in a segmented address space the segment part of *addr* must be zero and the offset part of *addr* must be (int *)1. Upon successful completion, the value of *data* is returned to the parent. This request will fail if *data* is not 0 or a valid signal number, in which case a value of – 1 is returned to the parent process and the parent's *errno* is set to EIO.

8 This request causes the child to terminate with the same consequences as *exit*(S).

9 Execution continues as in request 7; however, as soon as possible after execution of at least one instruction, execution stops again. The signal number from the stop is SIGTRAP. This is part of the mechanism for implementing breakpoints. The exact implementation and behaviour is somewhat CPU dependant.

As indicated, these calls (except for request 0) can be used only when the subject process has stopped. The *wait* system call is used to determine when a process stops; in such a case the termination status returned by *wait* has the value 0177 to indicate stoppage rather than genuine termination.

To prevent security violations, *ptrace* inhibits the set-user-id facility on subsequent *exec*(S) calls. If a traced process calls *exec*, it will stop before executing the first instruction of the new image showing signal SIGTRAP.

Errors

Ptrace will in general fail if one or more of the following are true:

Request is an illegal number. [EIO]

Pid identifies a child that does not exist or has not executed a *ptrace* with request 0. [ESRCH]

Notes

The implementation and precise behaviour of this system call is inherently tied to the specific CPU and process memory model in use on a particular machine. Code using this call is likely to not be

Name

putc, putchar, fputc, putw – Puts a character or word on a stream.

Syntax

```
#include <stdio.h>

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

putchar (c)

int fputc (c, stream)
FILE *stream;

int putw (w, stream)
int w;
FILE *stream;
```

Description

Putc appends the character *c* to the named output *stream*. It returns the character written.

Putchar(*c*) is defined as *putc*(*c*,stdout).

Fputc behaves like *putc*, but is a genuine function rather than a macro; it may therefore be used as an argument. *Fputc* runs more slowly than *putc*, but takes less space per invocation.

Putw appends the word (i.e., integer) *w* to the output *stream*. *Putw* neither assumes nor causes special alignment in the file.

The standard stream stdout is normally buffered if and only if the output does not refer to a terminal; this default may be changed by *setbuf*(S). The standard stream stderr is by default unbuffered unconditionally, but use of *freopen* (see *fopen*(S)) will cause it to become unbuffered; *setbuf*, again, will set the state to whatever is desired. When an output stream is unbuffered information appears on the destination file or terminal as soon as written; when it is buffered many characters are saved up and written as a block. See *fflush* is *fclose*(S).

See Also

fclose(S), ferror(S), fopen(S), fread(S), getc(S), printf(S), puts(S)

Name

putpwent – Writes a password file entry.

Syntax

```
#include <pwd.h>

int putpwent (p, f)
struct passwd *p;
FILE *f;
```

Description

Putpwent is the inverse of *getpwent*(S). Given a pointer to a *passwd* structure created by *getpwent* (or *getpwuid* or *getpwnam*), *putpwent* writes a line on the stream *f*. The line matches the format of /etc/passwd.

See Also

passwd(M), getpwent(S)

Diagnostics

Putpwent returns nonzero if an error was detected during its operation, otherwise zero.

Name

qsort – Performs a sort.

Syntax

```
qsort (base, nel, width, compar)
char *base;
int nel, width;
int (*compar)( );
```

Description

Qsort is an implementation of the quicker-sort algorithm. The first argument is a pointer to the base of the data; the second is the number of elements; the third is the width of an element in bytes; the last is the name of the comparison routine. It is called with two arguments which are pointers to the elements being compared. The routine must return an integer less than, equal to, or greater than 0 according to how much the first argument is to be considered less than, equal to, or greater than the second.

See Also

sort(C), bsearch(S), lsearch(S), string(S)

Name

rdchk – Checks to see if there is data to be read.

Syntax

```
rdchk(fdes);
int fdes;
```

Description

Rdchk checks to see if a process will block if it attempts to read the file designated by *fdes*. *Rdchk* returns 1 if there is data to be read or if it is the end of the file (EOF). In this context, the proper sequence of calls using rdchk is:

```
if(rdchk(fildes) > 0)
        read(fildes, buffer, nbytes);
```

See Also

read(S)

Diagnostics

Rdchk returns -1 if an error occurs (e.g., EBADF), 0 if the process will block if it issues a *read* and 1 if it is okay to read. EBADF is returned if a *rdchk* is done on a semaphore file or if the file specified doesn't exist.

Read will fail if one or more of the following are true:

Fildes is not a valid file descriptor open for reading. [EBADF]

Buf points outside the allocated address space. [EFAULT]

Return Value

Upon successful completion a nonnegative integer is returned indicating the number of bytes actually read. Otherwise, a -1 is returned and *errno* is set to indicate the error.

See Also

creat(S), dup(S), fcntl(S), ioctl(S), open(S), pipe(S), tty(M)

Notes

Reading a region of a file locked with *locking* causes *read* to hang indefinitely until the locked region is unlocked.

to be applied. {m,} is analogous to {m,infinity}. The plus (+) and star (*) operations are equivalent to {1,} and {0,} respectively.

(...)$*n* The value of the enclosed regular expression is to be returned. The value will be stored in the (*n*+*1*)th argument following the subject argument. At present, at most ten enclosed regular expressions are allowed. *Regex* makes its assignments unconditionally.

(...) Parentheses are used for grouping. An operator, e.g. *, +, {}, can work on a single character or a regular expression enclosed in parenthesis. For example, (a*(cb+)*)$0.

By necessity, all the above defined symbols are special. They must, therefore, be escaped to be used as themselves.

Examples

Example 1:

```
char *cursor, *newcursor, *ptr;
        ...
newcursor = regex((ptr=regcmp("^\n",0)),cursor);
free(ptr);
```

This example will match a leading newline in the subject string pointed at by cursor.

Example 2:

```
char ret0[9];
char *newcursor, *name;
        ...
name = regcmp("([A-Za-z][A-za-z0-9_]{0,7})$0",0);
newcursor = regex(name,"123Testing321",ret0);
```

This example will match through the string "Testing3" and will return the address of the character after the last matched character (cursor+11). The string "Testing3" will be copied to the character array *ret0*.

Example 3:

```
#include "file.i"
char *string, *newcursor;
        ...
newcursor = regex(name,string);
```

This example applies a precompiled regular expression in **file.i** (see *regcmp*(CP)) against *string*.

Name

regexp – Performs regular expression compile and match functions.

Syntax

```
#define INIT <declarations>

#define GETC( ) <getc code>

#define PEEKC( ) <peekc code>

#define UNGETC(c) <ungetc code>

#define RETURN(pointer) <return code>

#define ERROR(val) <error code>

#include        <regexp.h>

char *compile(instring, expbuf, endbuf, eof)
char *instring, *expbuf, *endbuf;

int step(string, expbuf)
char *string, *expbuf;
```

Description

This entry describes general purpose regular expression matching routines in the form of *ed*(C), defined in /usr/include/regexp.h. Programs such as *ed*(C), *eed*(C), *grep*(C), *bs*(C), *expr*(C), etc., which perform regular expression matching use this source file. In this way, only this file need be changed to maintain regular expression compatibility.

The interface to this file is unpleasantly complex. Programs that include this file must have the following five macros declared before the "#include <regexp.h>" statement. These macros are used by the *compile* routine.

GETC( )	Returns the value of the next character in the regular expression pattern. Successive calls to GETC( ) should return successive characters of the regular expression.
PEEKC( )	Returns the next character in the regular expression. Successive calls to PEEKC( ) should return the same character (which should also be the next character returned by GETC( )).

The parameter *eof* is the character which marks the end of the regular expression. For example, in *ed*(C), this character is usually a /.

Each programs that includes this file must have a #define statement for INIT. This definition will be placed right after the declaration for the function *compile* and the opening curly brace ({). It is used for dependent declarations and initializations. It is most often used to set a register variable to point the beginning of the regular expression so that this register variable can be used in the declarations for GETC( ), PEEKC( ) and UNGETC( ). Otherwise it can be used to declare external variables that might be used by GETC( ), PEEKC( ) and UNGETC( ). See the example below of the declarations taken from *grep*(C).

There are other functions in this file which perform actual regular expression matching, one of which is the function *step*. The call to *step* is as follows:

step(string, expbuf)

The first parameter to *step* is a pointer to a string of characters to be checked for a match. This string should be null terminated.

The second parameter *expbuf* is the compiled regular expression which was obtained by a call of the function *compile*.

The function *step* returns one, if the given string matches the regular expression, and zero if the expressions do not match. If there is a match, two external character pointers are set as a side effect to the call to *step*. The variable set in *step* is *loc1*. This is a pointer to the first character that matched the regular expression. The variable *loc2*, which is set by the function *advance*, points the character after the last character that matches the regular expression. Thus if the regular expression matches the entire line, loc1 will point to the first character of *string* and *loc2* will point to the null at the end of *string*.

Step uses the external variable *circf* which is set by *compile* if the regular expression begins with ^. If this is set then *step* will only try to match the regular expression to the beginning of the string. If more than one regular expression is to be compiled before the the first is executed the value of *circf* should be saved for each compiled expression and *circf* should be set to that saved value before each call to *step*.

The function *advance* is called from *step* with the same arguments as *step*. The purpose of *step* is to step through the *string* argument and call *advance* until *advance* returns a one indicating a match or until the end of *string* is reached. If one wants to constrain *string* to the beginning of the line in all cases, *step* need not be called, simply call *advance*.

When *advance* encounters a * or \{ \} sequence in the regular expression it will advance its pointer to the string to be matched as

Name

sbrk, brk - Changes data segment space allocation.

Syntax

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

Description

Sbrk is used to dynamically change the amount of space allocated for the calling process' data segment; see *exec*(S). The change is made by resetting the process' break value. The break value is the address of the first location beyond the end of the data segment. The amount of allocated space increases as the break value increases.

Sbrk adds *incr* bytes to the break value and changes the allocated space accordingly. *Incr* can be negative, in which case the amount of allocated space is decreased.

Sbrk will fail without making any change in the allocated space if such a change would result in more space being allocated than is allowed by a system-imposed maximum (see *ulimit*(S)). [ENOMEM]

Return Value

Upon successful completion, *sbrk* and *brk* return pointers to the beginning of the allocated space. Otherwise, a value of -1 is returned and *errno* is set to indicate the error.

See Also

exec(S)

% A single % is expected in the input at this point; no assignment is done.

d A decimal integer is expected; the corresponding argument should be an integer pointer.

o An octal integer is expected; the corresponding argument should be an integer pointer.

x A hexadecimal integer is expected; the corresponding argument should be an integer pointer.

s A character string is expected; the corresponding argument should be a character pointer pointing to an array of characters large enough to accept the string and a terminating **\0**, which will be added automatically. The input field is terminated by a space character or a newline.

c A character is expected; the corresponding argument should be a character pointer. The normal skip over space characters is suppressed in this case; to read the next nonspace character, use %1s. If a field width is given, the corresponding argument should refer to a character array; the indicated number of characters is read.

e,f A floating-point number is expected; the next field is converted accordingly and stored through the corresponding argument, which should be a pointer to a *float*. The input format for floating-point numbers is an optionally signed string of digits, possibly containing a decimal point, followed by an optional exponent field consisting of an E or an e, followed by an optionally signed integer.

[Indicates a string that is not to be delimited by space characters. The left bracket is followed by a set of characters and a right bracket; the characters between the brackets define a set of characters making up the string. If the first character is not a caret (^), the input field consists of all characters up to the first character that is not in the set between the brackets; if the first character after the left bracket is a ^, the input field consists of all characters up to the first character that is in the set of the remaining characters between the brackets. The corresponding argument must point to a character array.

The conversion characters d, o, and x may be capitalized and/or preceded by l to indicate that a pointer to long rather than to int is in the argument list. Similarly, the conversion characters e and f may be capitalized and/or preceded by l to indicate that a pointer to double rather than to float is in the argument list. The character h will, some time in the future, indicate short data items.

Scanf conversion terminates at EOF, at the end of the control string, or when an input character conflicts with the control string. *In the*

Name

sdenter, sdleave – Synchronizes access to a shared data segment.

Syntax

```
#include <sd.h>

int sdenter(addr,flags)
char *addr;
int flags;

int sdleave(addr)
char *addr;
```

Description

Sdenter is used to indicate that the current process is about to access the contents of a shared data segment. The actions performed depend on the value of *flags*. *Flags* values are formed by OR–ing together entries from the following list:

SD_NOWAIT — If another process has called *sdenter* but not *sdleave* for the indicated segment, and the segment was not created with the SD_UNLOCK flag set, return an error instead of waiting for the segment to become free.

SD_WRITE — Indicates that the process intends to modify the data. If SD_WRITE isn't specified changes made to data are not guarenteed to be reflected in other proceses.

Sdleave is used to indicate that the current process is done modifying the contents of a shared data segment.

Only changes made between invocatations of *sdenter* and *sdleave* are guaranteed to be reflected in other processes. *Sdenter* and *sdleave* are very fast; consequently, it is recommended that they be called frequently rather than leave *sdenter* in effect for any period of time. In particular, system calls should be avoided between *sdenter* and *sdleave* calls.

The *fork* system call is forbidden between calls to *sdenter* and *sdleave* if the segment was created without the SD_UNLOCK flag.

Name

sdget – Attachs and detachs a shared data segment.

Syntax

```
#include <sd.h>

char *sdget(path, flags, [size, mode])
char *path;
int flags, mode;
long size;

int sdfree(addr);
char *addr;
```

Description

Sdget attachs a shared data segment to the data space of the current process. The actions performed are controlled by the value of *flags*. *Flags* values are constructed by OR-ing flags from the following list:

SD_RDONLY
: Attach the segment for reading only.

SD_WRITE
: Attach the segment for both reading and writing.

SD_CREAT
: If the segment named by *path* exists, this flag has no effect. Otherwise, the segment is created according to the values of *size* and *mode*. Read and write access to the segment is granted to other processes based on the permissions passed in *mode*, and functions the same as those for regular files. Execute permission is meaningless. The segment is initialized to contain all zeroes.

SD_UNLOCK
: If the segment is created because of this call, the segment will be made so that more than one process can be between sdenter and sdleave calls.

Sdfree detachs the current process from the shared data segment that is attached at the specified address. If the current process has done an *sdenter* but not a *sdleave* for the specified segment, an *sdleave* will be done before detaching the segment.

When no process remains attached to the segment, the contents of that segment disappear, and no process can attach to the segment without creating it by using the SD_CREAT flag in *sdget*.

Name

sdgetv, sdwaitv – Synchronizes shared data access.

Syntax

```
#include <sd.h>

int sdgetv(addr)
int sdwaitv(addr, vnum)
char *addr;
int vnum;
```

Description

Sdgetv and *sdwaitv* may be used to synchronize cooperating processes that are using shared data segments. The return value of both routines is the version number of the shared data segment attached to the process at address *addr*. The version number of a segment changes whenever some process does an *sdleave* for that segment.

Sdgetv simply returns the version number of the indicated segment.

Sdwaitv forces the current process to sleep until the version number for the indicated segment is no longer equal to *vnum*.

Return Value

Upon successful completion, both *sdgetv* and *sdwaitv* return a positive integer that is the current version number for the indicated shared data segment. Otherwise, a value of -1 is returned, and *errno* is set to indicate the error.

See Also

sdenter(S), sdget(S)

Name

setjmp, longjmp – Performs a nonlocal "goto".

Syntax

```
#include <setjmp.h>

int setjmp (env)
jmp_buf env;

int longjmp (env, val)
jmp_buf env;
```

Description

These routines are useful for dealing with errors and interrupts encountered in a low-level subroutine of a program.

Setjmp saves its stack environment in *env* for later use by *longjmp*. It returns value 0.

Longjmp restores the environment saved by the last call of *setjmp*. It then returns in such a way that execution continues as if the call of *setjmp* had just returned the value *val* to the corresponding call to *setjmp*. The routine which calls *setjmp* must not itself have returned in the interim. *Longjmp* cannot return the value 0. If *longjmp* is invoked with a second argument of 0, it will return 1. All accessible data have values as of the time *longjmp* was called. The only exception to this are register variables. The value of register variables are undefined in the routine that called *setjmp* when the corresponding *longjmp* is invoked.

See Also

signal(S)

Name

setuid, setgid – Sets user and group IDs.

Syntax

```
int setuid (uid)
int uid;

int setgid (gid)
int gid;
```

Description

Setuid is used to set the real user ID and effective user ID of the calling process.

Setgid is used to set the real group ID and effective group ID of the calling process.

If the effective user ID of the calling process is super-user, the real user (group) ID and effective user (group) ID are set to *uid* (*gid*).

If the effective user ID of the calling process is not super-user, but its real user (group) ID is equal to *uid* (*gid*), the effective user (group) ID is set to *uid* (*gid*).

Setuid will fail if the real user (group) ID of the calling process is not equal to *uid* (*gid*) and its effective user ID is not super-user. [EPERM]

Return Value

Upon successful completion, a value of 0 is returned. Otherwise, a value of – 1 is returned and *errno* is set to indicate the error.

See Also

getuid(S), intro(S)

Name

signal – Specifies what to do upon receipt of a signal.

Syntax

```
#include <signal.h>

int (*signal (sig, func))( )
int sig;
int (*func)( );
```

Description

Signal allows the calling process to choose one of three ways in which it is possible to handle the receipt of a specific signal. *Sig* specifies the signal and *func* specifies the choice.

Sig can be assigned any one of the following except SIGKILL:

SIGHUP	01	Hangup
SIGINT	02	Interrupt
SIGQUIT	03*	Quit
SIGILL	04*	Illegal instruction (not reset when caught)
SIGTRAP	05*	Trace trap (not reset when caught)
SIGIOT	06*	I/O trap instruction
SIGEMT	07*	Emulator trap instruction
SIGFPE	08*	Floating-point exception
SIGKILL	09	Kill (cannot be caught or ignored)
SIGBUS	10*	Bus error
SIGSEGV	11*	Segmentation violation
SIGSYS	12*	Bad argument to system call
SIGPIPE	13	Write on a pipe with no one to read it
SIGALRM	14	Alarm clock
SIGTERM	15	Software termination signal
SIGUSR1	16	User-defined signal 1
SIGUSR2	17	User-defined signal 2
SIGCLD	18	Death of a child (see *Warning* below)
SIGPWR	19	Power fail (see *Warning* below)

See below for the significance of the asterisk in the above list.

Func is assigned one of three values: SIG_DFL, SIG_IGN, or a *function address*. The actions prescribed by these values of are described below.

The SIG_DFL value causes termination of the process upon receipt of a signal. Upon receipt of the signal *sig*, the receiving process is to be terminated with the following consequences:

2. When a signal that is to be caught occurs during a *read*, a *write*, an *open*, or an *ioctl* system call on a slow device (like a terminal; but not a file), during a *pause* system call, or during a *wait* system call that does not return immediately due to the existence of a previously stopped or zombie process, the signal catching function will be executed and then the interrupted system call will return a – 1 to the calling process with *errno* set to EINTR.

3. Note that the signal SIGKILL cannot be caught.

A call to *signal* cancels a pending signal *sig* except for a pending SIGKILL signal.

Signal will fail if one or more of the following are true:

Sig is an illegal signal number, including SIGKILL. [EINVAL]

Func points to an illegal address. [EFAULT]

Return Value

Upon successful completion, *signal* returns the previous value of *func* for the specified signal *sig*. Otherwise, a value of **– 1** is returned and *errno* is set to indicate the error.

See Also

kill(C), kill(S), pause(S), ptrace(S), wait(S), setjmp(S).

Warning

Two other signals that behave differently than the signals described above exist in this release of the system; they are:

SIGCLD	18	Death of a child (not reset when caught)
SIGPWR	19	Power fail (not reset when caught)

There is no guarantee that, in future releases of XENIX, these signals will continue to behave as described below; they are included only for compatibility with other versions of XENIX. Their use in new programs is strongly discouraged.

For these signals, *func* is assigned one of three values: **SIG_DFL**, **SIG_IGN**, or a *function address*. The actions prescribed by these values of are as follows:

SIG_DFL - ignore signal
The signal is to be ignored.

Name

sigsem – Signals a process waiting on a semaphore.

Syntax

```
sigsem(sem_num);
int sem_num;
```

Description

Sigsem signals a process that is waiting on the semaphore *sem_num* that it may proceed and use the resource governed by the semaphore. *Sigsem* is used in conjunction with *waitsem*(S) to allow synchronization of processes wishing to access a resource. One or more processes may *waitsem* on the given semaphore and will be put to sleep until the process which currently has access to the resource issues a *sigsem* call. If there are any waiting processes, *sigsem* causes the process which is next in line on the semaphore's queue to be rescheduled for execution. The semaphore's queue is organized in first in first out (FIFO) order.

See Also

creatsem(S), opensem(S), waitsem(S)

Diagnostics

Sigsem returns the value (int) -1 if an error occurs. If *sem_num* does not refer to a semaphore type file, *errno* is set to ENOTNAM. If *sem_num* has not been previously opened by *opensem*, *errno* is set to EBADF. If the process issuing a sigsem call is not the current "owner" of the semaphore (i.e., if the process has not issued a waitsem call before the sigsem), *errno* is set to ENAVAIL.

Name

sleep – Suspends execution for an interval.

Syntax

unsigned sleep (seconds)
unsigned seconds;

Description

The current process is suspended from execution for the number of *seconds* specified by the argument. The actual suspension time may be less than that requested for because scheduled wakeups occur at fixed 1-second intervals, and any caught signal will terminate the *sleep* following execution of that signal's catching routine. Also, the suspension time may be longer than requested by an arbitrary amount due to the scheduling of other activity in the system. The value returned by *sleep* will be the "unslept" amount (the requested time minus the time actually slept) in case the caller had an alarm set to go off earlier than the end of the requested *sleep* time, or premature arousal due to another caught signal.

The routine is implemented by setting an alarm signal and pausing until it (or some other signal) occurs. The previous state of the alarm signal is saved and restored. The calling program may have set up an alarm signal before calling *sleep*; if the *sleep* time exceeds the time till such alarm signal, the process sleeps only until the alarm signal would have occurred, and the caller's alarm catch routine is executed just before the *sleep* routine returns, but if the *sleep* time is less than the time till such alarm, the prior alarm time is reset to go off at the same time it would have gone off without the intervening *sleep*.

See Also

alarm(S), nap(S), pause(S), signal(S)

Notes

There are some additional signals with numbers outside the range 1 through 15 that are used by the standard C library to indicate error conditions. Thus, some signal numbers outside the range 1 through 15 are legal, although their use may interfere with the operation of the standard C library.

st_atime Time when file data was last accessed. Changed by the following system calls: *creat*(S), *mknod*(S), *pipe*(S), *utime*(S), and *read*(S).

st_mtime Time when data was last modified. Changed by the following system calls: *creat*(S), *mknod*(S), *pipe*(S), *utime*(S), and *write*(S).

st_ctime Time when file status was last changed. Changed by the following system calls: *chmod*(S), *chown*(S), *creat*(S), *link*(S), *mknod*(S), *pipe*(S), *utime*(S), and *write*(S).

st_rdev Device indentification. In the case of block and character special files this contains the device major and minor numbers; in the case of shared memory and semaphores, it contains the type code. The file /usr/include/sys/types.h contains the macros *major*() and *minor*() for extracting major and minor numbers from *st_rdev*. See /usr/include/sys/stat.h for the semaphore and shared memory type code values S_INSEM and S_INSHD.

Stat will fail if one or more of the following are true:

A component of the path prefix is not a directory. [ENOTDIR]

The named file does not exist. [ENOENT]

Search permission is denied for a component of the path prefix. [EACCES]

Buf or *path* points to an invalid address. [EFAULT]

Fstat will fail if one or more of the following are true:

Fildes is not a valid open file descriptor. [EBADF]

Buf points to an invalid address. [EFAULT]

Return Value

Upon successful completion a value of 0 is returned. Otherwise, a value of – 1 is returned and *errno* is set to indicate the error.

See Also

chmod(S), chown(S), creat(S), link(S), mknod(S), time(S), unlink(S)

See Also

open(S), close(S), read(S), write(S), ctermid(S), cuserid(S), fclose(S), ferror(S), fopen(S), fread(S), fseek(S), getc(S), gets(S), popen(S), printf(S), putc(S), puts(S), scanf(S), setbuf(S), system(S), tmpnam(S)

Diagnostics

Invalid stream pointers can cause grave disorder, possibly including program termination. Individual function descriptions describe the possible error conditions.

Name

string, strcat, strncat, strcmp, strncmp, strcpy, strncpy, strlen, strchr, strrchr, strpbrk, strspn, strcspn, strtok, strdup – Perform string operations.

Syntax

```
char *strcat (s1, s2)
char *s1, *s2;

char *strncat (s1, s2, n)
char *s1, *s2;
int n;

int strcmp (s1, s2)
char *s1, *s2;

int strncmp (s1, s2, n)
char *s1, *s2;
int n;

char *strcpy (s1, s2)
char *s1, *s2;

char *strncpy (s1, s2, n)
char *s1, *s2;
int n;

int strlen (s)
char *s;

char *strchr (s, c)
char *s, c;

char *strrchr (s, c)
char *s, c;

char *strpbrk (s1, s2)
char *s1, *s2;

int strspn (s1, s2)
char *s1, *s2;

int strcspn (s1, s2)
char *s1, *s2;

char *strtok (s1, s2)
char *s1, *s2;

char *strdup (s)
char *s;
```

Notes

Strcmp uses native character comparison, which is signed on some machines, unsigned on others.

All string movement is performed character by character starting at the left. Thus overlapping moves toward the left will work as expected, but overlapping moves to the right may yield surprises.

Name

sync – Updates the super-block.

Syntax

sync ( )

Description

Sync causes all information in memory that should be on disk to be written out. This includes modified super-blocks, modified inodes, and delayed block I/O.

It should be used by programs which examine a file system, for example *fsck*(C), *df*(C), etc.

The writing, although scheduled, is not necessarily complete upon return from *sync*.

See Also

sync(C)

Name

tgetent, tgetnum, tgetflag, tgetstr, tgoto, tputs – Performs terminal functions.

Syntax

```
char PC;
char *BC;
char *UP;
short ospeed;

tgetent(bp, name)
char *bp, *name;

tgetnum(id)
char *id;

tgetflag(id)
char *id;

char *tgetstr (id, area)
char *id, **area;

char *tgoto (cm, destcol, destline)
char *cm;

tputs(cp, affcnt, outc)
register char *cp;
int affcnt;
int (*outc)();
```

Description

These functions extract and use capabilities from the terminal capability data base *termcap*(M). These are low level routines; see *curses*(S) for a higher level package.

Tgetent extracts the entry for terminal *name* into the buffer at *bp*. *Bp* should be a character buffer of size 1024 and must be retained through all subsequent calls to *tgetnum, tgetflag,* and *tgetstr*. *Tgetent* returns – 1 if it cannot open the *termcap* file, 0 if the terminal name given does not have an entry, and 1 if all goes well. It will look in the environment for a TERMCAP variable. If found, and the value does not begin with a slash, and the terminal type name is the same as the environment string TERM, the TERMCAP string is used instead of reading the termcap file. If it does begin with a slash, the string is used as a pathname rather than */etc/termcap*. This can speed up entry into programs that call *tgetent*, as well as to help debug new terminal descriptions or to make one for your terminal if you can't

Name

time, ftime – Gets time and date.

Syntax

```
time_t time ((long *) 0)

time_t time (tloc)
time_t *tloc;

#include <sys/types.h>
#include <sys/timeb.h>

ftime(tp)
struct timeb *tp;
```

Description

Time returns the current system time in seconds since 00:00:00 GMT, January 1, 1970.

If *tloc* (taken as an integer) is nonzero, the return value is also stored in the location to which *tloc* points.

Ftime returns the time in a structure (see below under *Return Value*.)

Time will fail if *tloc* points to an illegal address. [EFAULT] Likewise, *ftime* will fail if *tp* points to an illegal address. [EFAULT]

Return Value

Upon successful completion, *time* returns the value of time. Otherwise, a value of – 1 is returned and *errno* is set to indicate the error.

The *ftime* entry fills in a structure pointed to by its argument, as defined by <sys/timeb.h>:

```
/*
 * Structure returned by ftime system call
 */
struct timeb {
        time_t  time;
        unsigned short millitm;
        short   timezone;
        short   dstflag;
};
```

Name

times – Gets process and child process times.

Syntax

```
#include <times.h>

long times (buffer)
struct tmbuf {
        long utime;
        long stime;
        long cutime;
        long cstime;
} buffer;
```

Description

Times fills the structure pointed to by *buffer* with time-accounting information. This information comes from the calling process and each of its terminated child processes for which it has executed a *wait*(S).

All times are in clock ticks where a tick is some fraction of a second defined in *machine*(M).

Utime is the CPU time used while executing instructions in the user space of the calling process.

Stime is the CPU time used by the system on behalf of the calling process.

Cutime is the sum of the *utimes* and *cutimes* of the child processes.

Cstime is the sum of the *stimes* and *cstimes* of the child processes.

Times will fail if *buffer* points to an illegal address. [EFAULT]

Return Value

Upon successful completion, *times* returns the elapsed real time, in clock ticks, since an arbitrary point in the past, such as the system start-up time. This point does not change from one invocation of *times* to another. If *times* fails, a – 1 is returned and *errno* is set to indicate the error.

See Also

exec(S), fork(S), time(S), wait(S), machine(M)

Name

tmpnam – Creates a name for a temporary file.

Syntax

```
#include <stdio.h>

char *tmpnam (s)
char *s;
```

Description

Tmpnam generates a filename that can safely be used for a temporary file. If (int)*s* is zero, *tmpnam* leaves its result in an internal static area and returns a pointer to that area. The next call to *tmpnam* will destroy the contents of the area. If (int)*s* is nonzero, *s* is assumed to be the address of an array of at least L_tmpnam bytes; *tmpnam* places its result in that array and returns *s* as its value.

Tmpnam generates a different filename each time it is called.

Files created using *tmpnam* and either *fopen* or *creat* are only temporary in the sense that they reside in a directory intended for temporary use, and their names are unique. It is the user's responsibility to use *unlink* (S) to remove the file when its use is ended.

See Also

creat(S), unlink(S), fopen(S), mktemp(S)

Notes

If called more than 17,576 times in a single process, *tmpnam* will start recycling previously used names.

Between the time a filename is created and the file is opened, it is possible for some other process to create a file with the same name. This can never happen if that other process is using *tmpnam* or *mktemp*, and the filenames are chosen so as to render duplication by other means unlikely.

Name

ttyname, isatty - Finds the name of a terminal.

Syntax

char *ttyname (fildes)

int isatty (fildes)

Description

Ttyname returns a pointer to the null-terminated pathname of the terminal device associated with file descriptor *fildes*.

Isatty returns **1** if *fildes* is associated with a terminal device, **0** otherwise.

Files

/dev/*

Diagnostics

Ttyname returns a null pointer (**0**) if *fildes* does not describe a terminal device in directory /dev.

Notes

The return value points to static data whose content is overwritten by each call.

Name

umask – Sets and gets file creation mask.

Syntax

```
int umask (cmask)
int cmask;
```

Description

Umask sets the process' file mode creation mask to *cmask* and returns the previous value of the mask. Only the low-order 9 bits of *cmask* and the file mode creation mask are used.

Return Value

The previous value of the file mode creation mask is returned.

See Also

mkdir(C), mknod(C), sh(C), chmod(S), mknod(S), open(S)

Name

uname – Gets name of current XENIX system.

Syntax

```
#include <sys/utsname.h>

int uname (name)
struct utsname *name;
```

Description

Uname stores information identifying the current XENIX system in the structure pointed to by *name*.

Uname uses the structure defined in **<sys/utsname.h>**:

```
struct utsname {
        char    sysname[9];
        char    nodename[9];
        char    release[9];
        char    version[9];
        unsigned short sysorigin;
        unsigned short sysoem;
        long    sysserial;
};
```

Uname returns a null-terminated character string naming the current XENIX system in the character array *syename*. Similarly, *nodename* contains the name that the system is known by on a communications network. *Release* and *version* further identify the operating system. *Sysorigin* and *sysoem* identify the source of the XENIX version. *Sysserial* is a software serial number which may be zero if unused.

Uname will fail if *name* points to an invalid address. [EFAULT]

Return Value

Upon successful completion, a nonnegative value is returned. Otherwise, – 1 is returned and *errno* is set to indicate the error.

Name

ungetc – Pushes character back into input stream.

Syntax

```
#include <stdio.h>

int ungetc (c, stream)
char c;
FILE *stream;
```

Description

Ungetc pushes the character *c* back on an input stream. The character will be returned by the next *getc* call on that stream. *Ungetc* returns *c*.

One character of pushback is guaranteed provided something has been read from the stream and the stream is actually buffered. Attempts to push EOF are rejected.

Fseek(S) erases all memory of pushed back characters.

See Also

fseek(S), getc(S), setbuf(S)

Diagnostics

Ungetc returns EOF if it can't push a character back.

Return Value

Upon successful completion, a value of 0 is returned. Otherwise, a value of – 1 is returned and *errno* is set to indicate the error.

See Also

rm(C), close(S), link(S), open(S)

Name

utime – Sets file access and modification times.

Syntax

```
#include <sys/types.h>
int utime (path, times)
char *path;
struct utimbuf *times;
```

Description

Path points to a pathname naming a file. *Utime* sets the access and modification times of the named file.

If *times* is NULL, the access and modification times of the file are set to the current time. A process must be the owner of the file or have write permission to use *utime* in this manner.

If *times* is not NULL, *times* is interpreted as a pointer to a *utimbuf* structure and the access and modification times are set to the values contained in the designated structure. Only the owner of the file or the super-user may use *utime* this way.

The times in the following structure are measured in seconds since 00:00:00 GMT, Jan. 1, 1970.

```
struct   utimbuf {
         time_t   actime;      /* access time */
         time_t   modtime;     /* modification time */
};
```

Utime will fail if one or more of the following are true:

The named file does not exist. [ENOENT]

A component of the path prefix is not a directory. [ENOTDIR]

Search permission is denied by a component of the path prefix. [EACCES]

The effective user ID is not super-user and not the owner of the file and *times* is not NULL. [EPERM]

The effective user ID is not super-user and not the owner of the file and *times* is NULL and write access is denied. [EACCES]

The file system containing the file is mounted read-only. [EROFS]

Name

wait – Waits for a child process to stop or terminate.

Syntax

```
int wait (stat_loc)
int *stat_loc;

int wait ((int *)0)
```

Description

Wait suspends the calling process until it receives a signal that is to be caught (see *signal*(S)), or until any one of the calling process' child processes stops in a trace mode (see *ptrace*(S)) or terminates. If a child process stopped or terminated prior to the call on *wait*, return is immediate.

If *stat_loc* (taken as an integer) is nonzero, 16 bits of information called "status" are stored in the low-order 16 bits of the location pointed to by *stat_loc*. *Status* can be used to differentiate between stopped and terminated child processes and if the child process terminated, status identifies the cause of termination and passes useful information to the parent. This is accomplished in the following manner:

If the child process stopped, the high-order 8 bits of status will be zero and the low-order 8 bits will be set equal to 0177.

If the child process terminated due to an *exit* call, the low-order 8 bits of status will be zero and the high-order 8 bits will contain the low-order 8 bits of the argument that the child process passed to *exit*; see *exit*(S).

If the child process terminated due to a signal, the high-order 8 bits of status will be zero and the low-order 8 bits will contain the number of the signal that caused the termination. In addition, if the low-order seventh bit (i.e., bit 200) is set, a "core image" will have been produced; see *signal*(S).

If a parent process terminates without waiting for its child processes to terminate, the parent process ID of each child process is set to 1. This means the initialization process inherits the child processes; see *intro*(S).

Name

waitsem, nbwaitsem - Awaits and checks access to a resource governed by a semaphore.

Syntax

waitsem(sem_num);
int sem_num;

nbwaitsem(sem_num);
int sem_num;

Description

Waitsem gives the calling process access to the resource governed by the semaphore *sem_num.* If the resource is in use by another process, *waitsem* will put the process to sleep until the resource becomes available; *nbwaitsem* will return the error ENAVAIL. *Waitsem* and *nbwaitsem* are used in conjunction with *sigsem* to allow synchronization of processes wishing to access a resource. One or more processes may *waitsem* on the given semaphore and will be put to sleep until the process which currently has access to the resource issues *sigsem. Sigsem* causes the process which is next in line on the semaphore's queue to be rescheduled for execution. The semaphore's queue is organized in first in first out (FIFO) order.

See Also

creatsem(S), opensem(S), sigsem(S)

Diagnostics

Waitsem returns the value (int) -1 if an error occurs. If *sem_num* has not been previously opened by a call to *opensem* or *creatsem*, *errno* is set to EBADF. If *sem_num* does not refer to a semaphore type file, *errno* is set to ENOTNAM. All processes waiting (or attempting to wait) on the semaphore when the process controlling the semaphore exits without relinquishing control (thereby leaving the resource in an undeterminate state) return with *errno* set to ENAVAIL.

below).

If the file being written is a pipe (or FIFO), no partial writes will be permitted. Thus, the write will fail if a write of *nbyte* bytes would exceed a limit.

If the file being written is a pipe (or FIFO) and the O_NDELAY flag of the file flag word is set, then write to a full pipe (or FIFO) will return a count of 0. Otherwise (O_NDELAY clear), writes to a full pipe (or FIFO) will block until space becomes available.

Return Value

Upon successful completion the number of bytes actually written is returned. Otherwise, – 1 is returned and *errno* is set to indicate the error.

See Also

creat(S), dup(S), lseek(S), open(S), pipe(S), ulimit(S)

Notes

Writing a region of a file locked with *locking* causes *write* to hang indefinitely until the locked region is unlocked.

Diagnostics

Xlist returns -1 and sets all type entries to zero if the file cannot be read, is not an object file, or contains an invalid name list. Otherwise, *xlist* returns zero. A return value of zero does *not* indicate that any or all of the given symbols were found.

CONTENTS

File Formats (F)

Index

Name

intro – Introduction to file formats.

Description

This section outlines the formats of various files. Usually, these structures can be found in the directories **/usr/include** or **/usr/include/sys**.

Name

acct – Format of per-process accounting file.

Description

Files produced as a result of calling *acct*(S) have records in the form defined by **<sys/acct.h>**.

In *ac_flag*, the AFORK flag is turned on by each *fork*(S) and turned off by an *exec*(S). The *ac_comm* field is inherited from the parent process and is reset by any *exec*. Each time the system charges the process with a clock tick, it also adds the current process size to *ac_mem* computed as follows:

(data size) + (text size) / (number of in-core processes using text)

The value of *ac_mem/ac_stime* can be viewed as an approximation to the mean process size, as modified by text-sharing.

See Also

acct(C), acctcom(C), acct(S)

Notes

The *ac_mem* value for a short-lived command gives little information about the actual size of the command, because *ac_mem* may be incremented while a different command (e.g., the shell) is being executed by the process.

Name

checklist – List of file systems processed by *fsck*.

Description

The */etc/checklist* file contains a list of the file systems to be checked when *fsck*(C) is invoked without arguments. The list contains at most 15 *special file* names. Each *special file* name must be on a separate line and must correspond to a file system.

See Also

fsck(C)

Name

cpio – Format of cpio archive.

Description

The *header* structure, when the c option is not used, is:

```
struct {
        short   h_magic,
                h_dev,
                h_ino,
                h_mode,
                h_uid,
                h_gid,
                h_nlink,
                h_rdev,
                h_mtime[2],
                h_namesize,
                h_filesize[2];
        char    h_name[h_namesize rounded to word];
} Hdr;
```

When the c option is used, the *header* information is described by the statement below:

```
sscanf(Chdr,"%6o%6o%6o%6o%6o%6o%6o%11lo%6o%6o%s",
        &Hdr.h_magic,&Hdr.h_dev,&Hdr.h_ino,&Hdr.h_mode,
        &Hdr.h_uid,&Hdr.h_gid,&Hdr.h_nlink,&Hdr.h_rdev,
        &Longtime,&Hdr.h_namesize,&Longfile,Hdr.h_name);
```

Longtime and *Longfile* are equivalent to *Hdr.h_mtime* and *Hdr.h_filesize*, respectively. The contents of each file is recorded in an element of the array of varying length structures, *archive*, together with other items describing the file. Every instance of *h_magic* contains the constant **070707** (octal). The items *h_dev* through *h_mtime* have meanings explained in *stat*(S). The length of the null-terminated pathname *h_name*, including the null byte, is given by *h_namesize*.

The last record of the *archive* always contains the name TRAILER!!!. Special files, directories, and the trailer are recorded with *h_filesize* equal to zero.

See Also

cpio(C), find(C), stat(S)

Name

dump – Incremental dump tape format.

Description

The *dump* and *restor* commands are used to write and read incremental dump magnetic tapes.

The dump tape consists of a header record, some bit mask records, a group of records describing file system directories, a group of records describing file system files, and some records describing a second bit mask.

The header record and the first record of each description have the format described by the structure included by:

#include <dumprestor.h>

Fields in the *dumprestor* structure are described below.

NTREC is the number of 512 byte blocks in a physical tape record. MLEN is the number of bits in a bit map word. MSIZ is the number of bit map words.

The TS_ entries are used in the *c_type* field to indicate what sort of header this is. The types and their meanings are as follows:

TS_TYPE	Tape volume label.
TS_INODE	A file or directory follows. The *c_dinode* field is a copy of the disk inode and contains bits telling what sort of file this is.
TS_BITS	A bit mask follows. This bit mask has a one bit for each inode that was dumped.
TS_ADDR	A subblock to a file (*TS_INODE*). See the description of *c_count* below.
TS_END	End of tape record.
TS_CLRI	A bit mask follows. This bit mask contains a one bit for all inodes that were empty on the file system when dumped.
MAGIC	All header blocks have this number in *c_magic*.
CHECKSUM	Header blocks checksum to this value.

Name

file system – Format of a system volume.

Syntax

```
#include <sys/filsys.h>
#include <sys/types.h>
#include <sys/param.h>
```

Description

Every file system storage volume (e.g., a hard disk) has a common format for certain vital information. Every such volume is divided into a certain number of 256 word (512 byte) blocks. Block 0 is unused and is available to contain a bootstrap program or other information.

Block 1 is the *super-block*. The format of a super-block is described in /usr/include/sys/filesys.h. In that include file, *S_isize* is the address of the first data block after the i-list. The i-list starts just after the super-block in block 2; thus the i-list is *s_isize*– 2 blocks long. *S_fsize* is the first block not potentially available for allocation to a file. These numbers are used by the system to check for bad block numbers. If an "impossible" block number is allocated from the free list or is freed, a diagnostic is written on the console. Moreover, the free array is cleared so as to prevent further allocation from a presumably corrupted free list.

The free list for each volume is maintained as follows. The *s_free* array contains, in *s_free*[1], ..., *s_free*[*s_nfree*– 1], up to 49 numbers of free blocks. *S_free*[0] is the block number of the head of a chain of blocks constituting the free list. The first long in each free-chain block is the number (up to 50) of free-block numbers listed in the next 50 longs of this chain member. The first of these 50 blocks is the link to the next member of the chain. To allocate a block: decrement *s_nfree*, and the new block is *s_free*[*s_nfree*]. If the new block number is 0, there are no blocks left, so give an error. If *s_nfree* becomes 0, read in the block named by the new block number, replace *s_nfree* by its first word, and copy the block numbers in the next 50 longs into the *s_free* array. To free a block, check if *s_nfree* is 50; if so, copy *s_nfree* and the *s_free* array into it, write it out, and set *s_nfree* to 0. In any event set *s_free*[*s_nfree*] to the freed block's number and increment *s_nfree*.

S_tfree is the total free blocks available in the file system.

S_ninode is the number of free i-numbers in the *s_inode* array. To allocate an inode: if *s_ninode* is greater than 0, decrement it and return *s_inode*[*s_ninode*]. If it was 0, read the i-list and place the numbers of all free inodes (up to 100) into the *s_inode* array, then

Name

inode - Format of an inode.

Syntax

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

Description

An inode for a plain file or directory in a file system has the structure defined by <sys/ino.h>. For the meaning of the defined types *off_t* and *time_t* see *types*(F).

Files

/usr/include/sys/ino.h

See Also

stat(S), filesystem(F), types(F)

Part 2 contains lines with 11 fields each. Each field is a maximum of 8 characters delimited by a blank if less than 8:

Field 1:
Device associated with this line
Field 2:
open routine
Field 3:
close routine
Field 4:
read routine
Field 5:
write routine
Field 6:
ioctl routine
Field 7:
receiver interupt routine
Field 8:
unused- should be nulldev
Field 9:
unused- should be nulldev
Field 10:
output start routine
Field 11:
unused- should be nulldev

Part 3 contains lines with 2 fields each:

Field 1:	alias name of device (8 chars. maximum).
Field 2:	reference name of device (8 chars. maximum; specified in part 1).

Part 4 contains lines with 2 or 3 fields each:

Field 1:	parameter name (as it appears in description file; 20 chars. maximum)
Field 2:	parameter name (as it appears in the c.c file; 20 chars. maximum)
Field 3:	default parameter value (20 chars. maximum; parameter specification is required if this field is omitted)

Devices that are not interrupt-driven have an interrupt vector size of zero. Devices which generate interupts but are not of the standard character or block device mold, should be specified with a type (field 4 in part 1) which has neither the block nor char bits set.

See Also

config(CP)

Name

sccsfile - Format of an SCCS file.

Description

An SCCS file is an ASCII file. It consists of six logical parts: the *checksum*, the *delta table* (contains information about each delta), *user names* (contains login names and/or numerical group IDs of users who may add deltas), *flags* (contains definitions of internal keywords), *comments* (contains arbitrary descriptive information about the file), and the *body* (contains the actual text lines intermixed with control lines). Each logical part of an SCCS file is described in detail below.

Throughout an SCCS file there are lines which begin with the ASCII SOH (start of heading) character (octal 001). This character is hereafter referred to as *the control character* and will be represented graphically as @. Any line described below which is not depicted as beginning with the control character is prevented from beginning with the control character. Entries of the form DDDDD represent a five digit string (a number between 00000 and 99999).

Checksum

The checksum is the first line of an SCCS file. The form of the line is:

@hDDDDD

The value of the checksum is the sum of all characters, except those of the first line. The @hR provides a *magic number* of (octal) 064001.

"Delta Table"

The delta table consists of a variable number of entries of the form:

```
@s DDDDD/DDDDD/DDDDD
@d <type> <SCCS ID> yr/mo/da hr:mi:se <pgmr> DDDDD DDDDD
@i DDDDD ...
@x DDDDD ...
@g DDDDD ...
@m <MR number>
  .
  .
@c <comments> ...
  .
  .
@e
```

validity checking program. The i flag controls the warning/error aspect of the "No id keywords" message. When the i flag is not present, this message is only a warning; when the i flag is present, this message will cause a "fatal" error (the file will not be gotten, or the delta will not be made). When the b flag is present the – b option may be used with the *get* command to cause a branch in the delta tree. The m flag defines the first choice for the replacement text of the sccsfile.F identification keyword. The f flag defines the "floor" release; the release below which no deltas may be added. The c flag defines the "ceiling" release; the release above which no deltas may be added. The d flag defines the default SID to be used when none is specified on a *get* command. The n flag causes *delta* to insert a "null" delta (a delta that applies *no* changes) in those releases that are skipped when a delta is made in a *new* release (e.g., when delta 5.1 is made after delta 2.7, releases 3 and 4 are skipped). The absence of the n flag causes skipped releases to be completely empty. The j flag causes *get* to allow concurrent edits of the same base SID. The l flag defines a *list* of releases that are *locked* against editing (*get*(CP) with the – e option). The q flag defines the replacement for the identification keyword.

Comments

Arbitrary text surrounded by the bracketing lines @ t and @ T. The comments section typically contains a description of the file's purpose.

Body

The body consists of text lines and control lines. Text lines don't begin with the control character, control lines do. There are three kinds of control lines: *insert*, *delete*, and *end*, as follows:

```
@I DDDDD
@D DDDDD
@E DDDDD
```

The digit string (DDDDD) is the serial number corresponding to the delta for the control line.

See Also

admin(CP), delta(CP), get(CP), prs(CP)

Xenix *Programmer's Guide*

Name

x.out − loader output

Synopsis

#include [a.out.h]

Description

x.out is the output file of the loader **ld(CP)**. **ld(CP)** makes **x.out** executable if there are no errors and no unresolved external references. The following layout information is given in the include file for the 68000:

```
struct  xexec {                 /* x.out header */
  unsigned short x_magic;    /* magic number */
  unsigned short x_ext;      /* size of header extension */
  long      x_text;          /* size of text segment */
  long      x_data;          /* size of initialized data */
  long      x_bss;           /* size of uninitialized data */
  long      x_syms;          /* size of symbol table */
  long      x_reloc;         /* relocation table length */
  long      x_entry;         /* entry point */
  char      x_cpu;           /* cpu type & byte/word order */
  char      x_relsym;        /* relocation & symbol format */
  unsigned short x_renv;     /* run-time environment */
};
struct   xext {         /* x.out header extension */
  long  xe_trsize;    /* size of text relocation */
  long  xe_drsize;    /* size of data relocation */
  long  xe_tbase;     /* text relocation base */
  long  xe_dbase;     /* data relocation base */
  long  xe_stksize;   /* stack size (if XE_FS set) */
};
```

The file has four sections: a **header**, the program's **text and data**, **relocation information**, and a **symbol table**, in that order. The header optionally has a header extension as shown above. The relocation and symbol section will be empty if the program was loaded with the **−s** option of **ld**, or if the symbols and relocation have been removed by **strip(CP)**. The sizes of each section in the header are given as longs, but have even alignment. The size of the header is not included in any of the other sizes. When an **x.out** file is loaded into core for execution, three logical segments are set up: the **text segment**, the **data segment** (with uninitialized data, which starts off as all 0, following initialized data), and a **stack**. The text segment begins at 0 in the core image; the header