SCO

THE SANTA CRUZ OPERATION

SCO

THE SANTA CRUZ OPERATION

For the Apple™ Lisa™ 2

Product

XENIX™ Development System
Introductory Softcare™ Support

XENIX is a fully licensed version of the UNIX™ Operating System as authorized by AT&T for single and multiuser operation.

WARNING: the XENIX Development System is licensed for use only with the XENIX Operating System. Any other use of this product is in violation of applicable license agreements.

Version

XENIX 3.0, Release 1.0

Requirements

System	XENIX Operating System **Apple Lisa 2**
Memory	512 KBytes or greater
Hard Disk	Apple hard disk(s) totalling 10Mb or greater

Please Note

Additional requirements information may be obtained by calling SCO Softcare™ Center 9:00 a.m. to 5:00 p.m. Pacific Time, Monday through Friday.

Read carefully the enclosed License Agreement before opening the diskette package. Opening the diskette package indicates your acceptance of this License Agreement. If you do not agree with it, you have ten (10) days from date of purchase to return this package (with the diskette package UNOPENED) and a copy of your purchase receipt, and your money will be refunded.

The XENIX™ Development System

Programmer's Reference for the Apple Lisa 2™

The Santa Cruz Operation, Inc.

The Santa Cruz Operation, Inc.
500 Chestnut Street
P.O. Box 1900
Santa Cruz, California 95061
(408) 425-7222 • TWX: 910-598-4510 SCO SACZ

Release: 68-5-24-84-1.0/1.0

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 5 describes the process control functions. These functions let a program execute other programs and create multiple copies of itself.

Chapter 6 describes the pipe functions. These functions let programs communicate with one another without resorting to the creation of temporary files.

Chapter 7 describes the signal functions. These functions let a program process signals that are normally processed by the system.

Chapter 8 describes system resource functions. These functions let a program dynamically allocate memory, share memory with other programs, lock files against access by other programs, and use semaphores.

Chapter 9 describes the error processing functions. These functions let a program process errors encountered while accessing the file system or allocating memory.

Appendix A describes the assembly language interface with C programs and explains the calling and return value conventions of C functions.

Appendix B explains how to create and use new XENIX system calls.

This manual assumes that you understand the C programming language and that you are familiar with the XENIX shell, *sh*. Nearly all programming examples in this guide are written in C, and all examples showing a shell use the *sh* shell.

1.4 Notational Conventions

This manual uses a number of special symbols to describe the form of the library function calls. The following is a list of these symbols and their meaning.

[ ]	Brackets indicate an optional function argument.
...	Ellipses indicate that the preceding argument may be repeated one or more times.
SMALL	Small capitals indicate manifest constants. These system-dependent constants and are defined in a variety of *include* files.
italics	Italic characters indicate placeholders for function arguments. These must be replaced with appropriate values or names of variables.

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.

The following is a list of the special names:

stdin	The name of the standard input file.
stdout	The name of the standard output file.
stderr	The name of the standard error file.
EOF	The value returned by the read routines on end-of-file or error.
NULL	The null pointer, returned by pointer-valued functions, to indicate an error.
FILE	The name of the file type used to declare pointers to streams.
BSIZE	The size in bytes (usually 1024) suitable for an I/O buffer supplied by the user.

2.1.3 Special Macros

The functions *getc*, *getchar*, *putc*, *putchar*, *feof*, *ferror*, and *fileno* are actually macros, not functions. This means that you cannot redeclare them or use them as targets for a breakpoint when debugging.

2.2 Using Command Line Arguments

The XENIX system lets you pass information to a program at the same time you invoke it for execution. You can do this with command line arguments.

A XENIX command line is the line you type to invoke a program. A command line argument is anything you type in a XENIX command line. A command line argument can be a filename, an option, or a number. The first argument in any command line must be the filename of the program you wish to execute.

When you type a command line, the system reads the first argument and loads the corresponding program. It also counts the other arguments, stores them in memory in the same order in which they appear on the line, and passes the count and the locations to the main function of the program. The function can then access the arguments by accessing the memory in which they are stored.

To access the arguments, the main function must have two parameters: "argc", an integer variable containing the argument count, and "argv", an array of pointers to the argument values. You can define the parameters by using the lines:

```
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.

2.3 Using the Standard Files

Whenever you invoke a program for execution, the XENIX system automatically creates a standard input, a standard output, and a standard error file to handle a program's input and output needs. Since the bulk of input and output of most programs is through the user's own terminal, the system normally assigns the user's terminal keyboard and screen as the standard input and output, respectively. The standard error file, which receives any error messages generated by the program, is also assigned to the terminal's screen.

A program can read and write to the standard input and output files with the *getchar*, *gets*, *scanf*, *putchar*, *puts*, and *printf* functions. The standard error file can be accessed using the stream functions described in the section "Using Stream I/O" later in this chapter.

The XENIX system lets you redirect the standard input and output using the shell's redirection symbols. This allows a program to use other devices and files as its chief source of input and output in place of the terminal's keyboard and screen.

The following sections explains how to read from and write to the standard input and output. It also explains how to redirect the standard input and output.

2.3.1 Reading From the Standard Input

You can read from the standard input with the *getchar*, *gets*, and *scanf* functions.

The *getchar* function reads one character at a time from the standard input. The function call has the form:

```
c = getchar()
```

where *c* is the variable to receive the character. It must have **int** type. The function normally returns the character read, but will return the end-of-file value EOF if the end of the file or an error is encountered.

The *getchar* function is typically used in a conditional loop to read a string of characters from the standard input. For example, the following function reads "cnt" number of characters from the keyboard.

```
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* ...)

where *format* is a pointer to a string that defines the format of the values to be read and *argptr* is one or more pointers to the variables that will receive the values. 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 values it read from the standard input, but it will return the value EOF if the end of the file or an error is encountered.

Unlike the *getchar* and *gets* functions, *scanf* skips all whitespace characters, reading only those characters which make up a value. It then converts the characters, if necessary, into the appropriate string or number.

The *scanf* function is typically used whenever formatted input is required, i.e., input that must be typed in a special way or which has a special meaning. For example, in the following program fragment *scanf* reads both a name and a number from the same line.

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

scanf("%s %d", name, &number);
```

In this example, the string "%s %d" defines what values are to be read (a string and a decimal number). The string is copied to the character array "name" and the number to the integer variable "number". Note that pointers to these variables are used in the call and not the actual variables themselves.

When reading from the keyboard, *scanf* waits for values to be typed before returning. Each value must be separated from the next by one or more whitespace characters (such as spaces, tabs, or even newline characters). For example, for the function:

```
scanf("%s %d %c", name, age, sex);
```

an acceptable input is:

```
John 27
M
```

If a value is a number, it must have the appropriate digits, that is, a decimal number must have decimal digits, octal numbers octal digits, and hexadecimal numbers hexadecimal digits.

If *scanf* encounters an error, it immediately stops reading the standard input. Before *scanf* can be used again, the illegal character that caused the error must be removed from the input using the *getchar* function.

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.

Since the function automatically appends a newline character, it is typically used when writing full lines to the standard output. For example, the following program fragment writes one of three strings to the standard output.

```
char c;

switch(c) {
        case('1'):
                puts("Continuing...");
                break;
        case('2'):
                puts("All done.");
                break;
        default:
                puts("Sorry, there was an error.");
}
```

The string to be written depends on the value of "c".

The *printf* function writes one or more values to the standard output where a value is a character string or a decimal, octal, or hexadecimal number. The function automatically converts numbers into the proper display format. The function call has the form:

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

where *format* is a pointer to a string which describes the format of each value to be written and *arg* is one or more variables containing the values 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. The function normally returns zero, but will return a nonzero value if an error is encountered.

The *printf* function is typically used when formatted output is required, i.e., when the output must be displayed in a certain way. For example, you may use the function to display a name and number on the same line as in the following example.

```
char name [];
int number;

printf("%s %d", name, number);
```

In this example, the string "%s %d" defines the type of output to be displayed (a string and a number separated by a space). The output values are copied from the character array "name" and the integer variable "number".

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, |.

For example, the command line

```
dial | wc
```

connects the standard output of the program *dial* to the standard input of the program *wc*. (The standard input of *dial* and standard output of *wc* are not affected.) If *dial* writes to its standard output with the *putchar*, *puts*, or *printf* functions, *wc* can read this output with the *getchar* and *scanf* functions.

Note that when the program on the output side of a pipe terminates, the system automatically places the constant value EOF in the standard input of the program on the input side. Pipes are described in more detail in Chapter 6, "Creating and Using Pipes".

2.3.6 Program Example

This section shows how you may use the standard input and output files to perform useful tasks. The *ccstrip* (for "control character strip") program defined below strips out all ASCII control characters from its input except for newline and tab. You may use this program to display text or data files which contain characters that may disrupt your terminal screen.

```
#include <stdio.h>

main()  /* ccstrip: strip nth characters */
{
        int c;
        while ((c = getchar()) != EOF)
                if ((c >= ' ' && c < 0177) ||
                        c == '\t' || c == '\n')
                        putchar(c);
        exit(0);
}
```

You can strip and display the contents of a single file by changing the standard input of the *ccstrip* program to the desired file. The command line

```
ccstrip <doc.t
```

reads the contents of the file *doc.t*, strips out control characters, then writes the stripped file to the standard output.

If you wish to strip several files at the same time, you can create a pipe between the cat command and *ccstrip*.

To read and strip the contents of the files *file1*, *file2*, and *file3*, then display them on the standard output use the command:

```
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;
```

The standard input, output, and error files, like other opened files, have corresponding file pointers. These file pointers are named *stdin* for standard input, *stdout* for standard output, and *stderr* for standard error. Unlike other file pointers, the standard file pointers are predefined in the *stdio.h* file. This means a program may use these pointers to read and write from the standard files without first using the *fopen* function to open them.

The predefined file pointers are typically used when a program needs to alternate between the standard input or output file and an ordinary file. Although the predefined file pointers have FILE type, they are constants, not variables. They must not be assigned values.

2.4.2 Opening a File

The *fopen* function opens a given file and returns a pointer (called a file pointer) to a structure containing the data necessary to access the file. The pointer may then be used in subsequent stream functions to read from or write to the file.

The function call has the form:

fp = fopen(*filename*, *type*)

where *fp* is the pointer to receive the file pointer, *filename* is a pointer to the name of the file to be opened and *type* is a pointer to a string that defines how the file is to be opened. The type string may be "r" for reading, "w" for writing, and "a" for appending, that is, open for writing at the end of the file.

A file may be opened for different operations at the same time if separate file pointers are used. For example, the following program fragment opens the file named */usr/accounts* for both reading and writing.

```
FILE *rp, *wp;

rp = fopen("/usr/accounts","r");
wp = fopen("/usr/accounts","a");
```

Opening an existing file for writing destroys the old contents. Opening an existing file for appending leaves the old contents unchanged and causes any data written to the file to be appended to the end.

Trying to open a nonexistent file for reading causes an error. Trying to open a nonexistent file for writing or appending causes a new file to be created. Trying to open any file for which the program does not have appropriate permission causes an error.

The function normally returns a valid file pointer, but will return the value NULL if an error opening the file is encountered. It is wise to check for the NULL value after each call to the function to prevent reading or writing after an error.

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 up to 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*.

The function is typically used to read a full line from a file. For example, the following program fragment reads a string of characters from the file given by "myfile".

```
char cmdln[MAX];
FILE *myfile;

if ( fgets( cmdln, MAX, myfile ) != NULL)
        parse( cmdln );
```

In this example, *fgets* copies the string to the character array "cmdln".

2.4.5 Reading Records from a File

The *fread* function reads one or more records from a file and copies them to a given memory location. The function call has the form:

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

where *ptr* is a pointer to the location to receive the records, *size* is the size (in bytes) of each record to be read, *nitems* is the number of records to be read, and *stream* is the file pointer of the file to be read. The *ptr* may be a pointer to a variable of any type (from a single character to a structrure). The *size*, an integer, should give the numbers of bytes in each item you wish to read. One way to ensure this is to use the *sizeof* function on the pointer *ptr* (see the example below). The function always returns the number of records it read, regardless of whether or not the end of the file or an error is encountered.

The function is typically used to read binary data from a file. For example, the following program fragment reads two records from the file given by "database" and copies the records into the structure "person".

```
FILE *database;
struct record {
        char name[20];
        int  age;
} person;

fread(&person, sizeof(person), 2, database);
```

Note that since *fread* does not explicitly indicate errors, the *feof* and *ferror* functions should be used to detect end of the file and errors. These functions are described later in this chapter.

2.4.6 Reading Formatted Data From a File

The *fscanf* function reads formatted input from a given file and copies it to the memory location given by the respective argument pointers, just as the *scanf* 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".

```
FILE *out;
char name[MAX];
int i;

for (i=0; i<MAX; i++)
        fputc( name[i], out);
```

The only difference between the *putc* and *fputc* functions is that *putc* is defined as a macro and *fputc* as an actual function. This means that *fputc*, unlike *putc*, may be used as an argument to another function, as the target of a breakpoint when debugging, and to avoid the side effects of macro processing.

2.4.8 Writing a String to a File

The *fputs* function writes a string to a given file. The function call has the form:

fputs(*s*, *stream*)

where *s* is a pointer to the string to be written and *stream* is the file pointer to the file.

The function is typically used to copy strings from one file to another. For example, in the following program fragment, *gets* and *fputs* are combined to copy strings from the standard input to the file given by "out".

```
FILE *out;
char cmdln[MAX];

if ( gets( cmdln ) != EOF )
        fputs( cmdln, out);
```

The function normally returns zero, but will return EOF if an error is encountered.

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".

```
FILE *database;
struct record {
        char name[20];
        int age;
} person;

fwrite(&person, sizeof(person), 2, database);
```

The records are copied from the structure "person".

Since the function does not report the end of the file or errors, the *feof* and *ferror* functions should be used to detect these conditions.

2.4.11 Testing for the End of a File

The *feof* function returns the value -1 if a given file has reached its end. The function call has the form:

feof (*stream*)

where *stream* is the file pointer of the file. The function returns -1 only if the file has reached its end, otherwise it returns 0. The return value is always an integer.

The *feof* function is typically used after those functions whose return value is not a clear indicator of an end-of-file condition. For example, in the following program fragment the function checks for the end of the file after each character is read. The reading stops as soon as *feof* returns -1.

```
char name[10];
FILE *stream;

do
        fread( name, size(name), 1, stream );
while(!feof( stream ));
```

2.4.12 Testing For File Errors

The *ferror* function tests a given stream file for an error. The function call has the form:

ferror (*stream*)

where *stream* is the file pointer of the file to be tested. The function returns a nonzero (true) value if an error is detected, otherwise it returns zero (false). The function returns an integer value.

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 *fclose* functions to open, close, read, and write to the given files. The program incorporates a basic design that is common to other XENIX programs, namely it uses the filenames found in the command line as the files to open and read, or if no names are present, it uses the standard input. This allows the program to be invoked on its own, or be the receiving end of a pipe.

```
#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 program writes an error message to the standard error file "stderr" with the *fprintf* function. The function prints the format string "wc: can't open %s", replacing the "%s" with the name pointed to by "argv[i]".

Once a file is opened, the program uses the *getc* function to read each character from the file. As it reads characters, the program keeps a count of the number of characters, words, and lines. The program continues to read until the end of the file is encountered, that is, when *getc* returns the value EOF.

Once a file has reached its end, the program uses the *printf* function to display the character, word, and line counts at the standard output. The format string in this function causes the counts to be displayed as long decimal numbers with no more than 7 digits. The program then closes the current file with the *fclose* function and examines the command line arguments to see if there is another filename.

When all files have been counted, the program uses the *printf* function to display a grand total at the standard output, then stops execution with the *exit* function.

2.5 Using More Stream Functions

The stream functions allow more control over a file than just opening, reading, writing, and closing. The functions also let a program take an existing file pointer and reassign it to another file (similar to redirecting the standard input and output files) as well as manipulate the buffer that is used for intermediate storage between the file and the program.

2.5.1 Using Buffered Input and Output

Buffered I/O is an input and output technique used by the XENIX system to cut down the time needed to read from and write to files. Buffered I/O lets the system collect the characters to be read or written and then transfer them all at once rather than one character at a time. This reduces the number of times the system must access the I/O devices and consequently provides more time for running user programs. Not all files have buffers. For example, files associated with terminals, such as the standard input and output, are not buffered. This prevents unwanted delays when transferring the input and output. When a file does have a buffer, the buffer size in bytes is given by the mainfest constant BSIZE, which is defined in the *stdio.h* file.

When a file has a buffer, the stream functions read from and write to the buffer instead of the file. The system keeps track of the buffer and when necessary fills it with new characters (when reading) or flushes (copies) it to the file (when writing). Normally, a buffer is not directly accessible to a program, however a program can define its own buffer for a file with the *setbuf* function. The function also lets a program change a buffered file to be an unbuffered one. The *ungetc* function lets a program put a character it has read back into the buffer, 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".

```
char *p;

p=malloc( BSIZE );
setbuf ( stdout, p );
```

The new buffer is BSIZE bytes long.

The function may also be used to change a file from buffered to unbuffered input or output. Unbuffered input and output generally increase the total time needed to transfer large numbers of characters to or from a file, but give the fastest transfer speed for individual characters.

The *setbuf* function should be called immediately after opening a file and before reading or writing to it. Furthermore, the *fclose* or *fflush* function must be used to flush the buffer before terminating the program. If not used, some data written to the buffer may not be written to the file.

2.5.4 Putting a Character Back into a Buffer

The *ungetc* function puts a character back into the buffer of a given file. The function call has the form:

ungetc (*c*, *stream*)

where *c* is the character to put back and *stream* is the file pointer of the file. The function normally returns the same character it put back, but will return the value EOF if an error is encountered.

The function is typically used when scanning a file for the first character of a string of characters. For example, the following program fragment puts the first character that is not a whitespace character back into the buffer of the file given by "infile", allowing the subsequent call to *gets* to read that character as the first character in the string.

```
FILE *infile
char name[20];

while( isspace( c=getc(infile) ) )
        ;
ungetc( c, stdin );
gets( name, stdin );
```

Putting a character back into the buffer does not change the corresponding file; it only changes the next character to be read.

Note that the function can put a character back only if one has been previously read. The function cannot put more than one character back at a time. This means if three characters are read, then only the last character can be put back, never the first two.

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 "errfiag" 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.

Once a file is opened for reading, a program can read bytes from it with the ***read*** function. A program can write to a file opened for writing or appending with the ***write*** function. A program can close a file with the ***close*** function.

2.6.1 Using File Descriptors

Each file that has been opened for access by the low-level functions has a unique integer called a "file descriptor" associated with it. A file descriptor is similar to a file pointer in that it identifies the file. A file descriptor is unlike a file pointer in that it does not point to any specific structure. Instead the descriptor is used internally by the system to access the necessary information. Since the system maintains all information about a file, the only access to a file for a program is through the file descriptor.

There are three predefined file descriptors (just as there are three predefined file pointers) for the standard input, output, and error files. The descriptors are 0 for the standard input, 1 for the standard output, and 2 for the standard error file. As with predefined file pointers, a program may use the predefined file descriptors without explicitly opening the associated files.

Note that if the standard input and output files are redirected, the system changes the default assignments for the file descriptors 0 and 1 to the named files. This is also true if the input or output is associated with a pipe. File descriptor 2 normally remains attached to the terminal.

2.6.2 Opening a File

The ***open*** function opens an existing or a new file and returns a file descriptor for that file. The function call has the form:

fd = open(***name***, ***access*** [,***mode***]);

where ***fd*** is the integer variable to receive the file descriptor, ***name*** is a pointer to a string containing the filename, ***access*** is an integer expression giving the type of file access, and ***mode*** is an integer number giving a new file's permissions. The function normally returns a file descriptor (a positive integer), but will return -1 if an error is encountered.

The ***access*** expression is formed by using one or more of the following manifest constants: O_RDONLY for reading, O_WRONLY for writing, O_RDWR for both reading and writing, O_APPEND for appending to the end of an existing file, and O_CREAT for creating a new file. (Other constants are described in *open*(S) in the XENIX *Reference Manual*.) The logical OR operator (|) may be used to combine the constants. The ***mode*** is required only if O_CREAT is given. For example, in the following program fragment, the function is used to open the existing file named ***/usr/accounts*** for reading and open the new file named ***/usr/tmp/scratch*** for reading and writing.

```
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 requested to be written.

The number of bytes to be written is arbitrary. The two most common values are 1, which means one character at a time and 512, which corresponds to the physical block size on many peripheral devices.

2.6.5 Closing a File

The *close* function breaks the connection between a file descriptor and an open file, and frees the file descriptor for use with some other file. The function call has the form:

close (*fd*)

where *fd* is the file descriptor of the file to close. The function normally returns 0, but will return -1 if an error is encountered.

The function is typically used to close files that are not longer needed. For example, the following program fragment closes the standard input if the argument count is greater than 1.

```
int fd;

if (argc >1)
        close( 0 );
```

Note that all open files in a program are closed when a program terminates normally or when the *exit* function is called, so no explicit call to *close* is required.

2.6.6 Program Examples

This section shows how to use the low-level functions to perform useful tasks. It presents three examples that incorporate the functions as the sole method of input and output.

The first program copies its standard input to its standard output.

```
#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:

```
#define CMASK 0377       /* for making char's > 0 */
#define BUFSIZE          BSIZE

getchar()/* buffered version */
{
        static char      buf[BUFSIZE];
        static char      *bufp = buf;
        static intn = 0;

        if (n == 0) {    /* buffer is empty */
                n = read(0, buf, BUFSIZE);
                bufp = buf;
        }
        return((--n >= 0) ? *bufp++ & CMASK : EOF);
}
```

Again, each character must be masked with the octal constant0377.

The final example is a simplified version of the XENIX utility, *cp*, a program that copies one file to another. The main simplification is that this version copies only one file, and does not permit the second argument to be a directory.

```
#define NULL 0
#define BUFSIZE BSIZE
#define PMODE 0644 /* RW for owner, R for group, others */

main(argc, argv)  /* cp: copy f1 to f2 */
int argc;
char *argv[];
{
        int     f1, f2, n;
        char    buf[ BUFSIZE ];

        if (argc != 3)
                error("Usage: cp from to", NULL);
        if ((f1 = open(argv[1], O_RDONLY)) == -1)
                error("cp: can't open %s", argv[1]);
        if ((f2 = open(argv[2], O_CREAT | O_WRONLY,
                        PMODE)) == -1)
                error("cp: can't create %s", argv[2]);

        while ((n = read(f1, buf, BUFSIZE)) > 0)
                if (write(f2, buf, n) != n)
                        error("cp: write error", NULL);
        exit(0);
}
```

```
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 to move the character pointer to the end of a file to allow appending, or to the beginning as in a rewind function. For example, the call

```
lseek(fd, (long)0, 2);
```

prepares the file for appending, and

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

rewinds the file (moves the character pointer to the beginning). Notice the "(long)0" argument; it could also be written as

```
0L
```

Using *lseek* it is possible to treat files more or less like large arrays, at the price of slower access. For example, the following simple function reads any number of bytes from any arbitrary place in a file:

```
get(fd, pos, buf, n) /* read n bytes from position pos */
int fd, n;
long pos;
char *buf;
{
        lseek(fd, pos, 0);  /* get to pos */
        return(read(fd, buf, n));
}
```

2.6.9 Moving the Character Pointer in a Stream

The *fseek* function, a stream function, moves the character pointer in a file to a given location. The function call has the form:

fseek (*stream*, *offset*, *ptrname*)

where *stream* is the file pointer of the file, *offset* is the number of characters to move to the new position (it must be a long integer), and *ptrname* is the starting position in the file of the move (it must be "0" for beginning, "1", for current position, or "2" for end of the file). The function normally returns zero, but will return the value EOF if an error is encountered.

For example, the following program fragment moves the character pointer to the end of the file given by "stream".

```
FILE *stream;

fseek(stream, (long)0, 2);
```

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 *refresh* or *wrefresh*, a program can maintain several different windows, each containing different characters for the same portion of the terminal screen. The program can choose which window should actually be displayed before updating.

A program can continue to add new characters to a screen or window as needed, and edit these characters by using functions such as *insertln*, *deleteln*, and *clear*. A program can also combine windows to make a composite screen using the *overlay* and *overwrite* functions. In each case, the *refresh* or *wrefresh* function is used to copy the changes to the terminal screen.

3.1.2 Using the Library

To use the *curses* library in a program, you must add the line

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

to the beginning of your program. The *curses.h* file contains definitions for types and variables used by the library.

The actual screen processing functions are in the library files *libcurses.a* and *libtermcap.a*. These files are not automatically read when you compile your program, so you must include the appropriate library switches in your invocation of the compiler. The command line must have the form:

cc *file* ... -lcurses -ltermcap

where *file* is the name of the source file you wish to compile. You may given more than one filename if desired. You may also use other compiler options in the command line. For example, the command

```
cc main.c intf.c -lcurses -ltermcap -o sample
```

compiles the files *main.c* and *intf.c*, and copies the executable program to the file *sample* after linking the screen processing library files to the program.

Note that the *curses.h* file automatically includes the file *sgtty.h* in your program. This file must not be included twice.

The screen processing library has a variety of predefined names. These names refer to variables, manifest constants, and types that can be used with the library functions. The following is a list of these names.

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.

Types and Constants

Name	Description
reg	A storage class. It is the same as register storage class.
bool	A type. It is the same a char type.
TRUE	The boolean true value (1).
FALSE	The boolean false value (0).

3.2 Preparing the Screen

The *initscr* and *endwin* functions perform the operations required to initialize and terminate programs that use the screen processing functions. The following sections describe these functions and how they affect the terminal.

3.2.1 Initializing the Screen

The *initscr* function initializes screen processing for a program by allocating the required memory space for the screen processing functions and variables, and by setting the terminal to the proper modes. The function call has the form:

```
initscr()
```

No arguments are required.

The *initscr* function must be used to prepare the program for subsequent calls to other screen processing functions and for use of the screen processing variables. For example, in the following program fragment *initscr* initializes the screening processing functions.

```
#include <curses.h>

main ()
{

initscr();
if ( cmpstr(ttytype,"dumb") )
        fprintf(stderr, "Terminal type can't display screen.");
```

In this example, the predefined variable "ttytype" is checked for the current terminal type.

The function returns (WINDOW*) ERR if memory allocation causes an overflow.

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.*

Note

> The terminal mode functions should only be used in conjunction with other screen processing functions. They should not be used alone.

3.2.4 Using Default Window Flags

The *initscr* function automatically clears the **cursor**, **scroll**, and **clear** flags of the standard screen to their default values. These flags, called the window flags, define how the *refresh* function affects the terminal screen when refreshing from the standard screen. When clear, the **cursor** flag prevents the terminal's cursor from moving back to its original location after the screen is updated, the **scroll** flag prevents scrolling on the screen, and the **clear** flag prevents the characters on the screen from being cleared before being updated. The flags may be changed by using the functions described in the section "Setting Window Flags," in this chapter.

3.2.5 Using the Default Terminal Size

The *initscr* function sets the terminal screen size to a default number of lines and columns. The default values are given in the predefined variables "LINES" and "COLS". You can change the default size of a terminal by setting the variables to new values. This should be done before the first call to *initscr*. If it is done after the first call, a second call to *initscr* must be made to delete the existing standard screen and create a new one.

3.2.6 Terminating Screen Processing

The *endwin* function terminates the screen processing in a program by freeing all memory resources allocated by the screen processing functions and restoring the terminal to the state before screen processing began. The function call has the form:

```
endwin()
```

No arguments are required.

The *endwin* function must be used before leaving a program that has called the *initscr* function to restore the terminal to its previous state. The function is generally the last function call in the program. For example, in the following program fragment *initscr* and *endwin* form the beginning and end of the program.

```
#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.

3.3.2 Adding a String

The *addstr* function adds a string of characters to the standard screen, placing the first character of the string at the current position and moving the pointer one position to the right for each character in the string. The function call has the form:

addstr(*str*)

where *str* is a character pointer to the given string. For example, if the current position is (0,0), the function call

```
addstr("line");
```

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

If the string contains newline, return, or tab characters, the function performs the same actions as described for the *addch* function. If the string does not fit on the current line, the string is truncated.

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

3.3.3 Printing Strings, Characters, and Numbers

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

printw(*fmt* [, *arg*] ...)

where *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 argument 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, if the current position is (0,0), the function call

```
printw("%s %d", name, 15);
```

prints the name given by the variable "name" starting at position (0,0). It then 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 keyboard and stores it in the array "name".

```
char name[20];

getstr(name);
```

If the terminal is set to ECHO mode, *getstr* copies the string to the standard screen. 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. Terminal modes are described later in the chapter.

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

3.3.6 Reading Strings, Characters, and Numbers

The *scanw* function reads one or more values from the terminal keyboard and copies the values to given locations. A value may be a string, character, or decimal, octal, or hexadecimal number. The function call has the form:

scanw(*fmt*, *argptr* ...)

where *fmt* is a pointer to a string defining the format of the values to be read, and *argptr* is a pointer to the variable to receive a value. If more than one *argptr* is given, each must be separated from the preceding item with a comma (,). For each *argptr* 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 *scanf*(S) in the XENIX *Reference Manual*.)

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

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

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

In this example, the input values are stored in the character array "name" and the integer variable "id".

If the terminal is set to ECHO mode, the function copies the string to the standard screen. 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 function returns ERR if it encounters an error such as illegal scrolling.

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 ff the bottom of the screen. The function call has the form:

```
insertln()
```

No arguments are required.

The function is used to insert additional lines of text in the standard screen. For example, in the following program fragment *insertln* is used to insert a blank line when the count in "cnt" is equal to 79.

```
int cnt;

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

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

3.3.10 Deleting a Character

The *delch* function deletes the character at the current position and shifts the character to the right of the deleted character (and all characters to its right) one position to the left. The last character on the line is replaced by a space. The function call has the form:

```
delch()
```

No arguments are required.

The function is typically used to delete a series of characters from the standard screen. For example, in the following program fragment *delch* deletes the character at the current position as long as the count in "cnt" is not 0.

```
int cnt;

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

3.3.11 Deleting a Line

The *deleteln* 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 on the screen blank. The function call has the form:

```
deleteln()
```

No arguments are required.

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();
```

clears all characters from (10,10) to (10,79). The characters at the beginning of the line remain unchanged.

Note that both the *clrtobot* and *clrtoeol* functions do not change the current position.

3.3.14 Refreshing From the Standard Screen

The *refresh* function updates the terminal screen by copying one or more characters from the standard screen to the terminal. The function effectively changes the terminal screen to reflect the new contents of the standard screen. The function call has the form:

```
refresh()
```

No arguments are required.

The function is used solely to display changes to the standard screen. The function copies only those characters that have changed since the last call to *refresh* and leaves any existing text on the terminal screen. For example, in the following program fragment *refresh* is called twice.

```
addstr("The first time.\n");
refresh();
addstr("The second time.\n");
refresh();
```

In this example, the first call to *refresh* copies the string "The first time." to the terminal screen. The second call copies only the string "The second time." to the terminal, since the original string has not been changed.

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 Creating and Using Windows

The following sections explain how to create and use windows to display and edit text on the terminal screen.

3.4.1 Creating a Window

The *newwin* function creates a window and returns a pointer that may be used in subsequent screen processing functions. The function call has the form:

win = newwin(*lines*, *cols*, *begin_y*, *begin_x*) 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:

swin = subwin(*win*, *lines*, *cols*, *begin_y*, *begin_x*)

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

The function is typically used to divide a large window into separate regions. For example, in the following program fragment *subwin* creates the subwindow named "cmdmenu" in the lower partof thestandardscreen.

```
WINDOW *cmdmenu;

cmdmenu = subwin(stdscr, 5, 80, 19, 0);
```

In thisexample, changes to "cmdmenu" affect the standard screen as well.

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

3.4.3 Adding and Printing to a Window

The *waddch*, *waddstr*, and *wprintw*functions add and print characters, strings, and numbers to a given window.

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

waddch(*win*, *ch*)

where *win* is a pointer to the window to receive the character, and *ch* gives the character to be added; *ch* must have **char** type. For example, if the current position in the window "midscreen" is (0,0), the function call

```
waddch(midscreen, 'A')
```

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

The *waddstr* function adds a string of characters to the given window, placing the first character of the string at the current position and moving the pointer one position to the right for each character in the string. The function call has the form:

waddstr(*win*, *str*)

where *win* is a pointer to the window to receive the string, and *str*is a character 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*)

where *win* is a pointer to a window, and *c* is the character variable to receive the character.

The function is typically used to read a series of characters from the keyboard. For example, in the following program fragment *wgetch* reads characters until a colon (:) is found.

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

i = 0;
while ((c=wgetch(cmdmenu)) != ':' && i <MAX)
        dir[i++] = c;
```

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

wgetstr(*win*, *str*)

where *win* is a pointer to a window, and *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 for storing the typed string.

The function is typically used to read names and other text from the keyboard. For example, in the following program fragment *wgetstr* reads a string from the keyboard and stores it in the array "filename".

```
char filename[20];

wgetstr(cmdmenu, filename);
```

The *wscanw* function reads one or more values from the standard input file and copies the values to given locations. A value may be a string, a character, or a decimal, octal, or hexadecimal number. The function call has the form:

wscanw(*win*, *fmt* [, *argptr*] ...)

where *win* is a pointer to a window, *fmt* is a pointer to a string defining the format of the values to be read, and *argptr* is a pointer to the variable to receive a value. If more than one *argptr* is given, each must be separated from the preceding by a comma (,). For each *argptr* 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 *scanf*(S) in the XENIX *Reference Manual*.)

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 function is typically used to edit the contents of the given window. For example, the function call

```
winsch(midscreen, 'X');
```

inserts the character "X" at the current position in the window "midscreen".

The *winsertln* 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:

winsertln(*win*)

where *win* is a pointer to the window to receive the blank line.

The function is used to insert lines into a window. For example, in the following program fragment *winsertln* inserts a blank line at the top of the window "cmdmenu" preparing it for a new line.

```
char line[80];

wmove(cmdmenu, 3, 0);
winsertln(cmdmenu);
waddstr(cmdmenu, line);
```

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

3.4.7 Deleting Characters and Lines

The *wdelch* and *wdeleteln* functions delete characters and lines from the given window.

The *wdelch* function deletes the character at the current position and shifts the character to the right of the deleted character (and all characters to its right) one position to the left. The last character on the line is replaced with a space. The function call has the form:

wdelch(*win*)

where *win* is a pointer to a window.

The function is typically used to edit the contents of the standard screen. For example, the function call

```
wdelch(midscreen);
```

deletes the character at the current position in the window "midscreen".

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 position in the window "midscreen" is (10,0), the function call

```
wclrtobot( midscreen );
```

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

The *wclrtoeol* function clears the standard screen from the current position to the end of the current line. The function call has the form:

wclrtoeol(*win*)

where *win* is a pointer to the window to be cleared. For example, if the current position in "midscreen" is (10, 10), the function call

```
wclrtoeol( midscreen );
```

clears all characters from (10,10) to the end of the line. The characters at the beginning of the line remain unchanged.

Note that the *wclrtobot* and *wclrtoeol* functions do not change the current position.

3.4.9 Refreshing From a Window

The *wrefresh* function updates the terminal screen by copying one or more characters from the given window to the terminal. The function effectively changes the terminal screen to reflect the new contents of the window. The function call has the form:

wrefresh(*win*)

where *win* is a pointer to a window.

The function is used solely to display changes to the window. The function copies only those characters that have changed since the last call to *wrefresh* and leaves any existing text on the terminal screen. For example, in the following program fragment *wrefresh* is called twice.

```
waddstr(cmdmenu, "Type a command name\n");
wrefresh(cmdmenu);
waddstr(cmdmenu, "Command: ");
wrefresh(cmdmenu);
```

In this example, the first call to *wrefresh* copies the string "Type a command name" to the terminal screen. The second call copies only the string "Command:" to the terminal, since the original string has not been changed.

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:

```
overwrite( 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. 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 display the contents of a temporary window in the middle of a larger window. For example, in the following program fragment *overwrite* is used to copy the contents of a work window to the standard screen.

```
WINDOW *work;

overwrite(work, stdscr);
refresh();
```

3.4.12 Moving a Window

The *mvwin* function moves a given window to a new position on the terminal screen, causing the upper left corner of the window to occupy a given line and column position. The function call has the form:

```
mvwin( win, y, x)
```

where *win* is a pointer to the window to be moved, *y* is an integer value giving the line to which the corner is to be moved, and *x* is an integer value giving the column to which the corner is to be moved.

The function is typically used to move a temporary window when an existing window under it contains information to be viewed. For example, in the following program fragment *mvwin* moves the window named "work" to the upper left corner of the terminal screen.

```
WINDOW *work;

mvwin(work, 0,0);
```

The function returns ERR if it encounters a error such as an attempt to move part of a window off the edge of the screen.

3.4.13 Reading a Character From a Window

The *inch* and *winch* functions read a single character from the current pointer position in a window or screen.

The *inch* function reads a character from the standard screen. 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 allocated variables. The function call has the form:

```
delwin( win )
```

where *win* is the pointer to the window to be deleted.

The function is typically used to remove temporary windows from a program or to free memory space for other uses. For example, the function call

```
delwin(midscreen);
```

removes the window named "midscreen".

3.5 Using Other Window Functions

The following sections explain how to perform a variety of operations on existing windows, such as setting window flags and drawing boxes around the window.

3.5.1 Drawing a Box

The box function draws a box around a window using the given characters to form the horizontal and vertical sides. The function call has the form:

```
box( win, vert, hor )
```

where *win* is the pointer to the desired window, *vert* is the vertical character, and *hor* is the horizontal character. Both *ver* and *hor* must have **char** type.

The function is typically used to distinguish one window from another when combining windows on a single screen. For example, in the following program fragment *box* creates a box around the window in the lower half of the screen.

```
WINDOW *cmdmenu;

cmdmenu = subwin(stdscr, 5, 80, 19, 0);
box(cmdmenu, '|', '-');
```

If necessary, the function will leave the corners of the box blank to prevent illegal scrolling.

3.5.2 Displaying Bold Characters

The *standout* and *wstandout* functions set the standout character attribute, causing characters subsequently added to the given window or screen to be displayed as bold characters.

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.

The functions are typically used after an error message or instructions have been added to a screen using the standout attribute. For example, in the following program fragment *standend* restores the normal attribute after an error message has been added to the standard screen.

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

3.5.4 Getting the Current Position

The *getyx* function copies the current line and column position of a given window pointer to a corresponding pair of variables. The function call has the form:

getyx(*win*, *y*, *x*)

where *win* is a pointer to the window containing the pointer to be examined, *y* is the integer variable to receive the line position, and *x* is the integer variable to receive the column position.

The function is typically used to save the current position so that the program can return to the position at a later time. For example, in the following program fragment *getyx* saves the current line and column position in the variables "line" and "column".

```
int line, column;

getyx(stdscr, line, column);
```

3.5.5 Setting Window Flags

The *leaveok*, *scrollok*, and *clearok* functions set or clear the cursor, scroll, and clear-screen flags. The flags control the action of the *refresh* function when called for the given window.

The *leaveok* function sets or clears the cursor flag which defines how the *refresh* function places the terminal cursor and the window pointer after updating the screen. If the flag is set, *refresh* leaves the cursor after the last character to be copied and moves the pointer to the corresponding position in the window. If the flag is cleared, *refresh* moves the cursor to the same position on the screen as the current pointer position in the window. The function call has the form:

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 in special cases only.

3.6 Combining Movement With Action

Many screen operations move the current position of a given window before performing an action on the window. For convenience, you can combine a number of functions with the movement prefix. This combination has the form:

mv*func* ([*win*,] *y*, *x* [, *arg*] ...)

where *func* is the name of a function, *win* is a pointer to the window to be operated on (*stdscr* used if none is given), *y* is an integer value giving the line to move to, *x* is an integer value giving the column to move to, and *arg* is a required argument for the given function. If more than one argument is required they must be separated with commas(,). For example, the function call

```
mvaddch(10, 5, 'X');
```

moves the position to (10,5) and adds the character "X". The operation is the same as moving the position with the *move* function and then adding a character with *addch*.

A complete list of the functions which may be used with the movement prefix is given in *curses*(S) in the XENIX *Reference Manual*.

3.7 Controlling the Terminal

The following sections explain how to set the terminal modes, how to move the cursor, and how to access other aspects of the terminal. These functions should only be used when using other screen processing functions.

3.7.1 Setting a Terminal Mode

The *crmode*, *echo*, *nl*, and *raw* functions set the terminal mode, causing subsequent input from the terminal's keyboard to be processed accordingly.

The *crmode* function sets the CBREAK mode for the terminal. The mode preserves the function of the signal keys, allowing allowing signals to be sent to a program from the keyboard, but disables the function of the editing keys. The function call has the form:

```
crmode()
```

No arguments are required.

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:

nonl()

No arguments are required.

The *noraw* function clears a terminal from RAW mode, restoring normal editing and signal generating function to the keyboard. The function call has the form:

noraw()

No arguments are required.

3.7.3 Moving the Terminal's Cursor

The *mvcur* function moves the terminal's cursor from one position to another in an optimal fashion. The function call has the form:

mvcur (*last_y*, *last_x*, *new_y*, *new_x*)

where *last_y* and *last_x* are integer values giving the last line and column position of the cursor, and *new_y* and *new_x* are integer values giving the new line and column position of the cursor. For example, the function call

mvcur(10, 5, 3, 0)

moves the cursor from (10,5) to (3,0) on the terminal screen.

Note

The *mvcur* function should only be used in programs that do not use other screen processing functions. This means the function can be used to perform optimal cursor motion without the aid of the other functions. For programs that do use other functions, the *move*, *wmove*, *refresh*, and *wrefresh* functions must be used to move the cursor.

3.7.4 Getting the Terminal Mode

The *gettmode* function returns the current tty mode. The function call has the form:

s = gettmode()

where *s* is the variable to receive the status.

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 *setterm* 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.

4.2.2 Converting to ASCII Characters

The *toascii* function converts non-ASCII characters to ASCII. The function call has the form:

```
c = toascii (i)
```

where *c* is the variable to receive the character, and *i* is the value to be changed. The function creates an ASCII character by truncating all but the low order 7 bits of the non-ASCII value. If the *i* value is already an ASCII character, no change takes place. For example, the function call

```
ascii = toascii(160)
```

converts value 160 to 32, the ASCII value of the space character.

The function is typically used to prepare non-ASCII characters for display at the standard output. For example, in the following program fragment *toascii* converts each character read from the file given by "oddstrm".

```
FILE *oddstrm;
int c;

c = toascii( getc( oddstrm ) );
if ( isprint(c) || isspace(c) )
        putchar(c);
```

If the resulting character is printable or is whitespace, it is written to the standard output.

4.2.3 Testing for Alphanumerics

The *isalnum* function tests for letters and decimal digits, i.e., the alphanumeric characters. The function call has the form:

```
isalnum (c)
```

where *c* is the character to test. The function returns a nonzero (true) value if the character is an alphanumeric, otherwise it returns zero (false). For example, the function call

```
isalnum('1')
```

returns a nonzero value, but the call

```
isalnum('>')
```

returns zero.

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*)

where *c* is the character to be tested. The function returns a nonzero value if the character is a digit, otherwise it returns zero. For example, in the following program fragment each new character in "c" is added to the running total if the character is a digit.

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

while ( isdigit( c=getc(infile) ) )
        num = num*10 + c-48;
```

4.2.7 Testing for a Hexadecimal Digit

The *isxdigit* function tests for a hexadecimal digit, that is, a character that is either a decimal digit or an uppercase or lowercase letter in the range A to F. The function call has the form:

isxdigit (*c*)

where *c* is the character to be tested. The function returns a nonzero value if the character is a digit, otherwise it returns zero. For example, in the following program fragment *isxdigit* tests whether a hexadecimal digit is read from the standard input.

```
int c;

c = getchar();
if ( isxdigit(c) )
        hexmode();
```

In this example, a function named *hexmode* is called if a hexadecimal digit is read.

4.2.8 Testing for Printable Characters

The *isprint* function tests for printable characters, i.e., characters whose ASCII values range from 32 to 126. The function call has the form:

isprint (*c*)

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

4.2.9 Testing for Punctuation

The *ispunct* function tests for punctuation characters, i.e., characters that are 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:

c = tolower (*i*)

and

c = toupper (*i*)

where *c* is the variable to receive the converted letter, and *i* is the letter to be converted. For example, the function call

```
lower = tolower('B')
```

converts "B" to "b" and assigns it to the variable "lower", and the call

```
upper = toupper('b')
```

converts "b" to "B" and assigns it to the variable "upper".

The *tolower* function returns the character unchanged if it is not an uppercase letter. Similarly, the *toupper* function returns the character unchanged if it is not a lowercase letter.

These functions are typically used to make the case of the characters read from a file or standard input consistent. For example, in the following statement *tolower* changes the character read from the standard input to lowercase before it is compared.

```
if ( tolower( getchar() ) != 'y')
        exit(0);
```

This conversion allows the user to type either "Y" or "y" to prevent the statement from executing the *exit* function.

4.3 Using the String Functions

The string functions concatenate, compare, copy, and count the number of characters in a string. Two special string functions, *sscanf* and *sprintf*, let a program read from and write to a string in the same way the standard input and output can be read and written. These functions are convenient when reading or writing whole lines containing values of several different formats.

Many string functions have two forms: a form that manipulates all characters in the string and one that manipulates a given number of characters. This gives programs very fine control over all or parts of strings.

4.3.1 Concatenating Strings

The *strcat* function concatenates two strings by appending the characters of one string to the end of another. The function call has 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.

4.3.3 Copying a String

The *strcpy* function copies a given string to a given location. The function call has the form:

```
strcpy (dst, src)
```

where *src* is a pointer to the string to be copied, and *dst* is a pointer to the location to receive the string. The function copies all characters in the source string *src* to the *dst* and appends a null character (\0) to the end of the new string. If *dst* contained a string before the copy, that string is destroyed. The function always returns the pointer to the new string.

For example, in the program fragment *strcpy* copies the string "not available" to the location given by "name".

```
char na[] = "not available";
char name[20];

strcpy( name, na );
```

Note that the location to receive a string must be large enough to contain the string. The function cannot detect overflow.

4.3.4 Getting a String's Length

The *strlen* function returns the number of character contained in a given string. The function call has the form:

```
strlen (s)
```

where *s* is a pointer to a string. The count includes all characters up to, but not including, the first null character. The return value is always an integer.

In the following program fragment, *strlen* is used to determine whether or not the contents of "inname" are short enough to be stored in "name".

```
char *inname;
char name[MAX];

if ( strlen(inname) < MAX )
        strcpy( name, inname);
```

4.3.5 Concatenating Characters to a String

The *strncat* function appends one or more characters to the end of a given string. The function call has the form:

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.

4.3.7 Copying Characters to a String

The *strncpy* function copies a given number of characters to a given string. The function call has the form:

strncpy (*dst*, *src*, *n*)

where *dst* is a pointer to the string to receive the characters, *src* is a pointer to the string containing the characters, and *n* is an integer value giving the number of characters to be copied. The function copies either the first *n* characters in *src* to *dst*, or if *src* has fewer than *n* characters, copies all characters up to the first null character. The function always returns the pointer *dst*.

In the following program fragment, *strncpy* copies the first three characters in "date" to "day".

```
char buf [MAX];
char date [29] = {"Fri Dec 29 09:35:44 EDT 1982"};
char *day = buf;

strncpy( day, date, 3);
```

In this example, "day" receives the string "Fri".

4.3.8 Reading Values from a String

The *sscanf* function reads one or more values from a given character string and stores the values at a given memory location. The function is similar to the *scanf* function which reads values from the standard input. The function call has the form:

sscanf (*s*, *format*, *argptr* ...)

where *s* is a pointer to the string to be read, *format* is a pointer to the string defining the format of the values to be read, and *argptr* is a pointer to the variable that is to receive the values read. If more than one *argptr* is given, they must be separated with commas. The *format* string may contain the same formats as given for *scanf* (see *scanf*(S) in the XENIX *Reference Manual*). The function always returns the number of values read.

The function is typically used to read values from a string containing several values of different formats, or to read values from a program's own input buffer. For example, in the following program fragment *sscanf* reads two values from the string pointed to by "datestr".

```
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:

```
system (command-line)
```

where *command-line* is a pointer to a string containing a shell command line. The command line must be exactly as it would be typed at the terminal, that is, it must begin with the program name followed by any required or optional arguments. For example, the call

```
system("date");
```

causes the system to execute the **date** command, which displays the current time and date at the standard output. The call

```
system("cat >response");
```

causes the system to execute the **cat** command. In this case, the standard output is redirected to the file *response*, so the command reads from the standard input and copies this input to the file *response*.

The *system* function is typically used in the same way as a function call to execute a program and return to the original program. For example, in the following program fragment *system* calls a program whose name is given in the string "cmd".

```
char *name, *cmd;

printf("Enter filename: ");
scanf("%s", name);
sprintf(cmd, "cat %s ", name);
system(cmd);
```

Note that the string in "cmd" is built using the *sprintf* function and contains the program name *cat* and an argument (the filename read by *scanf*). The effect is to execute the cat command with the given filename.

When using the *system* function, it is important to remember that buffered input and output functions, such as *getc* and *putc*, do not change the contents of their buffer until it is ready to be read or flushed. If a program uses one of these functions, then executes a command with the *system* function, that command may read or write data not intended for its use. To avoid this problem, the program should clear all buffered input and output before making a call to the *system* function. You can do this for output with the *fflush* function, and for input with the *setbuf* function described in the section "Using More Stream Functions" in Chapter 2.

5.4 Stopping a Program

The *exit* function stops the execution of a program by returning control to the system. 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:

```
execv (pathname, ptr);
```

where *pathname* is the full pathname of the program you want to execute, and *ptr* is pointer to an array of pointers. Each element in the array must point to a string. The array may have any number of elements, but the first element must point to a string containing the program name, and the last must be the null pointer, NULL.

The *execl* and *execv* functions are typically used in programs that execute in two or more phases and communicate through temporary files (for example a two-pass compiler). The first part of such a program can call the second part by giving the name of the second part and the appropriate arguments. For example, the following program fragment checks the status of "errflag", then either overlays the current program with the program *pass2*, or displays an error message and quits.

```
char *tmpfile;
int errflag;

if (errflag == 0)
        execl("/usr/bin/pass2", "pass2", tmpfile, NULL);
else {
        fprintf(stderr, "Error %d: Quitting", errflag);
        exit(2);
}
```

The *execv* function is typically used to pass arguments to a program when the precise number of arguments is not known beforehand. For example, the following program fragment reads arguments from the command line (beginning with the third one), copies the pointer of each to an element in "cmd", sets the last element in "cmd" to NULL, and executes the **cat** command.

```
char *cmd[ ];

cmd[0] = "cat";
for (i=3; i<argc; i++)
        cmd[i] = argv[i];
cmd[argc] = NULL;

execv("/bin/cat", cmd);
```

The *execl* and *execv* functions return control to the original program only if there is an error in finding the given program (e.g., a misspelled pathname or no execute permission). This allows the original program to check for errors and display an error message if necessary. For example, the following program fragment searches for the program *display* in the */usr/bin* directory.

```
execl("/usr/bin/display", "display", NULL);
fprintf(stderr, "Can't execute 'display' \n");
```

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" process, starts its execution at the same point, that is, just after the *fork* call. (The child never goes back to the beginning of the program to start execution.) The two processes are in effect synchronized, and continue to execute as independent programs.

The *fork* function returns a different value to each process. To the parent process, the function returns the process ID of the child. The process ID is always a positive integer and is always different than the parent's ID. To the child, the function returns 0. All other variables and values remain exactly as they were in the parent.

The return value is typically used to determine which steps the child and parent should take next. For example, in the program segment

```
char *cmd;

if (fork() == 0)
        execl("/bin/sh", "sh", "-c", cmd, NULL);
```

The child's return value, 0, causes the expression "fork() == 0", to be true, and therefore the *execl* function is called. The parent's return value, on the other hand, causes the expression to be false, and the function call is skipped. Executing the *execl* function causes the child to be overlayed by the program given by "command". This does not affect the parent.

If *fork* encounters an error and cannot create a child, it will return the value -1. It is a good idea to check for this value after each call.

5.8 Waiting for a Process

The *wait* function causes a parent process to wait until its child processes have completed their execution before continuing its own execution. The function call has the form:

wait (*ptr*)

where *ptr* is a pointer to an integer variable. It receives the termination status of the child from both the system and the child itself. The function normally returns the process ID of the terminated child, so the parent may check it against the value returned by *fork*.

The function is typically used to synchronize the execution of a parent and its child, and is especially useful if the parent and child processes access the same files. For example, the following program fragment causes the parent to wait while the program named by "pathname" (which has overlaid the child process) finishes its execution.

```
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 a program 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.

```
#include <stdio.h>

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

if (argc < 2) {
        fprintf(stderr,"No tty given.0);
        exit(1);
}
if (fork() == 0) {
        if (freopen(argv[1],"r",stdin) == NULL)
                exit(2);
        if (freopen(argv[1],"w",stdout) == NULL)
                exit(2);
        if (freopen(argv[1],"w",stderr) == NULL)
                exit(2);
        execl("/bin/sh","sh",NULL);
}
wait(&status);
if (status == 512)
        fprintf("Bad tty name: %s0, argv[1]);
}
```

In this example, the *fork* function creates a duplicate copy of the program. The child changes the standard input, output, and error files to the new terminal by closing and reopening them with the *freopen* function. The terminal name pointed to by "argv" must be the name of the device special file associated with the terminal, e.g., "/dev/tty03". The *execl* function then calls the shell which uses the new terminal as its standard input, output, and error files.

The parent process waits for the child to terminate. The *exit* function terminates the process if an error occurs when reopening the standard files. Otherwise, the process continues until the CNTRL-D key is pressed at the new terminal.

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.

```
FILE *pstrm;

pstrm = popen("cat >response","w");
```

The new pipe given by "pstrm" links the standard input of the command with the program. Data written to the pipe will be used as input by the cat command.

6.3 Reading and Writing to a Process

The *fscanf*, *fprintf*, and other stream functions may be used to read from or write to a pipe opened by the *popen* function. These functions have the same form as described in Chapter 2.

The *fscanf* function can be used to read from a pipe opened for reading. For example, in the following program fragment *fscanf* reads from the pipe given by *pstrm*.

```
FILE *pstrm;
char name[20];
int number;

pstrm = popen("cat","r");
fscanf(pstrm, "%s %d", name, &number);
```

This pipe is connected to the standard output of the cat command, so *fscanf* reads the first name and number written by cat to its standard output.

The *fprintf* function can be used to read from a pipe opened for writing. For example, in the following program fragment *fprintf* writes the string pointed to by "buf" to the pipe given by "pstrm".

```
FILE *pstrm;
char buf[MAX];

pstrm = popen("wc","w");
fprintf(pstrm,"%s",buf)
```

This pipe is connected to the standard input of the wc command, so the command reads and counts the contents of "buf".

6.4 Closing a Pipe

The *pclose* function closes the pipe opened by the *popen* function. The function call has the form:

pclose (*stream*)

```
FILE *pstrm;

pstrm = popen("cat >response","w");
```

The new pipe given by "pstrm" links the standard input of the command with the program. Data written to the pipe will be used as input by the cat command.

6.3 Reading and Writing to a Process

The *fscanf*, *fprintf*, and other stream functions may be used to read from or write to a pipe opened by the *popen* function. These functions have the same form as described in Chapter 2.

The *fscanf* function can be used to read from a pipe opened for reading. For example, in the following program fragment *fscanf* reads from the pipe given by *pstrm*.

```
FILE *pstrm;
char name[20];
int number;

pstrm = popen("cat","r");
fscanf(pstrm, "%s %d", name, &number);
```

'his pipe is connected to the standard output of the cat command, so *fscanf* ads the first name and number written by **cat** to its standard output.

e *fprintf* function can be used to read from a pipe opened for writing. For nple, in the following program fragment *fprintf* writes the string pointed to ›uf" to the pipe given by "pstrm".

```
'ILE *pstrm;
ıar buf[MAX];

rm = popen("wc","w");
ntf(pstrm,"%s",buf)
```

is connected to the standard input of the wc command, so the reads and counts the contents of "buf".

ing a Pipe

ıction closes the pipe opened by the *popen* function. The function m:

```
:am)
```

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.

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* function to create two copies of the original process. Each process has its own copy of the pipe. The child process decides whether it is supposed to read or write through the pipe, then closes the other side of the pipe and uses *execl* to create the new process and execute the desired program. The parent, on the other hand, closes the side of the pipe it does not use.

The sequence of *close* functions in the child process is a trick used to link the standard input or output of the child process to the pipe. The first *close* determines which side of the pipe should be closed and closes it. If "mode" is WRITE, the writing side is closed; if READ, the reading side is closed. The second *close* closes the standard input or output depending on the mode. If the mode is WRITE, the input is closed; if READ, the output is closed. The *dup* function creates a duplicate of the side of the pipe still open. Since the standard input or output was closed immediately before this call, this duplicate receives the same file descriptor as the standard file. The system always chooses the lowest available file descriptor for a newly opened file. Since the duplicate pipe has the same file descriptor as the standard file it becomes the standard input or output file for the process. Finally, the last *close* closes the original pipe, leaving only the duplicate.

The following example is a modified version of the *pclose* function. The modified version requires a file descriptor as an argument rather than a file pointer.

```
#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 signal, caused by pressing the QUIT key, or "SIGHUP" for hangup signal, caused by hanging up the line when connected to the system by modem. (Other constants for other signals are given in *signal*(S) in the XENIX *Reference Manual*.)

For example, the function call

```
signal(SIGINT, SIG_IGN)
```

changes the action of the interrupt signal to no action. The signal will have no effect on the program. The default action is usually to terminate the program.

The following sections show how to use the *signal* function to disable, change, and restore signals.

7.2.1 Disabling a Signal

You can disable a signal, i.e., prevent it from affecting a program, by using the "SIG_IGN" constant with *signal*. The function call has the form

signal (*sigtype*, SIG_IGN)

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

```
signal(SIGINT, SIG_IGN);
```

disables the interrupt signal.

The function call is typically used to prevent a signal from terminating a program executing in the background (e.g., a child process that is not using the terminal for input or output). The system passes signals generated from keystrokes at a terminal to all programs that have been invoked from that terminal. This means that pressing the INTERRUPT key to stop a program running in the foreground will also stop a program running in the background if it has not disabled that signal. For example, in the following program fragment *signal* is used to disable the interrupt signal for the child.

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

main ()
{
        FILE *fp;
        char *record[BUF], filename[MAX];

        signal (SIGINT, SIG_IGN);
        fp = fopen(filename, "a");
        fwrite(fp, BUF, record, 512);
        signal (SIGINT, SIG_DFL);

}
```

In this example, the interrupt signal is ignored while a record is record from the file given by "fp".

7.2.3 Catching a Signal

You can catch a signal and define your own action for it by providing a function that defines the new action and giving the function as an argument to *signal*. The function call has the form

signal (*sigtype*, *newptr*)

where *sigtype* is the manifest constant defining the signal to be caught, and *newptr* is a pointer to the function defining the new action. For example, the function call

signal(SIGINT, catch)

changes the action of the interrupt signal to the action defined by the function named *catch*.

The function call is typically used to let a program do additional processing before terminating. In the following program fragment, the function *catch* defines the new action for the interrupt signal.

```
#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.

7.2.4 Restoring a Signal

You can restore a signal to its previous value by saving the return value of a *signal* call, then using this value in a subsequent call. The function call has the form:

signal (*sigtype*, *oldptr*)

where *sigtype* is the manifest constant defining the signal to be restored and *oldptr* is the pointer value returned by a previous *signal* call.

The function call is typically used to restore a signal when its previous action may be one of many possible actions. For example, in the following program fragment the previous action depends solely on the return value of a function *keytest*.

```
#include <signal.h>

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

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

        savesig = signal (SIGINT, SIG_IGN);
        compute();
        signal(SIGINT, savesig);

}
```

In this example, the old pointer is saved in the variable "savesig". This value is restored after the function *compute* returns.

7.2.5 Program Example

This section shows how to use the *signal* function to create a modifed version of the *system* function. In this version, *system* disables all interrupts in the parent process until the child process has completed its operation. It then restores the signals to their previous actions.

```
#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.

Delaying a signal is especially useful in programs that must not be stopped at an arbitrary point. If, for example, a program updates a linked list, the action of a signal can be delayed to prevent the signal from interrupting the update and destroying the list. For example, in the following program fragment the function *delay* used to catch the interrupt signal sets the globally-defined flag "sigflag" and returns immediately to the point of interruption.

```
#include <signal.h>
int sigflag;

main ()
{
        int delay ();
        int (*savesig)();
        extern int sigflag;

        signal(SIGINT, delay); /* Delay the signal. */
        updatelist();
        savesig = signal(SIGINT, SIG_IGN); /* Disable the signal. */
        if (sigflag)
                /* Process delayed signals if any. */

}

delay ()
{
        extern int sigflag;

        sigflag=1;
}
```

In this example, if the signal is received while *updatelist* is executing, it is delayed until after *updatelist* returns. Note that the interrupt signal is disabled before processing the delayed signal to prevent a change to "sigflag" when it is being tested.

Note that the system automatically resets a signal to its default action immediately after the signal is processed. If your program delays a signal, make sure that the signal is redefined after each interrupt. Otherwise, the default action will be taken on the next occurrence of the signal.

7.3.2 Using Delayed Signals With System Functions

When a delayed signal is used to interrupt the execution of a XENIX system function, such as *read* or *wait*, the system forces the function to stop and return an error code. This action, unlike actions taken during execution of other functions, causes all processing performed by the system function to be discarded. A serious error can occur if a program interprets a system function error caused by delayed signals as a normal error. For example, if a program

Delaying a signal is especially useful in programs that must not be stopped at an arbitrary point. If, for example, a program updates a linked list, the action of a signal can be delayed to prevent the signal from interrupting the update and destroying the list. For example, in the following program fragment the function *delay* used to catch the interrupt signal sets the globally-defined flag "sigflag" and returns immediately to the point of interruption.

```
#include <signal.h>
int sigflag;

main ()
{
        int delay ();
        int (*savesig)();
        extern int sigflag;

        signal(SIGINT, delay);  /* Delay the signal. */
        updatelist();
        savesig = signal(SIGINT, SIG_IGN);  /* Disable the signal. */
        if (sigflag)
                /* Process delayed signals if any. */

}

delay ()
{
        extern int sigflag;

        sigflag=1;
}
```

In this example, if the signal is received while *updatelist* is executing, it is delayed until after *updatelist* returns. Note that the interrupt signal is disabled before processing the delayed signal to prevent a change to "sigflag" when it is being tested.

Note that the system automatically resets a signal to its default action immediately after the signal is processed. If your program delays a signal, make sure that the signal is redefined after each interrupt. Otherwise, the default action will be taken on the next occurrence of the signal.

7.3.2 Using Delayed Signals With System Functions

When a delayed signal is used to interrupt the execution of a XENIX system function, such as *read* or *wait*, the system forces the function to stop and return an error code. This action, unlike actions taken during execution of other functions, causes all processing performed by the system function to be discarded. A serious error can occur if a program interprets a system function error caused by delayed signals as a normal error. For example, if a program

The *longjmp* function has the form

```
longjmp (buffer)
```

where *buffer* is the variable containing the execution state. It must contain values previously saved with a *setbuf* function. The function copies the values in the *buffer* variable to the program counter, data and address registers, and the process status table. Execution continues as if it had just returned from the *setbuf* function which saved the previous execution state. For example, in the following program fragment *setbuf* saves the execution state of the program at the location just before the main processing loop and *longjmp* restores it on an interrupt signal.

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

main()
{
        int onintr();

        setjmp(sjbuf);
        signal(SIGINT, onintr);

        /* main processing loop */
}

onintr ()
{
printf("\nInterrupt\n");
longjmp(sjbuf);
}
```

In this example, the action of the interrupt signal as defined by *onintr* is to print the message "Interrupt" and restore the old execution state. When an interrupt signal is received in the main processing loop, execution passes to *onintr* which prints the message, then passes execution back to the main program function, making it appear as though control is returning from the *setbuf* function.

7.4 Using Signals in Multiple Processes

The XENIX system passes all signals generated at a given terminal to all programs invoked at that terminal. This means that a program has potential access to a signal even if that program is executing in the background or as a child to some other program. The following sections explain how signals may be used in multiple processes.

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.

7.4.2 Protecting Parent Processes

A program can create and wait for a child process that catches its own signals if and only if the program protects itself by disabling all signals before calling the *wait* function. By disabling the signals, the parent process prevents signals intended for the child processes from terminating its call to *wait*. This prevents serious errors that may result if the parent process continues execution before the child processes are finished.

For example, in the following program fragment the interrupt signal is disabled in the parent process immediately after the child is created.

```
#include <signal.h>

main ()
{
        int (*saveintr)();

        if (fork () == 0)
                execl( ... );

        saveintr = signal (SIGINT, SIG_IGN);
        wait( &status );
        signal (SIGINT, saveintr);
}
```

The signal's action is restored after the *wait* function returns normal control to the parent.

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.

The function is typically used to allocate storage for a group of strings that vary in length. For example, in the following program fragment *malloc* is used to allocate space for ten different strings, each of different length.

```
int i;
char *temp, *strings[10];
unsigned isize;

for ( i=0; i<10; i++) {
        scanf("%s", temp);
        isize = strlen(temp);
        string[i] = malloc(isize);
        }
```

In this example, the strings are read from the standard input. Note that the *strlen* function is used to get the size in bytes of each string.

8.2.2 Allocating Space for an Array

The *calloc* function allocates storage for a given array and initializes each element in the new array to zero. The function call has the form:

calloc (*n*, *size*)

where *n* is the number of elements in the array, and *size* is the number of bytes in each element. 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 memory. For example, the function call

```
table = calloc (10,4)
```

allocates sufficient space for a 10 element array. Each element has 4 bytes.

The function is typically used in programs which must process large arrays without knowing the size of an array in advance. For example, in the following program fragment *calloc* is used to allocate storage for an array of values read from the standard input.

```
int i;
char *table;
unsigned inum;

scanf("%d", &inum);
table = calloc (inum, 4);
for (i=0; i<inum; i++)
        scanf("%d", table[i]);
```

Note that the number of elements is read from the standard input before the elements are read.

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.

```
main ()
{
char *table;

if ( table[0] == -1 )
        free (table);
```

8.3 Locking Files

Locking a file is a way to synchronize file use when several processes may require access to a single file. The standard C library provides one file locking function, the *locking* function. This function locks any given section of a file, preventing all other processes which wish to use the section from gaining access. A process may lock the entire file or only a small portion. In any case, only the locked section is protected; all other sections may be accessed by other processes as usual.

File locking protects a file from the damage that may be caused if several processes try to read or write to the file at the same time. It also provides unhindered access to any portion of a file for a controlling process. Before a file can be locked, however, it must be prepared using the *open* and *lseek* functions described in Chapter 2, "Using the Standard I/O Functions." To use the *locking* function, you must add the line

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

to the beginning of the program. The file *sys/locking.h* contains definitions for the modes used with the function.

8.3.1 Preparing a File for Locking

Before a file can be locked, it must first be opened using the *open* function, then properly positioned by using the *lseek* function to move the file's character pointer to the first byte to be locked.

The *open* function is used once at the beginning of the program to open the file. The *lseek* function may be used any number of times to move the character pointer to each new section to be locked. For example, the following statements prepare the first 100 bytes beginning at the byte position 1024 from the beginning of the file *reservations* for locking.

```
fd = open("reservations", O_RDONLY)
lseek(fd, 1024, 0)
```

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.

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

main()
{
int fd, err;
char *data;


fd = open("data",2);          /* Open data for R/W */
if (fd == -1 )
        perror("");
else {
        lseek(fd, 100L, 0);             /* Seek to pos 100 */
        err = locking(fd, LK_LOCK, 100L); /* Lock bytes 100-200 */
        if (err == -1) {
                /* process error return */
                }

        /* read or write bytes 100 - 200 in the file */

        lseek(fd, 100L, 0);             /* Seek to pos 100 */
        locking(fd, LK_UNLCK, 100L);        /* Lock bytes 100-200 */

        }
}
```

8.4 Using Semaphores

The standard C library provides a group of functions, called the semaphore functions, which may be used to control the access to a given system resource. These functions create, open, and request control of "semaphores." Semaphores are regular files that have names and entries in the file system, but contain no data. Unlike other files, semaphores cannot be accessed by more than one process at a time. A process that wishes to take control of a semaphore away from another process must wait until that process relinquishes control. Semaphores can be used to control a system resource, such as a data file, by requiring that a process gain control of the semaphore before attempting to access the resource.

There are five semaphore functions: *creatsem*, *opensem*, *waitsem*, *nbwaitsem*, and *sigsem*. The *creatsem* function creates a semaphore. The semaphore may then be opened and used by other processes. A process can open a semaphore with the *opensem* function and request control of a semaphore with the *waitsem* or *nbwaitsem* function. Once a process has control of a semaphore it can carry out tasks using the given resource. All other processes must wait. When a process has finished accessing the resource, it can relinquish control of the semaphore with the *sigsem* function. This lets other processes get control of the semaphore and use the corresponding resource.

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 "ttyl" 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.

8.4.2 Opening a Semaphore

The *opensem* function opens an existing semaphore for use by the given process. The function call has the form:

opensem (*sem_name*)

where *sem_name* is a pointer to the name of the semaphore. This must be the same name used when creating the semaphore. The function returns a semaphore number that may be used in subsequent semaphore functions to refer to the semaphore. The function returns -1 if it encounters an error, such as trying to open a semaphore that does not exist or using the name of an existing regular file.

The function is typically used by a process just before it requests control of a given semaphore. A process need not use the function if it also created the semaphore. For example, in the following program fragment *opensem* is used to open the semaphore named *semaphore1*.

```
main ()
{
int sem1;

if ( (sem1 = opensem("semaphore1")) != -1)
        waitsem(sem1);
```

In this example, the semaphore number is assigned to the variable "sem1". If the number is not -1, then "sem1" is used in the semaphore function *waitsem* which requests control of the semaphore.

A semaphore must not be opened more than once during execution of a process.

8.4.3 Requesting Control of a Semaphore

The *waitsem* function requests control of a given semaphore for the calling process. If the semaphore is available, control is given immediately. Otherwise, the process waits. The function call has the form:

waitsem (*sem_num*)

where *sem_num* is the semaphore number of the semaphore to be controlled. If the semaphore is not available (if it is under control of another process), the function forces the requesting process to wait. If other processes are already waiting for control, the request is placed next in a queue of requests. When the semaphore becomes available, the first process to request control receives it. When this process relinquishes control, the next process receives control, and so on. The function returns -1 if it encounters an error such as requesting a 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 semaphore with the *waitsem* or *nbwaitsem* function. The function returns -1 if it encounters an error such as trying to take control of a semaphore that does not exist.

The function is typically used after a process has finished accessing the corresponding device or system resource. This allows waiting processes to take control. For example, in the following program fragment *sigsem* signals the end of control of the semaphore "tty1".

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

waitsem( tty1 );
while ((c=fgetc(temp)) != EOF)
        fputc(c, ftty1);
sigsem( tty1 );
```

This example also signals the end of the copy operation to the semaphore's corresponding device, given by "ftty1".

Note that a semaphore can become locked to a dead process if the process fails to signal the end of the control before terminating. In such a case, the semaphore must be reset by using the *creatsem* function.

8.4.6 Program Example

This section shows how to use the semaphore functions to control the access of a system resource. The following program creates five different processes which vie for control of a semaphore. Each process requests control of the semaphore five times, holding control for one second, then releasing it. Although, the program performs no meaningful work, it clearly illustrates the use of semaphores.

```
#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);
}
```

The program contains a number of global variables. The array "semf" contains the semaphore name. The name is used by the *creatsem* and *opensem* functions. The variable "sem_num" is the semaphore number. This is the value returned by *creatsem* and *opensem* and eventually used in *waitsem* and *sigsem*. Finally, the variable "holdsem" contains the number of times each process requests control of the semaphore.

The main program function uses the *mktemp* function to create a unique name for the semaphore and then uses the name with *creatsem* to create the semaphore. Once the semaphore is created, it begins to create child processes. These processes will eventually vie for control of the semaphore. As each child process is created, it opens the semaphore and calls the *doit* function. When control returns from *doit* the child process terminates. The parent process also calls the *doit* function, then waits for termination of each child process and finally deletes the semaphore with the *unlink* function.

The *doit* function calls the *waitsem* function to request control of the semaphore. The function waits until the semaphore is available, it then prints the process ID to the standard output, waits one second, and relinquishes control using the *sigsem* function.

Each step of the program is checked for possible errors. If an error is encountered, the program calls the *err* function. This function prints an error message and terminates the program.

8.5 Using Shared Data

Shared memory is a method by which one process shares its allocated data space with another. Shared memory allows processes to pool information in a central location and directly access that information without the burden of creating pipes or temporary files.

The standard C library provides several functions to access and control shared memory. The *sdget* function creates and/or adds a shared memory segment to a given process's data space. To access a segment, a process must signal its intention with the *sdenter* function. Once a segment has completed its access, it can signal that it is finished using the the segment with the *sdleave* function. The *sdfree* function is used to remove a segment from a process's data space. The *sdgetv* and *sdwaitv* functions are used to synchronize processes when several are accessing the segment at the same time.

To use the shared data functions, you must add the line

```
#include <sd.h>
```

at the beginning of the program. The *sd.h* file contains definitions for the mainfest constants and other macros used by the functions.

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.

8.5.2 Entering a Shared Data Segment

The *sdenter* signals a process's intention to access the contents of a shared data segment. A process cannot effectively access the contents of the segment unless it enters the segment. The function call has the form:

sdenter (*addr*, *flag*)

where *addr* is a character pointer to the segment to be accessed, and *flag* is an integer value which defines how the segment is to be accessed. The *flag* may be SD_RDONLY for indicating read only access to the segment, or SD_NOWAIT for returning an error if the segment is locked and another process is currently accessing it. These values may also be combined by logically ORing them.

The function normally waits for the segment to become available before allowing access to it. A segment is not available if the segment has been created without SD_UNLOCK flag and another process is currently accessing it.

In general, it is unwise to stay in a shared data segment any longer than it takes to examine or modify the desired location. The *sdleave* function should be used after each access. When in a shared data segment, a program should avoid using system functions. System functions can disrupt the normal operations required to support shared data and may cause some data to be lost. In particular, if a program creates a shared data segment that cannot be shared simultaneously, the program must not call the *fork* function when it is also accessing that segment.

8.5.3 Leaving a Shared Data Segment

The *sdleave* function signals a process's intention to leave a shared data segment after reading or modifying its contents. The function call has the form:

sdleave (*addr*)

where *addr* is a pointer with type **char** to the desired segment. The function returns -1 if it encounters an error, otherwise it returns 0. The return value is always an integer.

The function should be used after each access of the shared data to terminate the access. If the segment's lock flag is set, the function must be used after each access to allow other processes to access the segment. For example, in the following program fragment *sdleave* terminates each access to the segment given by "shared".

```
#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 has the form:

```
sdwaitv (addr, vnum)
```

where *addr* is a character pointer to the desired segment, and *vnum* is an integer value which defines the version number to wait on. The function normally returns the new version number. It returns -1 if it encounters an error. The return value is always an integer.

The function is typically used to synchronize the actions of two separate processes. For example, in the following program fragment the program waits while the program corresponding to the version number "radical_change" performs its operations in the segment.

```
#include <sd.h>

main ()
{
int radical_change = 3;

if ( sdwait ( sdseg, radical_change ) == -1 )
        fprintf(stderr, "Cannot find segment\n");
```

If an error occurs while waiting, an error message is printed.

8.5.6 Freeing a Shared Data Segment

The *sdfree* function detaches the current process from the given shared data segment. The function call has the form:

```
sdfree (addr)
```

where *addr* is a character pointer to the segment to be set free. The function returns the integer value 0, if the segment is freed. Otherwise, it returns -1.

If the process is currently accessing the segment, *sdfree* automatically calls *sdleave* to leave the segment before freeing it.

The contents of segments that have been freed by all attached processes are destroyed. To reaccess the segment, a process must recreate it using the *sdget* function and SD_CREAT flag.

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 number of the most recent XENIX system function error. Errors detected by system functions, such as access permission errors and lack of space, cause the system to set the errno variable to a number and return control to the program. The error number identifies the error condition. The variable may be used in subsequent statements to process the error.

The errno variable is typically used immediately after a system function has returned an error. In the following program fragment, errno is used to determine the course of action after an unsuccessful call to the *open* function.

```
if ( (fd=open("accounts", O_RDONLY)) == -1 )
        switch (errno) {
                case(EACCES):
                        fd = open("/usr/tmp/accounts",O_RDONLY);
                        break;
                default:
                        exit(errno);
        }
```

In this example, if errno is equal to EACCES (a manifest constant), permission to open the file *accounts* in the current directory is denied, so the file is opened in the directory */usr/tmp* instead. If the variable is any other value, the program terminates.

To use the errno variable in a program, it must be explicitly defined as an external variable with int type. Note that the file *errno.h* contains manifest constant definitions for each error number. These constants may be used in any program in which the line

```
#include <errno.h>
```

is placed at the beginning of the program. The meaning of each manifest constant is described in *Intro*(S) in the XENIX *Reference Manual.*

9.4 Printing Error Messages

The *perror* function copies a short error message describing the most recent system function error to the standard error file. The function call has the form:

perror (*s*)

where *s* is a pointer to a string containing additional information about the error.

The *perror* function places the given string before the error message and separates the two with a colon (:). Each error message corresponds to the current value of the errno variable. For example, in the following program fragment *perror* displays the message

```
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.*

Most system errors occur during calls to system functions. If the system error is recoverable, the system will return an error value to the program and set the errno variable to an appropriate value. No other information about the error is available.

Although the system lets two or more programs share a given resource, it does not keep close track of which program is using the resource at any given time. When an error occurs, the system returns an error value to all programs regardless of which caused the error. No information about which program caused the error is available.

System errors that occur during routine I/O operations initiated by the XENIX system itself generally do not affect user programs. Such errors cause the system to display appropriate system error messages on the system console.

Some system errors are not detected by the system until after the corresponding function has returned successfully. Such errors occur when data written to a file by a program has been queued for writing to disk at a more convenient time, or when a portion of data to be read from disk is found to already be in memory and the remaining portion is not read until later. In such cases, the system assumes that the subsequent read or write operation will be carried out successfully and passes control back to the program along with a successful return value. If operation is not carried out successfully, it causes a delayed error.

When a delayed error occurs, the system usually attempts to return an error on the next call to a system function that accesses the same file or resource. If the program has already terminated or does not make a suitable call, then the error is not reported.

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 procedure that is being setjmpedto. Hence, register variable values after a longjmp are the same as before a corresponding setjmp is called. If you need local variables to change between the call of setjmp and longjmp, they cannot be register variables.

A.1.2 Calling Sequence

The calling sequence is straightforward: arguments are pushed on the stack from the last to first: i.e., from right to left as you read them in the C source. The push quantum is 4 bytes, so if you are pushing a character, you must extend it appropriately before pushing. Structures and floating point numbers that are larger than 4 bytes are pushed in increments of 4 bytes so that they end up in the same order in stack memory as they are in any other memory. This means pushing the last word first and longword padding the last word (the first pushed) if necessary. The caller is responsible for removing his own arguments. Typically, an

```
addql  #constant,sp
```

is done. It is not really important whether the caller actually pushes and pops his arguments or just stores them in a static area at the top of the stack, but the debugger, *adb*, examines the **addql** or **addw** from the sp to decide how many arguments there were.

A.1.3 Stack Probes

XENIX is designed to dynamically allocate stack for local variables, function arguments, return addresses, etc. To do this, the XENIX kernel checks the offending instruction when a memory fault occurs. If it is a stack reference, the kernel maps enough stack memory for the instruction to complete its execution successfully. Then the procedure continues execution where it left off. Generally, this means restarting the offending memory reference instruction (usually a push or store). Unfortunately, the MC68000 does not provide a way to restart instructions.

Therefore, we need to perform a special instruction, which we call a *stack probe*, that potentially causes the memory fault, but that has no effect other than the memory fault itself. The kernel can then allocate any needed stack memory, ignore the fact that the stack probe instruction did not complete, and continue on to the next instruction. When we perform a stack probe and a memory fault occurs, the kernel allocates additional memory for the stack. The stack probe instruction for 68000 XENIX is

tstb −*value*(sp)

Value must be negative: since a negative index from the stack pointer is above the top of the stack— an otherwise absurd reference— XENIX knows that this instruction can only be a stack probe.

For the general case, use the following procedure entry sequence:

```
procedure_entry:
        link    a6,#−savesize
        tstb    −pushsize−slop−8(sp)
```

Any registers among d2−d7 and a2−a5 that are used in this procedure are saved with a **moveml** instruction after this sequence. The number of registers saved in the **moveml** needs to be accounted for in the push size. Thus, *pushsize* is the sum of the number of 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.

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

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 *ftime* functions. The *ftime* function, used with the *ctime* function, provides the default value for the time zone when the TZ environment variable has not been set. This means a binary configuration program can be used to change the default time zone. No source license is required.

B.5 Changes to the ioctl Function

XENIX 3.0 and UNIX System 3.0 have a full set of XENIX 2.3-compatible *ioctl* calls. Furthermore, XENIX 3.0 has resolved problems that previously hindered UNIX System 3.0 compatibility. For convenience, XENIX 2.3-compatible *ioctl* calls can be executed by a UNIX System 3.0 program. The available XENIX 2.3 *ioctl* calls are: TIOCSETP, TIOCSETN, TIOCGETP, TIOCSETC, TIOCGETC, TIOCEXCL, TIOCNXCL, TIOCHPCL, TIOCFLUSH, TIOCGETD, and TIOCSETD.

B.6 Pathname Resolution

If a null pathname is given, XENIX 2.3 interprets the name to be the current directory, but UNIX System 3.0 considers the name to be an error. XENIX 3.0 uses the version number in the *x.out* header to determine what action to take.

If the symbol ".." is given as a pathname when in a root directory that has been defined using the *chroot* function, XENIX 2.3 moves to the next higher directory. XENIX 3.0 also allows the ".." symbol, but restricts its use to the super-user.

B.7 Using the *mount* and *chown* Functions

Both XENIX 3.0 and UNIX System 3.0 restrict the use of the *mount* system call to the super-user. Also, both allow the owner of a file to use *chown* function to change the file ownership.

B.8 Super-Block Format

Both UNIX System 3.0 and UNIX System 5.0 have new super-block formats. XENIX 3.0 uses the System 5.0 format, but uses a different magic number for each revision. The XENIX 3.0 super-block has an additional field at the end which can be used to distinguish between XENIX 2.3 and 3.0 super-blocks. XENIX 3.0 checks this magic number at boot time and during a mount. If a XENIX 2.3 super-block is read, XENIX 3.0 converts it to the new format internally. Similarly, if a XENIX 2.3 super-block is written, XENIX 3.0 converts it back to the old format. This permits XENIX 2.3 kernels to be run on file systems also usable by UNIX System 3.0.

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.

CONTENTS

Programming Commands (CP)

strings	Finds the printable strings in an object
strip	Removes symbols and relocation bits
time	Times a command
tsort	Sorts a file topologically
unget	Undoes a previous get of an SCCS file
val	Validates an SCCS file
xref	Cross−references C programs
xstr	Extracts strings from C programs
yacc	Invokes a compiler−compiler

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 case of "normal" termination) one supplied by the program (see *wait*(S) and *exit*(S)). The former byte is 0 for normal termination; the latter is customarily 0 for successful execution and nonzero to indicate troubles such as erroneous parameters, or bad or inaccessible data. It is called variously "exit code", "exit status", or "return code", and is described only where special conventions are involved.

Notes

Not all commands adhere to the above syntax.

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

escape a '.

< *name* The value of *name*, which is either a variable name or a register name. *Adb* maintains a number of variables (see VARIABLES) named by single letters or digits. If *name* is a register name then the value of the register is obtained from the system header in *corfil*.

symbol A *symbol* is a sequence of upper or lower case letters, underscores or digits, not starting with a digit. The value of the *symbol* is taken from the symbol table in *objfil*. An initial _ or ~ will be prepended to *symbol* if needed.

_ *symbol*
In C, the 'true name' of an external symbol begins with an underscore (_). It may be necessary to use this name to distinguish it from the internal or hidden variables of a program.

(*exp*) The value of the expression *exp*.

Monadic operators

**exp* The contents of the location addressed by *exp* in *corfil*.

@*exp* The contents of the location addressed by *exp* in *objfil*.

−*exp* Integer negation.

~*exp* Bitwise complement.

Dyadic operators are left associative and are less binding than monadic operators.

e1+*e2* Integer addition.

e1−*e2* Integer subtraction.

*e1***e2* Integer multiplication.

e1%*e2* Integer division.

e1&*e2* Bitwise conjunction.

e1|*e2* Bitwise disjunction.

e1#*e2* *E1* rounded up to the next multiple of *e2*.

COMMANDS

Most commands consist of a verb followed by a modifier or list of modifiers. The following verbs are available. (The commands '?' and '/' may be followed by '*'; see ADDRESSES for further details.)

?*f* Locations starting at *address* in *objfil* are printed according 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 are checked to ensure that they have an appropriate type as indicated below.

/ local or global data symbol
? local or global text symbol
= local or global absolute symbol

p 2 Print the addressed value in symbolic form using the same rules for symbol lookup as **a**.

t 0 When preceded by an integer tabs to the next appropriate tab stop. For example, **8t** moves to the next 8−space tab stop.

r 0 Print a space.

n 0 Print a newline.

"..." 0 Print the enclosed string.

^ *Dot* is decremented by the current increment. Nothing is printed.

\+ *Dot* is incremented by 1. Nothing is printed.

− *Dot* is decremented by 1. Nothing is printed.

newline If the previous command temporarily incremented *dot*, make the increment permanent. Repeat the previous command with a *count* of 1.

[?/]**l** *value mask*

Words starting at *dot* are masked with *mask* and compared with *value* until a match is found. If **L** is used then the match is for 4 bytes at a time instead of 2. If no match is found then *dot* is unchanged; otherwise *dot* is set to the matched location. If *mask* is omitted then −1 is used.

[?/]**w** *value* ...

Write the 2−byte *value* into the addressed location. If the command is **W**, write 4 bytes. Odd addresses are not allowed when writing to the subprocess address space.

[?/]**m** *b1 e1 f1*[?/]

New values for (*b1*, *e1*, *f1*) are recorded. If less than three expressions are given then the remaining map parameters are left unchanged. If the '?' or '/' is followed by '*' then the second segment (*b2*, *e2*, *f2*) of the mapping is changed. If the list is terminated by '?' or '/' then the file (*objfil* or *corfil* respectively) is used for subsequent requests. (So that, for example, '/m?' will cause '/' to refer to *objfil*.)

>*name* *Dot* is assigned to the variable or register named.

! 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

the same line as the command. An argument starting with < or > causes the standard input or output to be established for the command. All signals are turned on on entry to the subprocess.

c*s* The subprocess is continued with signal *s* c *s*, see *signal*(S). If *address* is given then the subprocess is continued at this address. If no signal is specified then the signal that caused the subprocess to stop is sent. Breakpoint skipping is the same as for **r**.

s*s* As for **c** except that the subprocess is single stepped *count* times. If there is no current subprocess then *objfil* is run as a subprocess as for **r**. In this case no signal can be sent; the remainder of the line is treated as arguments to the subprocess.

k The current subprocess, if any, is terminated.

VARIABLES

Adb provides a number of variables. Named variables are set initially by *adb* but are not used subsequently. Numbered variables are reserved for communication as follows.

0 The last value printed.
1 The last offset part of an instruction source.
2 The previous value of variable 1.

On entry the following are set from the system header in the *corfil*. If *corfil* does not appear to be a **core** file then these values are set from *objfil*.

b The base address of the data segment.
d The data segment size.
e The entry point.
s The stack segment size.
t The text segment size.

ADDRESSES

The address in a file associated with a written address is determined by a mapping associated with that file. Each mapping is represented by two triples (*b1, e1, f1*) and (*b2, e2, f2*) and the *file address* corresponding to a written *address* is calculated as follows.

$b1 \leq address < e1$ => *file address*=*address*+$f1-b1$, otherwise,

$b2 \leq address < e2$ => *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.

Name

admin – Creates and administers SCCS files.

Syntax

admin [– **n**] [– i[name]] [– rrel] [– t[name]] [– fflag[flag-val]] [– dflag[flag-val]] [– alogin] [– elogin] [– m[mrlist]] [– y[comment]] [– **h**] [– z] files

Description

Admin is used to create new SCCS files and to change parameters of existing ones. Arguments to *admin*may appear in any order. They consist of options, which begin with – , and named files (note that SCCS filenames must begin with the characters s.). If a named file doesn't exist, it is created, and its parameters are initialized according to the specified options. Parameters not initialized by a option are assigned a default value. If a named file does exist, parameters corresponding to specified options are changed, and other parameters are left as is.

If a directory is named, *admin* 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 the dash – 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 options are as follows. Each is explained as though only one named file is to be processed since the effects of the arguments apply independently to each named file.

– n This option indicates that a new SCCS file is to be created.

– i[*name*] The *name* of a file from which the text for a new SCCS file is to be taken. The text constitutes the first delta of the file (see – **r** below for delta numbering scheme). If the i option is used, but the filename is omitted, the text is obtained by reading the standard input until an end-of-file is encountered. If this option is omitted, then the SCCS file is created empty. Only one SCCS file may be created by an *admin* command on which the i option is supplied. Using a single *admin* to create two or more SCCS files require that they be created empty (no – i option). Note that the – **i** option implies the – **n** option.

– 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 which deltas can no longer be made (get – e against one of these "locked" releases fails). The *list* has the following syntax:

<list> ::= <range> | <list> , <range>
<range> ::= *RELEASE NUMBER* | a

The character a in the *list* is equivalent to specifying *all releases* for the named SCCS file.

n Causes *delta*(CP) to create a "null" delta in each of those releases (if any) being skipped when a delta is made in a *new* release (e.g., in making delta 5.1 after delta 2.7, releases 3 and 4 are skipped). These null deltas serve as "anchor points" so that branch deltas may later be created from them. The absence of this flag causes skipped releases to be nonexistent in the SCCS file preventing branch deltas from being created from them in the future.

q*text* User-definable text substituted for all occurrences of the keyword in SCCS file text retrieved by *get*(CP).

m*mod* *Mod*ule name of the SCCS file substituted for all occurrences of the admin.CP keyword in SCCS file text retrieved by *get*(CP). If the m flag is not specified, the value assigned is the name of the SCCS file with the leading s. removed.

t*type* *Type* of module in the SCCS file substituted for all occurrences of
keyword in SCCS file text retrieved by *get*(CP).

v[*pgm*] Causes *delta*(CP) to prompt for Modification Request (MR) numbers as the reason for creating a delta. The optional value specifies the name of an MR number validity checking program (see *delta*(CP)). (If this flag is set when creating an SCCS file, the m option must also be used even if its value is null).

– d[*flag*] Causes removal (deletion) of the specified *flag* from an SCCS file. The – d option may be specified only when processing existing SCCS files. Several – d options may be supplied on a single *admin* command. See the – f option for allowable *flag* names.

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.

– z The SCCS file checksum is recomputed and stored in the first line of the SCCS file (see – h, above).

Note that use of this option on a truly corrupted file may prevent future detection of the corruption.

Files

The last component of all SCCS filenames must be of the form s.*file-name*. New SCCS files are created read-only (444 modified by umask) (see *chmod*(C)). Write permission in the pertinent directory is, of course, required to create a file. All writing done by *admin* is to a temporary x-file, called x.*filename*, (see *get*(CP)), created with read-only permission if the *admin* command is creating a new SCCS file, or with the same mode as the SCCS file if it exists. After successful execution of *admin*, the SCCS file is removed (if it exists), and the x-file is renamed with the name of the SCCS file. This ensures that changes are made to the SCCS file only if no errors occurred.

It is recommended that directories containing SCCS files be mode 755 and that SCCS files themselves be read-only. The mode of the directories allows only the owner to modify SCCS files contained in the directories. The mode of the SCCS files prevents any modification at all except by SCCS commands.

If it should be necessary to patch an SCCS file for any reason, the mode may be changed to 644 by the owner allowing use of a text editor. *Care must be taken!* The edited file should *always* be processed by an admin – h to check for corruption followed by an admin – z to generate a proper checksum. Another admin – h is recommended to ensure the SCCS file is valid.

Admin also makes use of a transient lock file (called z.*filename*), which is used to prevent simultaneous updates to the SCCS file by different users. See *get*(CP) for further information.

See Also

delta(CP), ed(C), get(CP), help(CP), prs(CP), what(C), sccsfile(F)

Diagnostics

Use *help*(CP) for explanations.

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 file.

v Verbose. Under the verbose option, *ar* gives a file-by-file description of the making of a new archive file from the old archive and the constituent files. When used with t, it gives a long listing of all information about the files. When used with x, it precedes each file with a name.

c Create. Normally *ar* will create *afile* when it needs to. The create option suppresses the normal message that is produced when *afile* is created.

l Local. Normally *ar* places its temporary files in the directory /tmp. This option causes them to be placed in the local directory.

Files

/tmp/v* Temporary files

See Also

ld(CP), lorder(CP), ar(F)

Notes

If the same file is mentioned twice in an argument list, it may be put in the archive twice.

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

cb – Beautifies C programs.

Syntax

cb [file]

Description

Cb places a copy of the C program in *file* (standard input if *file* is not given) on the standard output with spacing and indentation that displays the structure of the program.

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.

−K Do not generate stack probes. Stack probes are necessary for XENIX user programs to assure proper stack growth.

Other arguments
are taken to be either loader option arguments, or C−compatible object programs, typically produced by an earlier *cc* run, or perhaps libraries of C−compatible routines. These programs, together with the results of any compilations specified, are loaded (in the order given) to produce an executable program with name **a.out.**

Files

file.c	input file
file.o	object file
a.out	loaded output
file.[isx]	temporaries for *cc*
/lib/cpp	preprocessor
/lib/c68	compiler for *cc*
/lib/c68o	optional optimizer
/lib/crt0.o	runtime startoff
/lib/libc.a	standard library, see *intro*(S)
/usr/include	standard directory for '#include' files

See Also

B. W. Kernighan and D. M. Ritchie, *The C Programming Language,* Prentice−Hall, 1978
D. M. Ritchie, *C Reference Manual*
adb(CP), ld(CP)

DIAGNOSTICS

The diagnostics produced by C itself are intended to be self−explanatory. Occasional messages may be produced by the assembler or the loader. Of these, the most mystifying are from the assembler, *as*(C), which produces line number reports based on the generated code, which is only loosely related to the source linenumber. Running the compiler with the −S option and assem−bling the result by hand may help you resolve the difficulty.

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 *ID*entification (*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.

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** option).

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 validates the correctness of the MR numbers. If a nonzero exit status is returned from the MR number validation program, *cdc* terminates and the delta commentary remains unchanged.

− **y**[*comment*] Arbitrary text used to replace the *comment*(s) already existing for the delta specified by the − **r** option. The previous comments are kept and preceded by a comment line stating that they were changed. A null *comment* has no effect.

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.

In general, if you made the delta, you can change its delta commentary; or if you own the file and directory you can modify the delta commentary.

Examples

The following:

```
cdc - r1.6 - m"bl78-12345 !bl77-54321 bl79-00001" - ytrouble
s.file
```

adds bl78-12345 and bl79-00001 to the MR list, removes bl77-54321 from the MR list, and adds the comment trouble to delta 1.6 of s.file.

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.

Name

comb – Combines SCCS deltas.

Syntax

comb [– o] [– s] [– psid] [– clist] files

Description

Comb provides the means to combine one or more deltas in an SCCS file and make a single new delta. The new delta replaces the previous deltas, making the SCCS file smaller than the original.

Comb does not perform the combination itself. Instead, it generates a shell procedure that you must save and execute to reconstruct the given SCCS files. *Comb* copies the generated shell procedure to the standard output. To save the procedure, you must redirect the output to a file. The saved file can then be executed like any other shell procedure (see *sh*(C)).

When invoking *comb*, arguments may be specified in any order. All options apply to all named SCCS files. If a directory is named, *comb* 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.

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

– p*SID* The *S*CCS *ID*entification string (SID) of the oldest delta to be preserved. All older deltas are discarded in the reconstructed file.

– c*list* A *list* (see *get*(CP) for the syntax of a *list*) of deltas to be preserved. All other deltas are discarded.

– o For each get – e generated, this argument causes the reconstructed file to be accessed at the release of the delta to be created, otherwise the reconstructed file would be accessed at the most recent ancestor. Use of the – o option may decrease the size of the reconstructed SCCS file. It may also alter the shape of the delta tree of the original file.

– 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.

Name

config - configure a XENIX system

Syntax

/etc/config [- t] [- l file] [- c file] [- m file] dfile

Description

Config is a program that takes a description of a XENIX system and generates a file which is a C program defining the configuration tables for the various devices on the system.

The - c option specifies the name of the configuration table file; **c.c** is the default name.

The - m option specifies the name of the file that contains all the information regarding supported devices; /etc/master is the default name. This file is supplied with the XENIX system and should ***not*** be modified unless the user *fully* understands its construction.

The - t option requests a short table of major device numbers for character and block type devices. This can facilitate the creation of special files.

The user must supply *dfile*; it must contain device information for the user's system. This file is divided into two parts. The first part contains physical device specifications. The second part contains system-dependent information. Any line with an asterisk (*) in column 1 is a comment.

All configurations are assumed to have a set of required devices which must be present to run XENIX such as the system clock. These devices ***must not*** be specified in *dfile*.

First Part of *dfile*

Each line contains two fields, delimited by blanks and/or tabs in the following format:

    devname number

where ***devname*** is the name of the device (as it appears in the /etc/master device table), and ***number*** is the number (decimal) of devices associated with the corresponding controller; ***number*** is optional, and if omitted, a default value which is the maximum value for that controller is used.

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

We must also specify the following parameter information:
- root device is an HD (pseudo disk 3)
- pipe device is an HD (pseudo disk 3)
- swap device is an HD (pseudo disk 2)
  - with a swplo of 1 and an nswap of 2300
- number of buffers is 50
- number of processes is 50
- maximum number of processes per user ID is 15
- number of mounts is 8
- number of inodes is 120
- number of files is 120
- number of calls is 30
- number of texts is 35
- number of character buffers is 150
- number of swapmap entries is 50
- number of memory pages is 512
- number of file locks is 100
- timezone is pacific time
- daylight time is in effect

The actual system configuration would be specified as follows:

```
hd   1
fd   1
root    hd   3
pipe    hd   3
swap    hd   2    0    2300
* Comments may be inserted in this manner
buffers 50
procs        150
maxproc 15
mounts       8
inodes       120
files        120
calls        30
texts        35
clists       150
swapmap      50
pages (1024/2);
locks 100
timezone (8*60)
daylight 1
```

Files

/etc/master	default input master device table
c.c	default output configuration table file

See Also

master(F)

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.

Name

cref – Makes a cross-reference listing.

Syntax

cref [– acilnostux123] files

Description

Cref makes a cross-reference listing of assembler or C programs. The program searches the given *files* for symbols in the appropriate C or assembly language syntax.

The output report is in four columns:

1. Symbol
2. Filename
3. Current symbol or line number
4. Text as it appears in the file

Cref uses either an *ignore* file or an *only* file. If the – i option is given, the next argument is taken to be an *ignore* file; if the – o option is given, the next argument is taken to be an *only* file. *Ignore* and *only* files are lists of symbols separated by newlines. All symbols in an *ignore* file are ignored in columns 1 and 3 of the output. If an *only* file is given, only symbols in that file will appear in column 1. Only one of these options may be given; the default setting is – i using the default ignore file (see *FILES* below). Assembler predefined symbols or C keywords are ignored.

The – s option causes current symbols to be put in column 3. In the assembler, the current symbol is the most recent name symbol; in C, the current function name. The – l option causes the line number within the file to be put in column 3.

The – t option causes the next available argument to be used as the name of the intermediate file (instead of the temporary file /tmp/crt??). This file is created and is *not* removed at the end of the process.

The *cref* options are:

a Uses assembler format (default)

c Uses C format

i Uses an *ignore* file (see above)

l Puts line number in column 3 (instead of current symbol)

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.

Name

ctags – Creates a tags file.

Syntax

ctags [– u] [– w] [– x] name ...

Description

Ctags makes a tags file for *vi*(C) from the specified C sources. A tags file gives the locations of specified objects (in this case functions) in a group of files. Each line of the tags file contains the function name, the file in which it is defined, and a scanning pattern used to find the function definition. These are given in separate fields on the line, separated by blanks or tabs. Using the *tags* file, *vi* can quickly find these function definitions.

If the – x flag is given, *ctags* produces a list of function names, the line number and file name on which each is defined, as well as the text of that line and prints this on the standard output. This is a simple index which can be printed out as an off-line readable function index.

Files whose name ends in .c or .h are assumed to be C source files and are searched for C routine and macro definitions.

Other options are:

– w Suppresses warning diagnostics.

– u Causes the specified files to be *updated* in tags; that is, all references to them are deleted, and the new values are appended to the file. (Beware: this option is implemented in a way which is rather slow; it is usually faster to simply rebuild the *tags* file.)

The tag *main* is treated specially in C programs. The tag formed is created by prepending *M* to the name of the file, with a trailing .c removed, if any, and leading pathname components also removed. This makes use of *ctags* practical in directories with more than one program.

Files

tags   Output tags file

See Also

ex(C), vi(C)

Credit

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

Name

delta – Makes a delta (change) to an SCCS file.

Syntax

delta [– rSID] [– s] [– n] [– glist] [– m[mrlist]] [– y[comment]] [– p] files

Description

Delta is used to permanently introduce into the named SCCS file changes that were made to the file retrieved by *get*(CP) (called the *g-file*, or generated file).

Delta makes a delta to each SCCS file named by *files*. If a directory is named, *delta* 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.

Delta may issue prompts on the standard output depending upon certain options specified and flags (see *admin*(CP)) that may be present in the SCCS file (see – **m** and – **y** options below).

Options apply independently to each named file.

– r*SID* Uniquely identifies which delta is to be made to the SCCS file. The use of this keyletter is necessary only if two or more versions of the same SCCS file have been retrieved for editing (get – e) by the same person (login name). The SID value specified with the – **r** keyletter can be either the SID specified on the *get* command line or the SID to be made as reported by the *get* command (see *get*(CP)). A diagnostic results if the specified SID is ambiguous, or if it is necessary and omitted on the command line.

– s Suppresses the issue, on the standard output, of the created delta's SID, as well as the number of lines inserted, deleted and unchanged in the SCCS file.

– n Specifies retention of the edited *g-file* (normally removed at completion of delta processing).

– 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*.

p-file	Existed before the execution of *delta*; may exist after completion of *delta*.
q-file	Created during the execution of *delta*; removed after completion of *delta*.
x-file	Created during the execution of *delta*; renamed to SCCS file after completion of *delta*.
z-file	Created during the execution of *delta*; removed during the execution of *delta*.
d-file	Created during the execution of *delta*; removed after completion of *delta*.
/usr/bin/bdiff	Program to compute differences between the "retrieved" file and the *g-file*.

Warning

Lines beginning with an **SOH** ASCII character (binary 001) cannot be placed in the SCCS file unless the **SOH** is escaped. This character has special meaning to SCCS (see *sccsfile*(F)) and will cause an error.

A *get* of many SCCS files, followed by a *delta* of those files, should be avoided when the *get* generates a large amount of data. Instead, multiple *get/delta* sequences should be used.

If the standard input (–) is specified on the *delta* command line, the **– m** (if necessary) and **– y** options *must* also be present. Omission of these options causes an error to occur.

See Also

admin(CP), bdiff(C), get(CP), help(CP), prs(CP), sccsfile(F)

Diagnostics

Use *help*(CP) for explanations.

ist (see *get*(CP) for the definition of *list*) /hich are to be *ignored* when the file is t the change level (SID) created by this

CS file has the v flag set (see *admin*(CP)) Aodification Request (MR) number *must* be as the reason for creating the new delta.

is not used and the standard input is a termi-e prompt MRs? is issued on the standard out-;fore the standard input is read; if the standard is not a terminal, no prompt is issued. The ? prompt always precedes the comments? ıpt (see – y keyletter).

s in a list are separated by blanks and/or tab ıracters. An unescaped newline character ter-ınates the MR list.

/ote that if the v flag has a value (see *admin*(CP)), t 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).

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.

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.

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

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

GET (CP)

- k] [- e]

:ording . The to all ough that with is t is in,

e

gets for editing on the same SID until *delta* is executed or the j (joint edit) flag is set in the SCCS file (see *admin*(CP)). Concurrent use of get – e for different SIDs is always allowed.

If the *g-file* generated by *get* with an – e option is accidentally ruined in the editing process, it may be regenerated by reexecuting the *get* command with the – k option in place of the – e option.

SCCS file protection specified via the ceiling, floor, and authorized user list stored in the SCCS file (see *admin*(CP)) are enforced when the – e option is used.

– b
: Used with the – e option to indicate that the new delta should have an SID in a new branch. This option is ignored if the b flag is not present in the file (see *admin*(CP)) or if the retrieved *delta* is not a leaf *delta*. (A leaf *delta* is one that has no successors on the SCCS file tree.)

  Note: A branch *delta* may always be created from a non-leaf *delta*.

– i*list*
: A *list* of deltas to be included (forced to be applied) in the creation of the generated file. The *list* has the following syntax:

  <list> ::= <range> | <list> , <range>
  <range> ::= SID | SID – SID

  SID, the SCCS Identification of a delta, may be in any form described in Chapter 5, "SCCS: A Source Code Control System," in the XENIX *Programmer's Guide*.

– x*list*
: A *list* of deltas to be excluded (forced not to be applied) in the creation of the generated file. See the – i option for the *list* format.

– k
: Suppresses replacement of identification keywords (see below) in the retrieved text by their value. The – k option is implied by the – e option.

– l[p]
: Causes a delta summary to be written into an *l-file*. If – lp is used then an *l-file* is not created; the delta summary is written on the standard output instead. See *FILES* for the format of the *l-file*.

– p
: Causes the text retrieved from the SCCS file to be written on the standard output. No *g-file* is created. All output that normally goes to the standard output goes to file descriptor 2 instead, unless the – s option is used, in which case it disappears.

– **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

(see *get*(CP) for the definition of *list*)
ch are to be *ignored* when the file is
he change level (SID) created by this

file has the v flag set (see *admin*(CP))
ification Request (MR) number *must* be
the reason for creating the new delta.

ot used and the standard input is a termi-
ompt MRs? is issued on the standard out-
the standard input is read; if the standard
ot a terminal, no prompt is issued. The
ompt always precedes the comments?
see – y keyletter).

a list are separated by blanks and/or tab
rs. An unescaped newline character ter-
the MR list.

at if the v flag has a value (see *admin*(CP)),
ken to be the name of a program (or shell
ure) which will validate the correctness of the
umbers. If a nonzero exit status is returned
MR number validation program, *delta* ter-
es (it is assumed that the MR numbers were
ll valid).

trary text used to describe the reason for mak-
the delta. A null string is considered a valid
ment.

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

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) for-
mat.

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

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

e used in the

ag in the file
the SCCS file

%S%) of the

D).
Y).
).
file (see

r iden-
"this
ended
rs.
).
s for
ntal-

(C)
les.

n
e
e
f

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 created mode 444.

See Also

admin(CP), delta(CP), help(CP), prs(CP), what(C), sccsfile(F)

Diagnostics

Use *help*(CP) for explanations.

Notes

If the effective user has write permission (either explicitly or implicitly) in the directory containing the SCCS files, but the real user doesn't, then only one file may be named when the – e option is used.

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)

Name

hdr - Displays selected parts of object files.

Syntax

hdr [- dhprsSt] file ...

Description

Hdr displays object file headers, symbol tables, and text or data relocation records in human-readable formats. It also prints out seek positions for the various segments in the object file.

A.out, x.out, and x.out segmented formats and archives are understood.

The symbol table format consists of six fields. In a.out formats the third field is missing. The first field is the symbol's index or position in the symbol table, printed in decimal. The index of the first entry is zero. The second field is the type, printed in hexadecimal. The third field is the s_seg field, printed in hexadecimal. The fourth field is the symbol's value in hexadecimal. The fifth field is a single character which represents the symbol's type as in *nm*(CP), except C common is not recognized as a special case of undefined. The last field is the symbol name.

If long form relocation is present, the format consists of six fields. The first is the descriptor, printed in hexadecimal. The second is the symbol ID, or index, in decimal. This field is used for external relocations as an index into the symbol table. It should reference an undefined symbol table entry. The third field is the position, or offset, within the current segment at which relocation is to take place; it is printed in hexadecimal. The fourth field is the name of the segment referenced in the relocation: text, data, bss or EXT for external. The fifth field is the size of relocation: byte, word (2 bytes), or long. The last field will indicate, if present, that the relocation is relative.

If short form relocation is present, the format consist of three fields. The first field is the relocation command in hexadecimal. the second field contains the name of the segment referenced; text or data. The last field indicates the size of relocation: word or long.

Options and their meanings are:

- h Causes the object file header and extended header to be printed out. Each field in the header or extended header is labeled. This is the default option.

– **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

help – Asks for help about SCCS commands.

Syntax

help [args]

Description

Help finds information to explain a message from an SCCS command or explain the use of a command. Zero or more arguments may be supplied. If no arguments are given, *help* will prompt for one.

The arguments may be either message numbers (which normally appear in parentheses following messages) or command names. There are the following types of arguments:

type 1 Begins with nonnumerics, ends in numerics. The nonnumeric prefix is usually an abbreviation for the program or set of routines which produced the message (e.g., ge6, for message 6 from the *get* command).

type 2 Does not contain numerics (as a command, such as get)

type 3 Is all numeric (e.g., 212)

The response of the program will be the explanatory information related to the argument, if there is any.

When all else fails, try "help stuck".

Files

/usr/lib/help Directory containing files of message text

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**

, **−i** , or **−F** options can be used to produce different types of executable files.

Ld understands several options. Except for **−l**, they should appear before the names of all object file arguments.

−s 'Strip' the output to save space by removing the symbol table and relocation records. Note that stripping impairs the usefulness of the debugger. This information can also be removed later with *strip*(CP).

−sr Do not attach the short form of relocation. This does *not* imply removing the symbol table, as with **−s** .

−u Take the following argument as a symbol and enter it as undefined in the symbol table. This is useful for loading wholly from a library, since initially the symbol table is empty and an unresolved reference is needed to force the loading of the first routine.

−U Discard all symbols except those that are undefined external.

−g The same as **−U**, except also retain the following list of global symbols. The list consists of the next command line arguments and is terminated by the end of the command line, by − alone, or by any further option beginning with a −.

−G The same as **−g**, except that the list of global symbols is taken from the file named by the following argument. If the next argument is − alone, the standard input is read. The symbols may be separated by any type of whitespace.

−lx This option is an abbreviation for the library name '/lib/lib*x*.a', where *x* is a string. If the library does not exist, *ld* then tries '/usr/lib/lib*x*.a'. A library is searched when its name is encountered, so the placement of a **−l** is significant. Note that **−l** with no argument, defaults to **−lc** . If the processor on which **ld** is running is not the same as the target processor, then it is possible that **−p** may be implied. In the case of the MC68000 target, **−p /usr/lib/mlib** is implied.

−p Take the following argument as the directory in which **−lx** libraries will be found.

−x Do not preserve local (non.globl) symbols in the output 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 segment. With **−nn**, it is used to compute the base of the data segment. With **−nr**, it is used to compute the base of the text segment.

−R The next argument is taken to be a hexadecimal number that is used as the base address for text relocation. With **−i** or **−nn** , it also specifies the text base address; with **−nr** it specifies the data base address.

−F The next argument is taken to be a hexadecimal number that specifies the size of the stack required by the object file when executing. This only has meaning on those processors that cannot expand the stack dynamically.

Files

/lib/lib*.a	libraries
/usr/mlib/lib*.a	more libraries
x.out	output file

See Also

as(CP), ar(CP), cc(CP), ranlib(CP), strip(CP), x.out(F)

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 and write to, defaulted to stdin and stdout, respectively.

Any line beginning with a blank is assumed to contain only C text and is copied; if it precedes %% it is copied into the external definition area of the lex.yy.c file. All rules should follow a %% as in YACC. Lines preceding %% which begin with a nonblank character define the string on the left to be the remainder of the line; it can be called out later by surrounding it with {}. Note that curly brackets do not imply parentheses; only string substitution is done.

Example

```
D       [0- 9]
%%
if      printf("IF statement\n");
[a- z]+ printf("tag, value %s\n",yytext);
0{D}+   printf("octal number %s\n",yytext);
{D}+    printf("decimal number %s\n",yytext);
"++"    printf("unary op\n");
"+"     printf("binary op\n");
"/*"    {       loop:
                while (input() != '*');
                switch (input())
                        {
                        case '/': break;
                        case '*': unput('*');
                        default: go to loop;
                        }
                }
```

The external names generated by *lex* all begin with the prefix yy or YY.

The options must appear before any files. The option – c indicates C actions and is the default, – t causes the lex.yy.c program to be written instead to standard output, – v provides a one-line summary of statistics of the machine generated, – n will not print out the – summary. Multiple files are treated as a single file. If no files are specified, standard input is used.

Certain table sizes for the resulting finite state machine can be set in the definitions section:

%p *n*
: number of positions is *n* (default 2000)

%n *n*
: number of states is *n* (500)

%t *n*
: number of parse tree nodes is *n* (1000)

%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*

Name

lint – Checks C language usage and syntax.

Syntax

lint [– abchlnpuvx] file ...

Description

Lint attempts to detect features of the C program *file* that are likely to be bugs, nonportable, or wasteful. It also checks type usage more strictly than the C compiler. Among the things which are currently detected are unreachable statements, loops not entered at the top, automatic variables declared and not used, and logical expressions whose value is constant. Moreover, the usage of functions is checked to find functions which return values in some places and not in others, functions called with varying numbers of arguments, and functions whose values are not used.

If more than one *file* is given, it is assumed that all the files are to be loaded together; they are checked for mutual compatibility. If routines from the standard library are called from *file*, *lint* checks the function definitions using the standard lint library llibc.ln. If *lint* is invoked with the – p option, it checks function definitions from the portable lint library llibport.ln.

Any number of *lint* options may be used, in any order. The following options are used to suppress certain kinds of complaints:

– a Suppresses complaints about assignments of long values to variables that are not long.

– b Suppresses complaints about break statements that cannot be reached. (Programs produced by *lex* or *yacc* will often result in a large number of such complaints.)

– c Suppresses complaints about casts that have questionable portability.

– h Does not apply heuristic tests that attempt to intuit bugs, improve style, and reduce waste.

– u Suppresses complaints about functions and external variables used and not defined, or defined and not used. (This option is suitable for running *lint* on a subset of files of a larger program.)

– v Suppresses complaints about unused arguments in functions.

– x Does not report variables referred to by external declarations but never used.

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)

/usr/lib/llibc, /usr/lib/llibport, /usr/lib/llibm, /usr/lib/llibdbm, /usr/lib/llibtermlib
Standard lint libraries (source format)

/usr/tmp/*lint* Temporaries

See Also

cc(CP)

Notes

Exit(S), and other functions which do not return, are not understood. This can cause improper error messages.

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.

Name

m4 – Invokes a macro processor.

Syntax

m4 [options] [files]

Description

M4 is a macro processor intended as a front end for Ratfor, C, and other languages. Each of the argument *files* is processed in order; if there are no files, or if a filename is – , the standard input is read. The processed text is written on the standard output.

The options and their effects are as follows:

– e Operates interactively. Interrupts are ignored and the output is unbuffered.

– s Enables line sync output for the C preprocessor (#line ...)

– B*int*
Changes the size of the push-back and argument collection buffers from the default of 4,096.

– H*int*
Changes the size of the symbol table hash array from the default of 199. The size should be prime.

– S*int*
Changes the size of the call stack from the default of 100 slots. Macros take three slots, and nonmacro arguments take one.

– T*int*
Changes the size of the token buffer from the default of 512 bytes.

To be effective, these flags must appear before any filenames and before any – D or – U flags:

– D*name*[=*val*]
Defines *name* to *val* or to null in *val*'s absence.

– U*name*
Undefines *name*.

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*.

shift — Returns all but its first argument. The other arguments are quoted and pushed back with commas in between. The quoting nullifies the effect of the extra scan that will subsequently be performed.

changequote — Changes quotation marks to the first and second arguments. The symbols may be up to five characters long. *Changequote* without arguments restores the original values (i.e., ` ').

changecom — Changes left and right comment markers from the default # and newline. With no arguments, the comment mechanism is effectively disabled. With one argument, the left marker becomes the argument and the right marker becomes newline. With two arguments, both markers are affected. Comment markers may be up to five characters long.

divert — *M4* maintains 10 output streams, numbered 0-9. The final output is the concatenation of the streams in numerical order; initially stream 0 is the current stream. The *divert* macro changes the current output stream to its (digit-string) argument. Output diverted to a stream other than 0 through 9 is discarded.

undivert — Causes immediate output of text from diversions named as arguments, or all diversions if no argument. Text may be undiverted into another diversion. Undiverting discards the diverted text.

divnum — Returns the value of the current output stream.

dnl — Reads and discards characters up to and including the next newline.

ifelse — Has three or more arguments. If the first argument is the same string as the second, then the value is the third argument. If not, and if there are more than four arguments, the process is repeated with arguments 4, 5, 6 and 7. Otherwise, the value is either the fourth string, or if it is not present, null.

incr — Returns the value of its argument incremented by 1. The value of the argument is calculated by interpreting an initial digit-string as a decimal number.

decr — Returns the value of its argument decremented by 1.

eval — Evaluates its argument as an arithmetic expression, using 32-bit arithmetic. Operators include +, – , *, /, %, ^ (exponentiation), bitwise &, |, ^, and ~; relationals; parentheses. Octal and hex numbers may be specified as in C. The second argument specifies the 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*.

Name

make – Maintains, updates, and regenerates groups of programs.

Syntax

make [– f makefile] [– p] [– i] [– k] [– s] [– r] [– n] [– b] [– e] [– t] [– q] [– d] [names]

Description

The following is a brief description of all options and some special names:

– f *makefile* Description filename. *Makefile* is assumed to be the name of a description file. A filename of – denotes the standard input. The contents of *makefile* override the built-in rules if they are present.

– p Prints out the complete set of macro definitions and target descriptions.

– i Ignores error codes returned by invoked commands. This mode is entered if the fake target name **.IGNORE** appears in the description file.

– k Abandons work on the current entry, but continues on other branches that do not depend on that entry.

– s Silent mode. Does not print command lines before executing. This mode is also entered if the fake target name .SILENT appears in the description file.

– r Does not use the built-in rules.

– n No execute mode. Prints commands, but does not execute them. Even lines beginning with an **@** are printed.

– b Compatibility mode for old makefiles.

– e Environment variables override assignments within makefiles.

– t Touches the target files (causing them to be up-to-date) rather than issues the usual commands.

– d Debug mode. Prints out detailed information on files and times examined.

– 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 line is always executed (see discussion of the **MAKEFLAGS** macro under *Environment*). The – t (touch) option updates the modified date of a file without executing any commands.

Commands returning nonzero status normally terminate ***make***. If the – i option is present, or the entry .IGNORE: appears in *makefile*, or if the line specifying the command begins with <tab><hyphen>, the error is ignored. If the – k option is present, work is abandoned on the current entry, but continues on other branches that do not depend on that entry.

The – b option allows old makefiles (those written for the old version of *make*) to run without errors. The difference between the old version of *make* and this version is that this version requires all dependency lines to have a (possibly null) command associated with them. The previous version of *make* assumed if no command was specified explicitly that the command was null.

Interrupt and quit cause the target to be deleted unless the target depends on the special name .PRECIOUS.

Environment

The environment is read by *make*. All variables are assumed to be macro definitions and processed as such. The environment variables are processed before any makefile and after the internal rules; thus, macro assignments in a makefile override environment variables. The – e option causes the environment to override the macro assignments in a makefile.

The **MAKEFLAGS** environment variable is processed by *make* as containing any legal input option (except – f, – p, and – d) defined for the command line. Further, upon invocation, *make* "invents" the variable if it is not in the environment, puts the current options into it, and passes it on to invocations of commands. Thus, **MAKEFLAGS** always contains the current input options. This proves very useful for "super-makes". In fact, as noted above, when the – n option is used, the command **$(MAKE)** is executed anyway; hence, one can perform a **make – n** recursively on a whole software system to see what would have been executed. This is because the – n is put in **MAKEFLAGS** and passed to further invocations of **$(MAKE)**. This is one way of debugging all of the makefiles for a software project without actually doing anything.

Macros

Entries of the form *string1* = *string2* are macro definitions. Subsequent appearances of $(*string1*[:*subst1*=[*subst2*]]) are replaced by *string2*. The parentheses are optional if a single character macro name is used and there is no substitute sequence. The optional :*subst1*=*subst2* is a substitute sequence. If it is specified, all nonoverlapping occurrences of *subst1* in the named macro are replaced by *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 dependents such as .c, .s, etc. If no update commands for such a file appear in *makefile*, and if a default dependent exists, that prerequisite is compiled to make the target. In this case, *make* has inference rules which allow building files from other files by examining the suffixes and determining an appropriate inference rule to use. The current default inference rules are:

```
.c .c~ .sh .sh~ .c.o .c~.o .c~.c .s.o .s~.o .y.o .y~.o .l.o .l~.o
.y.c .y~.c .l.c .c.a .c~.a .s~.a .h~.h
```

The internal rules for *make* are contained in the source file **rules.c** for the *make* program. These rules can be locally modified. To print out the rules compiled into the *make* on any machine in a form suitable for recompilation, the following command is used:

```
make -fp - 2>/dev/null </dev/null
```

The only peculiarity in this output is the **(null)** string which *printf*(S) prints when handed a null string.

A tilde in the above rules refers to an SCCS file (see *sccsfile*(F)). Thus, the rule .c~.o would transform an SCCS C source file into an object file (.o). Because the **s.** of the SCCS files is a prefix it is incompatible with *make*'s suffix point-of-view. Hence, the tilde is a way of changing any file reference into an SCCS file reference.

A rule with only one suffix (i.e. .c:) is the definition of how to build *x* from *x*.c. In effect, the other suffix is null. This is useful for building targets from only one source file (e.g., shell procedures, simple C programs).

Additional suffixes are given as the dependency list for **.SUFFIXES**. Order is significant; the first possible name for which both a file and a rule exist is inferred as a prerequisite.

The default list is:

```
.SUFFIXES: .o .c .y .l .s
```

Here again, the above command for printing the internal rules will display the list of suffixes implemented on the current machine. Multiple suffix lists accumulate; **.SUFFIXES:** with no dependencies clears the list of suffixes.

Inference Rules

The first example can be done more briefly:

```
pgm: a.o b.o
    cc a.o b.o -o pgm
a.o b.o: incl.h
```

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

C source files are out of date. The substitution mode translates the .o to .c. (Unfortunately, one cannot as yet transform to .c~) Note also, the disabling of the .c.a: rule, which would have created each object file, one by one. This particular construct speeds up archive library maintenance considerably. This type of construct becomes very cumbersome if the archive library contains a mix of assembly programs and C programs.

Files

[Mm]akefile

s.[Mm]akefile

See Also

sh(C)

Notes

Some commands return nonzero status inappropriately; use **-i** to overcome the difficulty. Commands that are directly executed by the shell, notably *cd*(C), are ineffectual across newlines in *make*. The syntax (lib(file1.o file2.o file3.o) is illegal. You cannot build lib(file.o) from file.o. The macro $(a:.o=.c~) is not available.

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.

Example

```
char    efilname[] = "/usr/lib/pi_strings";
int     efil = -1;

error(a1, a2, a3, a4)
{
        char buf[256];

        if (efil < 0) {
                efil = open(efilname, 0);
                if (efil < 0) {
                        perror(efilname);
                        exit(C);
                }
        }
        if (lseek(efil, (long) a1, 0) || read(efil, buf, 256) <= 0)
                goto oops;
        printf(buf, a2, a3, a4);
}
```

See Also

lseek(S), xstr(CP)

Credit

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

Notes

All the arguments except the name of the file to be processed are unnecessary.

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

prof – display profile data

Syntax

prof [**−a**] [**−l**] [*−low* [*−high*]] [file]

Description

Prof interprets the file *mon.out* produced by the *monitor* subroutine. Under default modes, the symbol table in the named object file (*x.out* default) is read and correlated with the *mon.out* profile file. For each external symbol, the percentage of time spent executing between that symbol and the next is printed (in decreasing order), together with the number of times that routine was called and the number of milliseconds per call.

If the **−a** option is used, all symbols are reported rather than just external symbols. If the **−l** option is used, the output is listed by symbol value rather than decreasing percentage.

If the **−v** option is used, all printing is suppressed and a graphic version of the profile is produced on the standard output for display by the *plot*(C) filters. The numbers *low* and *high*, by default 0 and 100, cause a selected percentage of the profile to be plotted with accordingly higher resolution.

In order for the number of calls to a routine to be tallied, the **−p** option of *cc* must have been given when the file containing the routine was compiled. This option also arranges for the *mon.out* file to be produced automatically.

Files

mon.out for profile
x.out for namelist

See Also

monitor(S), profil(S), cc(CP) , plot(C)

Notes

Beware of quantization errors.

If you use an explicit call to *monitor*(S) you will need to make sure that the buffer size is equal to or smaller than the program size.

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.

Data Keywords

Data keywords specify which parts of an SCCS file are to be retrieved and output. All parts of an SCCS file (see *sccsfile*(F)) have an associated data keyword. There is no limit on the number of times a data keyword may appear in a *dataspec.*

The information printed by *prs* consists of the user-supplied text and appropriate values (extracted from the SCCS file) substituted for the recognized data keywords in the order of appearance in the *dataspec.* The format of a data keyword value is either simple, in which keyword substitution is direct, or multiline, in which keyword substitution is followed by a carriage return.

User-supplied text is any text other than recognized data keywords. A tab is specified by \t and carriage return/newline is specified by \n.

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:

Examples

The following:

```
prs - d"Users and/or user IDs for :F: are:\n:UN:" s.file
```

may produce on the standard output:

```
Users and/or user IDs for s.file are:
xyz
131
abc
```

```
prs - d"Newest delta for pgm :M:: :I: Created :D: By :P:" - r
s.file
```

may produce on the standard output:

```
Newest delta for pgm main.c: 3.7 Created 77/12/1 By cas
```

As a *special case:*

```
prs s.file
```

may produce on the standard output:

```
D 1.1 77/12/1 00:00:00 cas 1 000000/00000/00000
MRs:
bl78-12345
bl79-54321
COMMENTS:
this is the comment line for s.file initial delta
```

for each delta table entry of the "D" type. The only option allowed to be used with the *special case* is the – a option.

Files

/tmp/pr?????

See Also

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

Diagnostics

Use *help*(CP) for explanations.

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.

Name

ratfor – Converts Rational FORTRAN into standard FORTRAN.

Syntax

ratfor [option ...] [filename ...]

Description

Ratfor converts a rational dialect of FORTRAN into ordinary irrational FORTRAN. *Ratfor* provides control flow constructs essentially identical to those in C:

```
statement grouping:
     { statement; statement; statement }

decision-making:
     if (condition) statement [ else statement ]
     switch (integer value) {
               case integer:    statement
               ...
               [ default: ]     statement
     }

loops:
     while (condition) statement
     for (expression; condition; expression) statement
     do limits statement
     repeat statement [ until (condition) ]
     break [n]
     next [n]
```

and some additional syntax to make programs easier to read and write:

```
Free form input:
     multiple statements/line; automatic continuation

Comments:
     # this is a comment

Translation of relationals:
     >, >=, etc., become .GT., .GE., etc.

Return (expression)
     returns expression to caller from function

Define:
     define name replacement
```

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

regcmp – Compiles regular expressions.

Syntax

regcmp [–] files

Description

Regcmp, in most cases, precludes the need for calling *regcmp* (see *regex*(S)) from C programs. This saves on both execution time and program size. The command *regcmp* compiles the regular expressions in file and places the output in file .i. If the – option is used, the output will be placed in file .c. The format of entries in *file* is a name (C variable) followed by one or more blanks followed by a regular expression enclosed in double quotation marks. The output of *regcmp* is C source code. Compiled regular expressions are represented as extern char vectors. *File*.i files may thus be *included* into C programs, or *file*.c files may be compiled and later loaded. In the C program which uses the *regcmp* output, *regex*(*abc*,*line*) applies the regular expression named *abc* to *line*. Diagnostics are self-explanatory.

Examples

```
name    "([A- Za- z][A- Za- z0- 9_]*)$0"

telno   "\({0,1}([2- 9][01][1- 9])$0\){0,1} *"
        "([2- 9][0- 9]{2})$1[ - ]{0,1}"
        "([0- 9]{4})$2"
```

In the C program that uses the *regcmp* output,

```
regex(telno, line, area, exch, rest)
```

will apply the regular expression named *telno* to *line*.

See Also

regex(S)

Name

rmdel – Removes a delta from an SCCS file.

Syntax

rmdel – rSID 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

sact – Prints current SCCS file editing activity.

Syntax

sact files

Description

Sact informs the user of any impending deltas to a named SCCS file. This situation occurs when *get*(CP) with the – e option has been previously executed without a subsequent execution of *delta*(CP). If a directory is named on the command line, *sact* behaves as though each file in the directory were specified as a named file, except that nonSCCS files and unreadable files are silently ignored. If a name of – is given, the standard input is read with each line being taken as the name of an SCCS file to be processed.

The output for each named file consists of five fields separated by spaces.

Field 1	Specifies the SID of a delta that currently exists in the SCCS file to which changes will be made to make the new delta
Field 2	Specifies the SID for the new delta to be created
Field 3	Contains the logname of the user who will make the delta i.e., executed a *get* for editing
Field 4	Contains the date that get – e was executed
Field 5	Contains the time that get – e was executed

See Also

delta(CP), get(CP), unget(CP)

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

size – Prints the size of an object file.

Syntax

size [object ...]

Description

Size prints the (decimal) number of bytes required by the text, data, and bss portions, and their sum in decimal and hexadecimal, of each object-file argument. If no file is specified, a.out is used.

See Also

a.out(F)

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

$$y_0'' = ky_1' \ , \ . \ . \ . \ , \ y_n'' = ky_{n-1}'$$

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

strings – Finds the printable strings in an object file.

Syntax

strings [–] [– o] [– *number*] file ...

Description

Strings looks for ASCII strings in a binary file. A string is any sequence of four or more printing characters ending with a newline or a null character. Unless the – flag is given, *strings* only looks in the initialized data space of object files. If the – o flag is given, then each string is preceded by its decimal offset in the file. If the – *number* flag is given then *number* is used as the minimum string length rather than 4.

Strings is useful for identifying random object files and many other things.

See Also

hd(C), od(C)

Credit

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

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

time – Times a command.

Syntax

time command

Description

The given *command* is executed; after it is complete, *time* prints the elapsed time during the command, the time spent in the system, and the time spent in execution of the command. Times are reported in seconds.

The times are printed on the standard error.

See Also

times(S)

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

unget – Undoes a previous get of an SCCS file.

Syntax

unget [– rSID] [– s] [– n] files

Description

Unget undoes the effect of a get – e done prior to creating the intended new delta. If a directory is named, *unget* behaves as though each file in the directory were specified as a named file, except that nonSCCS files and unreadable files are silently ignored. If a name of – is given, the standard input is read with each line being taken as the name of an SCCS file to be processed.

Options apply independently to each named file.

– r*SID* Uniquely identifies which delta is no longer intended. (This would have been specified by *get* as the "new delta".) The use of this option is necessary only if two or more versions of the same SCCS file have been retrieved for editing by the same person (login name). A diagnostic results if the specified *SID* is ambiguous, or if it is necessary and omitted on the command line.

– s Suppresses the printout, on the standard output, of the intended delta's *SID*.

– n Causes the retention of the file which would normally be removed from the current directory.

See Also

delta(CP), get(CP), sact(CP)

Diagnostics

Use *help*(CP) for explanations.

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*.

– y*type* The argument value *type* is compared with the SCCS %Y% keyword in *file*.

The 8-bit code returned by *val* is a disjunction of the possible errors, i. e., can be interpreted as a bit string where (moving from left to right) set bits are interpreted as follows:

bit 0 = Missing file argument

bit 1 = Unknown or duplicate option

bit 2 = Corrupted SCCS file

bit 3 = Can't open file or file not SCCS

bit 4 = *SID* is invalid or ambiguous

bit 5 = *SID* does not exist

bit 6 = %Y%, - y mismatch

bit 7 = %M%, - m mismatch

Note that *val* can process two or more files on a given command line and in turn can process multiple command line (when reading the standard input). In these cases an aggregate code is returned; a logical OR of the codes generated for each command line and file processed.

See Also

admin(CP), delta(CP), get(CP), prs(CP)

Diagnostics

Use *help*(CP) for explanations.

Notes

Val can process up to 50 files on a single command line.

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)

Name

xstr - Extracts strings from C programs.

Syntax

xstr [- c] [-] [file]

Description

Xstr maintains a file *strings* into which strings in component parts of a large program are hashed. These strings are replaced with references to this common area. This serves to implement shared constant strings, most useful if they are also read-only.

The command

```
xstr - c name
```

will extract the strings from the C source in name, replacing string references by expressions of the form (&xstr[number]) for some number. An appropriate declaration of *xstr* is prepended to the file. The resulting C text is placed in the file *x.c*, to then be compiled. The strings from this file are placed in the *strings* data base if they are not there already. Repeated strings and strings which are suffices of existing strings do not cause changes to the data base.

After all components of a large program have been compiled, a file *xs.c* declaring the common *xstr* space can be created by a command of the form

```
xstr -c name1 name2 name3 ...
```

This *xs.c* file should then be compiled and loaded with the rest of the program. If possible, the array can be made read-only (shared) saving space and swap overhead.

Xstr can also be used on a single file. A command

```
xstr name
```

creates files *x.c* and *xs.c* as before, without using or affecting any *strings* file in the same directory.

It may be useful to run *xstr* after the C preprocessor if any macro definitions yield strings or if there is conditional code which contains strings which may not, in fact, be needed. *Xstr* reads from its standard input when the argument - is given. An appropriate command sequence for running *xstr* after the C preprocessor is:

```
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.

Name

yacc – Invokes a compiler-compiler.

Syntax

yacc [– vd] grammar

Description

Yacc converts a context-free grammar into a set of tables for a simple automaton which executes an LR(1) parsing algorithm. The grammar may be ambiguous; specified precedence rules are used to break ambiguities.

The output file, y.tab.c, must be compiled by the C compiler to produce a program *yyparse*. This program must be loaded with the lexical analyzer program, *yylex*, as well as *main* and *yyerror*, an error handling routine. These routines must be supplied by the user; *lex*(CP) is useful for creating lexical analyzers usable by *yacc*.

If the – **v** flag is given, the file y.output is prepared, which contains a description of the parsing tables and a report on conflicts generated by ambiguities in the grammar.

If the – d flag is used, the file y.tab.h is generated with the #define statements that associate the *yacc*-assigned "token codes" with the user-declared "token names". This allows source files other than y.tab.c to access the token codes.

Files

y.output	
y.tab.c	
y.tab.h	Defines for token names
yacc.tmp, yacc.acts	Temporary files
/usr/lib/yaccpar	Parser prototype for C programs

See Also

lex(CP)

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)

defopen, defread	Reads default entries
dup, dup2	Duplicates an open file descriptor
ecvt, fcvt	Performs output conversions
execl, execv, execle, execve, execlp, execvp	Executes a file
exit	Terminates a process
exp, log, pow, sqrt	Performs exponential, logarithm, power, square root functions
fclose, fflush	Closes or flushes a stream
fcntl	Controls open files
ferror, feof, clearerr, fileno,	Determines stream status
floor, fabs, ceil, fmod	Performs absolute value, floor, ceiling, and remainder functions
fopen, freopen, fdopen	Opens a stream
fork	Creates a new process
fread, fwrite	Performs buffered binary input and output
frexp, ldexp, modf	Splits floating–point number into a mantissa and an exponent
fseek, ftell, rewind	Repositions a stream
gamma	Performs log gamma functions
getc, getchar, fgetc, getw	Gets character or word from a stream
getcwd	Gets pathname of current working directory
getenv	Gets value for environment name
getgrent, getgrgid, getgrnam, setgrent, endgrent	Get group file entry
getlogin	Gets login name
getopt	Gets option letter from argument vector
getpass	Reads a password
getpid, getpgrp, getppid	Gets process, process group, and parent process IDs
getpw	Gets name from UID
getpwent, getpwuid, getpwnam, setpwent, endpwent	Gets password file entry
gets, fgets	Gets a string from a stream
getuid, geteuid, getgid, getegid	Gets real user, effective user, real group and effective group IDs
hypot	Determines Euclidean distance
ioctl	Controls character devices
kill	Sends a signal to a process or a group of processes

i3tol, ltol3	Converts between 3-byte integers and long integers
link	Links a file to an existing file
lock	Locks a process in primary memory
locking	Locks a file region for reading or writing
logname	Finds login name of user
lsearch	Performs linear search and update
lseek	Moves read/write file pointer
malloc, free, realloc, calloc	Allocates main memory
mknod	Makes a directory, or a special or ordinary file
mktemp	Makes a unique filename
monitor	Prepares execution profile
mount	Mounts a file system
nap	Suspends execution for a short interval
nice	Changes priority of a process
nlist	Gets entries from name list
open	Opens file for reading or writing
opensem	Opens a semaphore
pause	Suspends a process until a signal occurs
perror, sys_errlist, sys_nerr, errno	Sends system error messages
pipe	Creates an interprocess channel
popen, pclose	Initiates I/O to or from a process
printf, fprintf, sprintf	Formats output
profil	Creates an execution time profile
ptrace	Traces a process
putc, putchar, fputc, putw	Puts a character or word on a stream
putpwent	Writes a file password entry
puts, fputs	Puts a string on a stream
qsort	Performs a sort
rand, srand	Generates a random number
rdchk	Checks to see if there is data to be read
read	Reads from a file
regex, regcmp	Compiles and executes regular expressions
regexp	Performs regular expression compile and match functions
sbrk	Changes data segment space allocation
scanf, fscanf, sscanf	Converts and formats input
sdenter, sdleave	Synchronizes access to a shared data segment
sdget	Attaches and detaches a shared data segment

sdgetv, sdwaitv	Synchronizes shared data access
setbuf	Assigns buffering to a stream
setjmp, longjmp	Performs a nonlocal "goto"
setpgrp	Sets process group ID
setuid, setgid	Sets user and group IDs
shutdn	Flushes block I/O and halts the CPU
signal	Specifies what to do upon receipt of a signal
sigsem	Signals a process waiting on a semaphore
sinh, cosh, tanh	Peforms hyperbolic functions
sleep	Suspends execution for an interval
ssignal, gsignal	Implements software signals
stat, fstat	Gets file status
stdio	Performs standard buffered input and output
stime	Sets the time
string, strcat, strncat, strcmp, strncmp, strcpy, strncpy, strlen, strchr, strrchr, strpbrk, strspn, strcspn, strtok	Performs string operations
swab	Swaps bytes
sync	Updates the super–block
system	Executes a shell command
termcap, tgetent, tgetnum, tgetflag, tgetstr, tgoto, tputs	Performs terminal functions
time, ftime	Gets time and date
times	Gets process and child process times
tmpfile	Creates a temporary file
tmpnam	Creates a name for a temporary file
trig, sin, cos, tan, asin, acos, atan, atan2	Performs trigonometric functions
ttyname, isatty	Finds the name of a terminal
ulimit	Gets and sets user limits
umask	Sets and gets file creation mask
umount	Unmounts a file system
uname	Gets name of current XENIX system
ungetc	Pushes character back into input stream
unlink	Removes directory entry
ustat	Gets files system statistics
utime	Sets file access and

	modification times
wait	Waits for a child process to stop or terminate
waitsem, nbwaitsem	Awaits and checks access to a resource goverened by a semaphore
write	Writes to a file
xlist, fxlist	Gets namelist entries from files

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>.

1 EPERM Not owner
Typically this error indicates an attempt to modify a file in some way forbidden except to its owner or super-user. It is also returned for attempts by ordinary users to do things allowed only to the super-user.

2 ENOENT No such file or directory
This error occurs when a filename is specified and the file should exist but doesn't, or when one of the directories in a pathname does not exist.

3 ESRCH No such process
No process can be found corresponding to that specified by *pid* in *kill* or *ptrace*.

4 EINTR Interrupted system call
An asynchronous signal (such as interrupt or quit), which the user has elected to catch, occurred during a system call. If execution is resumed after processing the signal, it will appear as if the interrupted system call returned this error condition.

5 EIO I/O error
Some physical I/O error. This error may in some cases occur on a call following the one to which it actually applies.

6 ENXIO No such device or address
I/O on a special file refers to a subdevice which does not exist, or beyond the limits of the device. It may also occur when, for example, a tape drive is not on-line or no disk pack is loaded on a drive.

7 E2BIG Arg list too long
An argument list longer than 5,120 bytes is presented to a member of the *exec* family.

8 ENOEXEC Exec format error
A request is made to execute a file which, although it has the appropriate permissions, does not start with a valid magic number (see *a.out*(F)).

9 EBADF Bad file number
Either a file descriptor refers to no open file, or a read (respectively write) request is made to a file which is open only for writing (respectively reading).

10 ECHILD No child processes
A *wait*, was executed by a process that had no existing or unwaited-for child processes.

11 EAGAIN No more processes
A *fork*, failed because the system's process table is full or the user is not allowed to create any more processes.

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.

23 ENFILE File table overflow
The system's table of open files is full, and temporarily no more *opens* can be accepted.

24 EMFILE Too many open files
No process may have more than 20 file descriptors open at a time.

25 ENOTTY Not a typewriter

26 ETXTBSY Text file busy
An attempt to execute a pure-procedure program which is currently open for writing (or reading). Also an attempt to open for writing a pure-procedure program that is being executed.

27 EFBIG File too large
The size of a file exceeded the maximum file size (1,082,201,088 bytes) or ULIMIT; see *ulimit*(S).

28 ENOSPC No space left on device
During a *write* to an ordinary file, there is no free space left on the device.

29 ESPIPE Illegal seek
An *lseek* was issued to a pipe.

30 EROFS Read-only file system
An attempt to modify a file or directory was made on a device mounted read-only.

31 EMLINK Too many links
An attempt to make more than the maximum number of links (1000) to a file.

32 EPIPE Broken pipe
A write on a pipe for which there is no process to read the data. This condition normally generates a signal; the error is returned if the signal is ignored.

33 EDOM Math arg out of domain of func
The argument of a function in the math package is out of the domain of the function.

34 ERANGE Math result not representable
The value of a function in the math package is not representable within machine precision.

35 EUCLEAN File system needs cleaning
An attempt was made to *mount*(S) a file system whose super-block is not flagged clean.

36 EDEADLOCK Would deadlock
A process' attempt to lock a file region would cause a deadlock

between processes vying for control of that region.

37 ENOTNAM Not a name file
A *createsem*(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 *createsem*(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).

Real User ID and Real Group ID

Each user allowed on the system is identified by a positive integer called a real user ID.

Each user is also a member of a group. The group is identified by a positive integer called the real group ID.

An active process has a real user ID and real group ID that are set to the real user ID and real group ID, respectively, of the user responsible for the creation of the process.

Effective User ID and Effective Group ID

An active process has an effective user ID and an effective group ID that are used to determine file access permissions (see below). The effective user ID and effective group ID are equal to the process' real user ID and real group ID respectively, unless the process or one of its ancestors evolved from a file that had the set-user-ID bit or set-group ID bit set; see *exec*(S).

Super-User

A process is recognized as a *super-user* process and is granted special privileges if its effective user ID is 0.

Special Processes

The processes with a process ID of 0 and a process ID of 1 are special processes and are referred to as *proc0* and *proc1*.

Proc0 is the scheduler. *Proc1* is the initialization process (*init*). Proc1 is the ancestor of every other process in the system and is used to control the process structure.

Filename

Names consisting of up to 14 characters may be used to name an ordinary file, special file or directory.

These characters may be selected from the set of all character values excluding 0 (null) and the ASCII code for a / (slash).

Note that it is generally unwise to use *, ?, [, or] as part of filenames because of the special meaning attached to these characters by the shell. Likewise, the high order bit of the character should not be set.

ıl User ID and Real Group ID

.ch user allowed on the system is identified by a positive integer .lled a real user ID.

Each user is also a member of a group. The group is identified by a positive integer called the real group ID.

An active process has a real user ID and real group ID that are set to the real user ID and real group ID, respectively, of the user responsible for the creation of the process.

Effective User ID and Effective Group ID

An active process has an effective user ID and an effective group ID that are used to determine file access permissions (see below). The effective user ID and effective group ID are equal to the process' real user ID and real group ID respectively, unless the process or one of its ancestors evolved from a file that had the set-user-ID bit or set-group ID bit set; see *exec*(S).

Super-User

A process is recognized as a *super-user* process and is granted special privileges if its effective user ID is 0.

Special Processes

The processes with a process ID of 0 and a process ID of 1 are special processes and are referred to as *proc0* and *proc1*.

Proc0 is the scheduler. *Proc1* is the initialization process (*init*). Proc1 is the ancestor of every other process in the system and is used to control the process structure.

Filename

Names consisting of up to 14 characters may be used to name an ordinary file, special file or directory.

These characters may be selected from the set of all character values excluding 0 (null) and the ASCII code for a / (slash).

Note that it is generally unwise to use *, ?, [, or] as part of filenames because of the special meaning attached to these characters by the shell. Likewise, the high order bit of the character should not be set.

-propriate access bit of

INTRO(S)

'en processes vying for control of that region.

AM Not a name file
sem(S), opensem(S), waitsem(S), or sigsem(S) was issued
invalid semaphore identifier.

Not available
m(S), waitsem(S) or sigsem(S) was issued to a sema-
has not been initialized by a call to createsem(S). A
issued to a semaphore out of sequence; i.e., before
has issued the corresponding waitsem to the sema-
nbwaitsem was issued to a semaphore guarding a
is currently in use by another process. The sema-
h a process was waiting has been left in an incon-
hen the process controlling the semaphore exits
ishing control properly; i.e., without issuing a
emaphore.

file

phore, shared data, etc.) was specified when

stem is uniquely identified by a positive
he range of this ID is from 0 to 30,000.

urrently active process; see *fork*(S).
is the process ID of its creator.

process group that is identified
group ID. This ID is the pro-
uping permits the signaling of

a terminal group that is
group ID. This grouping
ess upon termination of
and *signal*(S).

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

abort – Generates an IOT fault.

Syntax

abort ( )

Description

Abort causes an I/O trap signal (SIGIOT) to be sent to the calling process. This usually results in termination with a core dump.

Abort can return control if the calling process is set to catch or ignore the SIGIOT signal; see *signal*(S).

See Also

adb(CP), exit(S), signal(S)

Diagnostics

If an aborted process returns control to the shell (*sh*(C)), the shell usually displays the message "abort – core dumped".

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.

Name

access – Determines accessibility of a file.

Syntax

```
int access (path, amode)
char *path;
int amode;
```

Description

Path points to a pathname naming a file. *Access* checks the named file for accessibility according to the bit pattern contained in *amode*, using the real user ID in place of the effective user ID and the real group ID in place of the effective group ID. The bit pattern for *amode* can be formed by adding any combination of the following:

04	Read
02	Write
01	Execute (search)
00	Check existence of file

Access to the file is denied if one or more of the following are true:

A component of the path prefix is not a directory. [ENOTDIR]

Read, write, or execute (search) permission is requested for a null pathname. [ENOENT]

The named file does not exist. [ENOENT]

Search permission is denied on a component of the path prefix. [EACCES]

Write access is requested for a file on a read-only file system. [EROFS]

Write access is requested for a pure procedure (shared text) file that is being executed. [ETXTBSY]

Permission bits of the file mode do not permit the requested access. [EACCES]

Path points outside the process' allocated address space. [EFAULT]

Access checks the permissions for the owner of a file by checking the "owner" read, write, and execute mode bits. For members of the file's group, the "group" mode bits are checked. For all others, the "other" mode bits are checked.

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.

Name

acct – Enables or disables process accounting.

Syntax

```
int acct (path)
char *path;
```

Description

Acct is used to enable or disable the system's process accounting routine. If the routine is enabled, an accounting record will be written on an accounting file for each process that terminates. A process can be terminated by a call to *exit* or by receipt of a signal which it does not ignore or catch; see *exit*(S) and *signal*(S). The effective user ID of the calling process must be super-user to use this call.

Path points to the pathname of the accounting file. The accounting file format is given in *acct*(F).

The accounting routine is enabled if *path* is nonzero and no errors occur during the system call. It is disabled if *path* is zero and no errors occur during the system call.

Acct will fail if one or more of the following are true:

The effective user ID of the calling process is not super-user. [EPERM]

An attempt is being made to enable accounting when it is already enabled. [EBUSY]

A component of the path prefix is not a directory. [ENOTDIR]

One or more components of the accounting file's pathname do not exist. [ENOENT]

A component of the path prefix denies search permission. [EACCES]

The file named by *path* is not an ordinary file. [EACCES]

Mode permission is denied for the named accounting file. [EACCES]

The named file is a directory. [EACCES]

The named file resides on a read-only file system. [EROFS]

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

alarm - Sets a process' alarm clock.

Syntax

```
unsigned alarm (sec)
unsigned sec;
```

Description

Alarm sets the calling process' alarm clock to *sec* seconds. After *sec* "real-time" seconds have elasped, the alarm clock sends a SIGALRM signal to the process; see *signal*(S).

Although *alarm* does not wait for the signal after setting the alarm clock, *pause*(S) may be used to make the calling process wait.

Alarm requests are not stacked; successive calls reset the calling process' alarm clock.

If *sec* is 0, any previously made alarm request is canceled.

Return Value

Alarm returns the amount of time previously remaining in the calling process' alarm clock.

See Also

pause(S), signal(S)

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

atof, atoi, atol – Converts ASCII to numbers.

Syntax

```
double atof (nptr)
char *nptr;

int atoi (nptr)
char *nptr;

long atol (nptr)
char *nptr;
```

Description

These functions convert a string pointed to by *nptr* to floating, integer, and long integer numbers respectively. The first unrecognized character ends the string.

Atof recognizes a string of the form:

[+ | –] digits[. digits][e| E [+ | –] digits]

where the digits are continguous decimal digits. Any number of tabs and spaces may precede the string. The + and – signs are optional. Either e or E may be used to mark the beginning of the exponent.

Atoi and *atol* recognize strings of the form:

[+ | –] digits

where the digits are contiguous decimal digits. Any number of tabs and spaces may precede the string. The + and – signs are optional.

See Also

scanf(S)

Notes

There are no provisions for overflow.

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

bsearch – Performs a binary search.

Syntax

```
char *bsearch (key, base, nel, width, compar)
char *key;
char *base;
int nel, width;
int (*compar)( );
```

Description

Bsearch is a binary search routine generalized from Knuth (6.2.1) Algorithm B. It returns a pointer into a table indicating the location at which a datum may be found. The table must be previously sorted in increasing order. 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 is the number of elements in the table. The fourth 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 an integer less than, equal to, or greater than 0, depending on whether the first argument is to be considered less than, equal to, or greater than the second.

Return Value

If the key cannot be found in the table, a value of 0 is returned.

See Also

lsearch(S), qsort(S)

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)

Name

chmod – Changes mode of a file.

Syntax

```
int chmod (path, mode)
char *path;
int mode;
```

Description

Path points to a pathname naming a file. *Chmod* sets the access permission portion of the named file's mode according to the bit pattern contained in *mode*.

Access permission bits for *mode* can be formed by adding any combination of the following:

04000 Set user ID on execution
02000 Set group ID on execution
01000 Save text image after execution
00400 Read by owner
00200 Write by owner
00100 Execute (or search if a directory) by owner
00040 Read by group
00020 Write by group
00010 Execute (or search) by group
00004 Read by others
00002 Write by others
00001 Execute (or search) by others

To change the mode of a file, the effective user ID of the process must match the owner of the file or must be super-user.

If the effective user ID of the process is not super-user, mode bit 01000 (save text image on execution) is cleared.

If the effective user ID of the process is not super-user or the effective group ID of the process does not match the group ID of the file, mode bit 02000 (set group ID on execution) is cleared.

If an executable file is prepared for sharing, then mode bit 01000 prevents the system from abandoning the swap-space image of the program-text portion of the file when its last user terminates. Thus, when the next user executes the file, the text need not be read from the file system but can simply be swapped in, saving time. Many systems have relatively small amounts of swap space, and the same-text bit should be used sparingly, if at all.

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]

e effective user ID does not match the owner of the file and effective user ID is not super-user. [EPERM]

med file resides on a read-only file system. [EROFS]

'nts outside the process' allocated address space.

wise, a

letion, a value of 0 is returned. Otherwise, a and *errno* is set to indicate the error.

Page 1

Name

chown – Changes the owner and group of a file.

Syntax

```
int chown (path, owner, group)
char *path;
int owner, group;
```

Description

Path points to a pathname naming a file. The owner ID and group ID of the named file are set to the numeric values contained in *owner* and *group* respectively.

Only processes with an effective user ID equal to the file owner or super-user may change the ownership of a file.

If *chown* is invoked by other than the super-user, the set-user-ID and set-group-ID bits of the file mode, 04000 and 02000 respectively, will be cleared.

Chown will fail and the owner and group of the named file will remain 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

chmod(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

chsize – Changes the size of a file.

Syntax

```
int chsize (fildes, size)
int fildes;
long size;
```

Description

Fildes is a file descriptor obtained from a *creat*, *open*, *dup*, *fcntl*, or *pipe* system call. *Chsize* changes the size of the file associated with the file descriptor *fildes* to be exactly *size* bytes in length. The routine either truncates the file, or pads it with an appropriate number of bytes. If *size* is less than the initial size of the file, then all allocated disk blocks between *size* and the initial file size are freed.

The maximum file size as set by *ulimit*(S) is enforced when *chsize* is called, rather than on subsequent writes. Thus *chsize* fails, and the file size remains unchanged if the new changed file size would exceed the *ulimit*.

Return Value

Upon successful completion, a value of 0 is returned. Otherwise, the value – 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

In general if *chsize* is used to expand the size of a file, when data is written to the end of the file, intervening blocks are filled with zeros. In a few rare cases, reducing the file size may not remove the data beyond the new end-of-file.

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)

Name

conv, toupper, tolower, toascii – Translates characters.

Syntax

```
#include <ctype.h>

int toupper (c)
int c;

int tolower (c)
int c;

int _toupper (c)
int c;

int _tolower (c)
int c;

int toascii (c)
int c;
```

Description

Toupper and *tolower* convert the argument *c* to a letter of opposite case. Arguments may be the integers – 1 through 255 (the same values returned by *getc*(S)). If the argument of *toupper* represents a lowercase letter, the result is the corresponding uppercase letter. If the argument of *tolower* represents an uppercase letter, the result is the corresponding lowercase letter. All other arguments are returned unchanged.

_toupper and *_tolower* are macros that accomplish the same thing as *toupper* and *tolower* but have restricted argument values and are faster. *_toupper* requires a lowercase letter as its argument; its result is the corresponding uppercase letter. *_tolower* requires an uppercase letter as its argument; its result is the corresponding lowercase letter. All other arguments cause unpredictable results.

Toascii converts integer values to ASCII characters. The function clears all bits of the integer that are not part of a standard ASCII character; it is intended for compatibility with other systems.

See Also

ctype(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.

Closes a file descriptor.

ildes)

descriptor obtained from a *creat*, *open*, *dup*, *fcntl*, or
l. *Close* closes the file descriptor indicated by *fildes*.

fildes is not a valid open file descriptor. [EBADF]

ompletion, a value of 0 is returned. Otherwise, a
rned and *errno* is set to indicate the error.

c(S), fcntl(S), open(S), pipe(S)

l
e
s'
er
ay
en

ared.

. See

the file
n if the
e begin-
ross exec
n 20 files
mode that

ENOTDIR]

IOENT]

he path prefix.

ich the file is to

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*.

Name

creatsem – Creates an instance of a binary semaphore.

Syntax

```
sem_num = creatsem(sem_name,mode);
int sem_num,mode
char *sem_name;
```

Description

Creatsem defines a binary semaphore named by *sem_name* to be used by *waitsem*(S) and *sigsem*(S) to manage mutually exclusive access to a resource, shared variable, or critical section of a program. *Creatsem* returns a unique semaphore number *sem_num* which may then be used as the parameter in *waitsem* and *sigsem* calls. Semaphores are special files of 0 length. The filename space is used to provide unique identifiers for semaphores. *Mode* sets the accessibility of the semaphore using the same format as file access bits. Access to a semaphore is granted only on the basis of the read access bit; the write and execute bits are ignored.

A semaphore can be operated on only by a synchronizing primitive, such as *waitsem* or *sigsem*, by *creatsem* which initializes it to some value, or by *opensem* which opens the semaphore for use by a process. Synchronizing primitives are guaranteed to be executed without interruption once started. These primitives are used by associating a semaphore with each resource (including critical code sections) to be protected.

The process controlling the semaphore should issue

```
sem_num = creatsem("semaphore", mode);
```

to create, initialize, and open the semaphore for that process. All other processes using the semaphore should issue

```
sem_num = opensem("semaphore")
```

to access the semaphore's identification value. Note that a process cannot open and use a semaphore that has not been initialized by a call to *creatsem*, nor should a process open a semaphore more than once in one period of execution. Both the creating and opening processes use *waitsem* and *sigsem* to use the semaphore *sem_num*.

See Also

opensem(S), waitsem(S), sigsem(S).

Diagnostics

Creatsem 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 *creatsem*, *errno* is set to ENAVAIL.

Notes

After a *creatsem* you must do a *waitsem* to gain control of a given resource.

Name

crypt, setkey, encrypt – Performs encryption functions.

Syntax

```
char *crypt (key, salt)
char *key, *salt;

setkey (key)
char *key;

encrypt (block, edflag)
char *block;
int edflag;
```

Description

Crypt is the password encryption routine. It is based on the NBS Data Encryption Standard (DES), with variations intended (among other things) to frustrate use of hardware implementations of the DES for key search.

The first argument to *crypt* is a user's typed password. The second is a 2-character string chosen from the set [a-zA-Z0-9./]; this *salt* string is used to perturb the DES algorithm in one of 4096 different ways, after which the password is used as the key to encrypt repeatedly a constant string. The returned value points to the encrypted password, in the same alphabet as the *salt*. The first two characters are the *salt* itself.

The *setkey* and *encrypt* entries provide access to the actual DES algorithm. The argument of *setkey* is a character array of length 64 containing only the characters with numerical value 0 and 1. If this string is divided into groups of 8, the low-order bit in each group is ignored, leading to a 56-bit key which is set into the machine.

The argument to the *encrypt* entry is likewise a character array of length 64 containing zeroes and ones. The argument array is modified in place to a similar array representing the bits of the argument after having been subjected to the DES algorithm using the key set by *setkey*. If *edflag* is 0, the argument is encrypted; if nonzero, it is decrypted.

See Also

passwd(C), getpass(S), passwd(M)

Notes

The return value from *crypt* points to static data that is overwritten by each call.

Name

ctermid – Generates a filename for a terminal.

Syntax

```
#include <stdio.h>

char *ctermid(s)
char *s;
```

Description

Ctermid returns a pointer to a string that, when used used as a filename, refers to the controlling terminal of the calling process.

If (int)*s* is zero, the string is stored in an internal static area, the contents of which are overwritten at the next call to *ctermid*, and the address of which is returned. If (int)*s* is nonzero, then *s* is assumed to point to a character array of at least L_ctermid elements; the string is placed in this array and the value of *s* is returned. The manifest constant L_ctermid is defined in <stdio.h>.

Notes

The difference between *ctermid* and *ttyname*(S) is that *ttyname* must be given a file descriptor and it returns the actual name of the terminal associated with that file descriptor, while *ctermid* returns a magic string (/dev/tty) that will refer to the terminal if used as a filename. Thus *ttyname* is useless unless the process already has at least one file open to a terminal.

See Also

ttyname(S)

'ne, asctime, tzset – Converts date and time

s to static data that is overwritten

CRYPT(S)

ime pointed to by *clock* (such as returned by
II and returns a pointer to a 26-character string in
m:

16 01:03:52 1973\n\0

ry, fields in this string are padded with spaces to keep the
constant length.

ime and *gmtime* return pointers to structures containing the
as a variety of individual quantities. These quantities give the
le on a 24-hour clock, day of month (1-31), month of year (0-
1), 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.

The structure declaration for *tm* is defined in /usr/include/time.h.

The external long variable *timezone* contains the difference, in seconds, between GMT and local standard time (e.g., in Eastern Standard Time (EST), *timezone* is 5*60*60); the external integer variable *daylight* is nonzero if and only if the standard U.S.A. Daylight Savings Time conversion should be applied. The program knows about the peculiarities of this conversion in 1974 and 1975.

If an environment variable named TZ is present, *asctime* uses the contents of the variable to override the default time zone. The value of TZ must be a three-letter time zone name, followed by a number representing the difference between local time (with optional sign) and Greenwich time in hours, followed by an optional three-letter name for a daylight time zone. For example, the setting for New Jersey would be EST5EDT. The effects of setting TZ are thus to change the values of the external variables *timezone* and *daylight*. In addition, the time zone names contained in the external variable

```
char *tzname[2] = {"EST", "EDT"};
```

are set from the environment variable. The function *tzset* sets the external variables from TZ ; it is called by *asctime* and may also be called explicitly by the user.

See Also

time(S), getenv(S), environ(M)

Notes

The return values point to static data those content is overwritten by each call.

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

See Also

ascii(M)

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

nl()	Sets newline mapping
nocrmode()	Unsets cbreak mode
noecho()	Unsets echo mode
nonl()	Unsets newline mapping
noraw()	Unsets raw mode
overlay(win1,win2)	Overlays win1 on win2
overwrite(win1,win2)	Overwrites win1 on top of win2
printw(fmt,arg1,arg2,...)	Printfs on *stdscr*
raw()	Sets raw mode
refresh()	Makes current screen look like *stdscr*
restty()	Resets tty flags to stored value
savetty()	Stored current tty flags
scanw(fmt,arg1,arg2,...)	Scanf through *stdscr*
scroll(win)	Scrolls *win* one line
scrollok(win,boolf)	Sets scroll flag
setterm(name)	Sets term variables for name
unctrl(ch)	Printable version of *ch*
waddch(win,ch)	Adds char to *win*
waddstr(win,str)	Adds string to *win*
wclear(win)	Clear *win*
wclrtobot(win)	Clears to bottom of *win*
wclrtoeol(win)	Clears to end of line on *win*
werase(win)	Erase *win*
wgetch(win)	Gets a char through *win*
wgetstr(win,str)	Gets a string through *win*
winch(win)	Gets char at current (y,x) in *win*
wmove(win,y,x)	Sets current (y,x) co-ordinates on *win*
wprintw(win,fmt,arg1,arg2,...)	Printf on *win*
wrefresh(win)	Makes screen look like *win*
wscanw(win,fmt,arg1,arg2,...)	Scanf through *win*

Credit

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

ınd cursor functions.

ltermlib [libraries]

a method of updating screens with rea-
keep an image of the current screen,
e of a new one. Then the *refresh()* tells
'rent screen look like the new one. In
:s, the routine *initscr()* must be called
ıes that deal with windows and screens

ıe loader option -lcurses.

a character to *stdscr*
a string to *stdscr*
ı a box around a window
break mode
stdscr
ear flag for *scr*
to bottom on *stdscr*
to end of line on *stdscr*
win
ho mode
tdscr
:har through *stdscr*
ɪtring through *stdscr*
modes
x) coordinates
ır at current (y,x) co-ordinates
s screens
e flag for *win*
g name from *termbuf*
› (y,x) on *stdscr*
lly moves cursor
Creates a new window

the login name
·, this represen-
ress of which is
to an array of at
ft in this array.
o.h>.

ıs NULL; if *s* is

results of a user's
ɜquent call to the

Name

dbminit, fetch, store, delete, firstkey, nextkey – Performs database functions.

Syntax

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

dbminit(file)
char *file;

datum fetch(key)
datum key;

store(key, content)
datum key, content;

delete(key)
datum key;

datum firstkey( );

datum nextkey(key);
datum key;

Description

These functions maintain key/content pairs in a database. The functions will handle very large (a billion blocks) databases and will access a keyed item in one or two file system accesses. The functions are obtained with the loader option – ldbm.

Keys and *contents* are described by the *datum* typedef. A *datum* specifies a string of *dsize* bytes pointed to by *dptr*. Arbitrary binary data, as well as normal ASCII strings, are allowed. The database is stored in two files. One file is a directory containing a bit map and has ".dir" as its suffix. The second file contains all data and has ".pag" as its suffix.

Before a database can be accessed, it must be opened by *dbminit*. At the time of this call, the files *file*.dir and *file*.pag must exist. (An empty database is created by creating zero-length ".dir" and ".pag" files.)

Once open, the data stored under a key is accessed by *fetch* and data is placed under a key by *store*. A key (and its associated contents) is deleted by *delete*. A linear pass through all keys in a database may be made, in an (apparently) random order, by use of *firstkey* and *nextkey*. *Firstkey* will return the first key in the database. With any key *nextkey* will return the next key in the database. This code will

Name

dbminit, fetch, store, delete, firstkey, nextkey – Performs database functions.

Syntax

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

dbminit(file)
char *file;

datum fetch(key)
datum key;

store(key, content)
datum key, content;

delete(key)
datum key;

datum firstkey();

datum nextkey(key);
datum key;
```

Description

These functions maintain key/content pairs in a database. The functions will handle very large (a billion blocks) databases and will access a keyed item in one or two file system accesses. The functions are obtained with the loader option – ldbm.

Keys and *contents* are described by the *datum* typedef. A *datum* specifies a string of *dsize* bytes pointed to by *dptr*. Arbitrary binary data, as well as normal ASCII strings, are allowed. The database is stored in two files. One file is a directory containing a bit map and has ".dir" as its suffix. The second file contains all data and has ".pag" as its suffix.

Before a database can be accessed, it must be opened by *dbminit*. At the time of this call, the files *file*.dir and *file*.pag must exist. (An empty database is created by creating zero-length ".dir" and ".pag" files.)

Once open, the data stored under a key is accessed by *fetch* and data is placed under a key by *store*. A key (and its associated contents) is deleted by *delete*. A linear pass through all keys in a database may be made, in an (apparently) random order, by use of *firstkey* and *nextkey*. *Firstkey* will return the first key in the database. With any key *nextkey* will return the next key in the database. This code will

Name

defopen, defread – Reads default entries.

Syntax

```
int defopen(filename)
char *filename;

char *defread(pattern)
char *pattern;
```

Description

Defopen and *defread* are a pair of routines designed to allow easy access to default definition files. XENIX is normally distributed in binary form; the use of default files allows OEMS or site administrators to customize utility defaults without having the source code.

Defopen opens the default file named by the pathname in *filename*. *Defopen* returns null if it is successful in opening the file, or the *fopen* failure code (*errno*) if the open fails.

Defread reads the previously opened file from the beginning until it encounters a line beginning with *pattern*. *Defread* then returns a pointer to the first character in the line after the initial *pattern*. If a trailing newline character is read it is replaced by a null byte.

When all items of interest have been extracted from the opened file the program may call *defopen* with the name of another file to be searched, or it may call *defopen* with NULL, which closes the default file without opening another.

Files

The XENIX convention is for a system program *xyz* to store its defaults (if any) in the file /etc/default/xyz.

Diagnostics

Defopen returns zero on success and nonzero if the open fails. The return value is the *errno* value set by *fopen*(S).

Defread returns NULL if a default file is not open, if the indicated pattern could not be found, or if it encounters any line in the file greater than the maximum length of 128 characters.

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

ecvt, fcvt, gcvt – Performs output conversions.

Syntax

```
char *ecvt (value, ndigit, decpt, sign)
double value;
int ndigit, *decpt, *sign;

char *fcvt (value, ndigit, decpt, sign)
double value;
int ndigit, *decpt, *sign;

char *gcvt (value, ndigit, buf)
double value;
char *buf;
```

Description

Ecvt converts the *value* to a null-terminated string of *ndigit* ASCII digits and returns a pointer to the string. The position of the decimal point relative to the beginning of the string is stored indirectly through *decpt* (negative means to the left of the returned digits). If the sign of the result is negative, the word pointed to by *sign* is nonzero, otherwise it is zero. The low-order digit is rounded.

Fcvt is identical to *ecvt*, except that the correct digit has been rounded for FORTRAN F format output of the number of digits specified by *ndigits*.

Gcvt converts the *value* to a null-terminated ASCII string in *buf* and returns a pointer to *buf*. It attempts to produce *ndigit* significant digits in FORTRAN F format if possible, otherwise E format, ready for printing. Trailing zeros may be suppressed.

See Also

printf(S)

Notes

The return values point to static data whose content is overwritten by each call.

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.

Envp is an array of character pointers to null-terminated strings. These strings constitute the environment for the new process. *Envp* is terminated by a null pointer.

File descriptors open in the calling process remain open in the new process, except for those whose close-on-exec flag is set; see *fcntl*(S). For those file descriptors that remain open, the file pointer is unchanged.

Signals set to terminate the calling process will be set to terminate the new process. Signals set to be ignored by the calling process will be set to be ignored by the new process. Signals set to be caught by the calling process will be set to terminate new process; see *signal*(S).

If the set-user-ID mode bit of the new process file is set (see *chmod*(S)), *exec* sets the effective user ID of the new process to the owner ID of the new process file. Similarly, if the set-group-ID mode bit of the new process file is set, the effective group ID of the new process is set to the group ID of the new process file. The real user ID and real group ID of the new process remain the same as those of the calling process.

Profiling is disabled for the new process; see *profil*(S).

The new process also inherits the following attributes from the calling process:

Nice value (see *nice*(S))

Process ID

Parent process ID

Process group ID

tty group ID (see *exit*(S) and *signal*(S))

Trace flag (see *ptrace*(S) request 0)

Time left until an alarm clock signal (see *alarm*(S))

Current working directory

Root directory

File mode creation mask (see *umask*(S))

File size limit (see *ulimit*(S))

utime, *stime*, *cutime*, and *cstime* (see *times*(S))

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

execl, execv, execle, execve, execlp, execvp – Executes a

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 n cess. The new process is constructed from an ordinary, ex file called the "new process file" . There can be no return successful *exec* because the calling process is overlaid by t process.

Path points to a pathname that identifies the new process file.

File points to the new process file. The path prefix for this obtained by a search of the directories passed as the *environm* "PATH =" (see *environ*(M)). The environment is supplied shell (see *sh*(C)).

Arg0, *arg1*, ..., *argn* are pointers to null-terminated ch strings. These strings constitute the argument list available new process. By convention, at least *arg0* must be present, must point to a string that is the same as *path* (or its last ponent).

Argv is an array of character pointers to null-terminated st These strings constitute the argument list available to the nev cess. By convention, *argv* must have at least one member, must point to a string that is the same as *path* (or its last ponent). *Argv* is terminated by a null pointer.

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

exp, log, pow, sqrt, log10 – Performs exponential, logarithm, power, square root functions.

Syntax

```
#include <math.h>

double exp (x)
double x;

double log (x)
double x;

double pow (x, y)
double x, y;

double sqrt (x)
double x;

double log10 (x)
double x;
```

Description

Exp returns the exponential function of *x*.

Log returns the natural logarithm of *x*.

Pow returns x^y.

Sqrt returns the square root of *x*.

See Also

intro(S), hypot(S), sinh(S)

Diagnostics

Exp and *pow* return a huge value when the correct value would overflow. A truly outrageous argument may also result in *errno* being set to ERANGE . *Log* returns a huge negative value and sets *errno* to EDOM when *x* is nonpositive. *Pow* returns a huge negative value and sets *errno* to EDOM when *x* is nonpositive and *y* is not an integer, or when *x* and *y* are both zero. *Sqrt* returns 0 and sets *errno* to EDOM when *x* is negative.

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)

Name

fcntl – Controls open files.

Syntax

```
#include <fcntl.h>

int fcntl (fildes, cmd, arg)
int fildes, cmd, arg;
```

Description

Fcntl provides for control over open files. *Fildes* is an open file descriptor obtained from a *creat*, *open*, *dup*, *fcntl*, or *pipe* system call.

The *cmds* available are:

F_DUPFD Returns a new file descriptor as follows:

Lowest numbered available file descriptor greater than or equal to *arg*.

Same open file (or pipe) as the original file.

Same file pointer as the original file (i.e., both file descriptors share one file pointer).

Same access mode (read, write or read/write).

Same file status flags (i.e., both file descriptors share the same file status flags).

The close-on-exec flag associated with the new file descriptor is set to remain open across *exec*(S) system calls.

F_GETFD Gets the close-on-exec flag associated with the file descriptor *fildes*. If the low-order bit is **0** the file will remain open across *exec*, otherwise the file will be closed upon execution of *exec*.

F_SETFD Sets the close-on-exec flag associated with *fildes* to the low-order bit of *arg* (**0** or **1** as above).

F_GETFL Gets *file* status flags.

F_SETFL Sets *file* status flags to *arg*. Only certain flags can be set.

›ne or more of the following is true:

not a valid open file descriptor. [EBADF]

; F_DUPFD and 20 file descriptors are currently open. ›E]

is F_DUPFD and *arg* is negative or greater than 20. 'AL]

e

uccessful completion, the value returned depends on *cmd* as :

_DUPFD	A new file descriptor
_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

;herwise, a value of – 1 is returned and *errno* is set to indicate the ror.

Also

close(S), exec(S), open(S)

Name

ferror, feof, clearerr, fileno – Determines stream status.

Syntax

```
#include <stdio.h>

int feof (stream)
FILE *stream;

int ferror (stream)
FILE *stream

clearerr (stream)
FILE *stream

int fileno(stream)
FILE *stream;
```

Description

Feof returns nonzero when end-of-file is read on the named input *stream*, otherwise zero.

Ferror returns nonzero when an error has occurred reading or writing the named *stream*, otherwise zero. Unless cleared by *clearerr*, the error indication lasts until the stream is closed.

Clearerr resets the error indication on the named *stream*.

Fileno returns the integer file descriptor associated with the *stream*, see *open*(S).

Feof, *ferror*, and *fileno* are implemented as macros; they cannot be redeclared.

See Also

open(S), fopen(S)

Name

ferror, feof, clearerr, fileno – Determines stream status.

Syntax

```
#include <stdio.h>

int feof (stream)
FILE *stream;

int ferror (stream)
FILE *stream

clearerr (stream)
FILE *stream

int fileno(stream)
FILE *stream;
```

Description

Feof returns nonzero when end-of-file is read on the named input *stream*, otherwise zero.

Ferror returns nonzero when an error has occurred reading or writing the named *stream*, otherwise zero. Unless cleared by *clearerr*, the error indication lasts until the stream is closed.

Clearerr resets the error indication on the named *stream*.

Fileno returns the integer file descriptor associated with the *stream*, see *open*(S).

Feof, *ferror*, and *fileno* are implemented as macros; they cannot be redeclared.

See Also

open(S), fopen(S)

a stream.

stream)

'mptied, and
nput/output

by *filename* and associates a stream with
to be used to identify the stream in sub-

ream to be

aving one of the following values:

rors were

riting at end of file, or create for writing

eading and writing)

create for update at end of file

named file in place of the open *stream*. It
lue of *stream*. The original stream is closed,
he open call ultimately succeeds.

sed to attach the preopened constant names
err to specified files.

tream with a file descriptor obtained from *open*,
). The *type* of the stream must agree with the
e. The *type* must be provided because the stan-
no way to query the type of an open file descrip-
the new stream.

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.

Name

fork – Creates a new process.

Syntax

int fork ( )

Description

Fork causes creation of a new process. The new process (child process) is an exact copy of the calling process (parent process) except for the following:

The child process has a unique process ID.

The child process has a different parent process ID (i.e., the process ID of the parent process).

The child process has its own copy of the parent's file descriptors. Each of the child's file descriptors shares a common file pointer with the corresponding file descriptor of the parent.

The child process' *utime*, *stime*, *cutime*, and *cstime* are set to 0; see *times*(S).

The time left on the parent's alarm clock is not passed on to the child.

Fork returns a value of 0 to the child process.

Fork returns the process ID of the child process to the parent process.

Fork will fail and no child process will be created if one or more of the following are true:

The system-imposed limit on the total number of processes under execution would be exceeded. [EAGAIN]

The system-imposed limit on the total number of processes under execution by a single user would be exceeded. [EAGAIN]

Not enough memory is available to create the forked image. [ENOMEM]

Return Value

Upon successful completion, *fork* returns a value of 0 to the child process and returns the process ID of the child process to the parent 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

fread, fwrite – Performs buffered binary input and output.

Syntax

```
#include <stdio.h>

int fread ((char *) ptr, sizeof (*ptr), nitems, stream)
FILE *stream;

int fwrite ((char *) ptr, sizeof (*ptr), nitems, stream)
FILE *stream;
```

Description

Fread reads, into a block beginning at *ptr*, *nitems* of data of the type of **ptr* from the named input *stream*. It returns the number of items actually read.

Fwrite appends at most *nitems* of data of the type of **ptr* beginning at *ptr* to the named output *stream*. It returns the number of items actually written.

See Also

read(S), write(S), fopen(S), getc(S), putc(S), gets(S), puts(S), printf(S), scanf(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

fseek, ftell, rewind – Repositions a stream.

Syntax

```
#include <stdio.h>

int fseek (stream, offset, ptrname)
FILE *stream;
long offset;
int ptrname;

long ftell (stream)
FILE *stream;

rewind(stream)
FILE *stream;
```

Description

Fseek sets the position of the next input or output operation on the *stream*. The new position is at the signed distance *offset* bytes from the beginning, the current position, or the end of the file, according as *ptrname* has the value 0, 1, or 2.

Fseek undoes any effects of *ungetc*(S).

After *fseek* or *rewind*, the next operation on an update file may be either input or output.

Ftell returns the current value of the offset relative to the beginning of the file associated with the named *stream*. The offset is measured in bytes.

Rewind(*stream*) is equivalent to *fseek*(*stream*, 0L, 0).

See Also

lseek(S), fopen(S)

Diagnostics

Fseek returns nonzero for improper seeks, otherwise zero.

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

getc, getchar, fgetc, getw – Gets character or word from a stream.

Syntax

```
#include <stdio.h>

int getc (stream)
FILE *stream;

int getchar ( )

int fgetc (stream)
FILE *stream;

int getw (stream)
FILE *stream;
```

Description

Getc returns the next character from the named input *stream*.

Getchar( ) is identical to *getc*(*stdin*).

Fgetc behaves like *getc*, but is a genuine function, not a macro; it may therefore be used as an argument. *Fgetc* runs more slowly than *getc*, but takes less space per invocation.

Getw returns the next word from the named input *stream*. It returns the constant EOF upon end-of-file or error, but since that is a valid integer value, *feof* and *ferror*(S) should be used to check the success of *getw*. *Getw* assumes no special alignment in the file.

See Also

ferror(S), fopen(S), fread(S), gets(S), putc(S), scanf(S)

Diagnostics

These functions return the integer constant EOF at the end-of-file or upon a read error.

Notes

Because *getc* is implemented as a macro, *stream* arguments with side effects are treated incorrectly. In particular, "getc(*f++)" doesn't work properly.

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

getenv – Gets value for environment name.

Syntax

```
char *getenv (name)
char *name;
```

Description

Getenv searches the environment list (see *environ*(M)) for a string of the form *name=value* and returns *value* if such a string is present, otherwise 0 (NULL).

See Also

sh(C), exec(S)

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

See Also

getlogin(S), getpwent(S), group(M)

Diagnostics

A null pointer (0) is returned on end-of-file or error.

Notes

All information is contained in a static area, so it must be copied if it is to be saved.

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.

Name

getopt – Gets option letter from argument vector.

Syntax

```
#include <stdio.h>

int getopt (argc, argv, optstring)
int argc;
char **argv;
char *optstring;
extern char *optarg;
extern int optind;
```

Description

Getopt returns the next option letter in *argv* that matches a letter in *optstring*. *Optstring* is a string of recognized option letters; if a letter is followed by a colon, the option is expected to have an argument that may or may not be separated from it by whitespace. *Optarg* is set to point to the start of the option argument on return from *getopt*.

Getopt places in *optind* the *argv* index of the next argument to be processed. Because *optind* is external, it is normally initialized to zero automatically before the first call to *getopt*.

When all options have been processed (i.e., up to the first nonoption argument), *getopt* returns EOF. The special option -- may be used to delimit the end of the options; EOF will be returned, and -- will be skipped.

Diagnostics

Getopt prints an error message on *stderr* and returns a question mark (?) when it encounters an option letter not included in *optstring*.

Examples

The following code fragment shows how one might process the arguments for a command that can take the mutually exclusive options a and b, and the options f and o, both of which require arguments:

```
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

getpass – Reads a password.

Syntax

char *getpass (prompt)
char *prompt;

Description

Getpass reads a password from the file /dev/tty, or if that cannot be opened, from the standard input, after prompting with the null-terminated string *prompt* and disabling echoing. A pointer is returned to a null-terminated string of at most eight characters.

Files

/dev/tty

See Also

crypt(S)

Notes

The return value points to static data whose content is overwritten by each call.

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

getpw – Gets password for a given user ID.

Syntax

```
getpw (uid, buf)
int uid;
char *buf;
```

Description

Getpw searches the password file for the *uid*, and fills in *buf* with the corresponding line; it returns nonzero if *uid* could not be found. The line is null-terminated. *Uid* must be an integer value.

Files

/etc/passwd

See Also

getpwent(S), passwd(M)

Diagnostics

Returns nonzero on error.

Notes

This routine is included only for compatibility with prior systems and should not be used; see *getpwent*(S) for routines to use instead.

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)

Diagnostics

Null pointer (0) returned on EOF or error.

Notes

All information is contained in a static area so it must be copied if it is to be saved.

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

getuid, geteuid, getgid, getegid – Gets real user, effective user, real group, and effective group IDs.

Syntax

int getuid ( )

int geteuid ( )

int getgid ( )

int getegid ( )

Description

Getuid returns the real user ID of the calling process.

Geteuid returns the effective user ID of the calling process.

Getgid returns the real group ID of the calling process.

Getegid returns the effective group ID of the calling process.

See Also

intro(S), setuid(S)

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

ioctl – Controls character devices.

Syntax

#include <sys/ioctl.h>

ioctl(fildes, request, arg)
int fildes;

Description

Ioctl performs a variety of functions on character special files (devices). The writeups of various devices in Section M discuss how *ioctl* applies to them.

Ioctl will fail if one or more of the following are true:

> *Fildes* is not a valid open file descriptor. [EBADF]
>
> *Fildes* is not associated with a character special device. [ENOTTY]
>
> *Request* or *arg* is not valid. See *tty*(M). [EINVAL]

Return Value

If an error has occurred, a value of – 1 is returned and *errno* is set to indicate the error.

See Also

tty(M)

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]

The sending process is not sending to itself, its effective user ID is not super-user, and its effective user ID does not match the real user ID of the receiving process. [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

kill(C), getpid(S), setpgrp(S), signal(S)

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)

Name

link - Links a new filename to an existing file.

Syntax

int link (path1, path2)
char *path1, *path2;

Description

Path1 points to a pathname naming an existing file. *Path2* points to a pathname giving the new filename to be linked. *Link* makes a new link by creating a new directory entry for the existing file using the new name. The contents of the existing file can then be accessed using either name.

Link will fail and no link will be created if one or more of the following are true:

A component of either path prefix is not a directory. [ENOTDIR]

A component of either path prefix does not exist. [ENOENT]

A component of either path prefix denies search permission. [EACCES]

The file named by *path1* does not exist. [ENOENT]

The link named by *path2* already exists. [EEXIST]

The file named by *path1* is a directory and the effective user ID is not super-user. [EPERM]

The link named by *path2* and the file named by *path1* are on different logical devices (file systems). [EXDEV]

Path2 points to a null pathname. [ENOENT]

The requested link requires writing in a directory with a mode that denies write permission. [EACCES]

The requested link requires writing in a directory 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

ln(C)

Name

lock – Locks a process in primary memory.

Syntax

lock(flag)

Description

If the *flag* argument is nonzero, the process executing this call will not be swapped except if it is required to grow. If the argument is zero, the process is un*lock*ed. This call may only be executed by the super-user.

Notes

*Lock*ed processes interfere with the compaction of primary memory and can cause deadlock. Systems with small memory configurations should avoid using this call. It is best to lock process soon after booting because that will tend to lock them into one end of memory.

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).

LK_NBLCK 2
Locks the specified region. If any part of the region is already locked by a different process, return the error EACCES instead of waiting for the region to become available for locking (nonblocking lockrequest).

LK_RLCK 3
Same as LK_LOCK except that the locked region may be read by other processes (read permitted lock).

LK_NBRLCK 4
Same as LK_NBLCK except that the locked region may be read by other processes (nonblocking, read permitted lock).

Size is the number of contiguous bytes to be locked or unlocked. The region to be locked starts at the current offset in the file. If *size* is 0, the entire file (up to a maximum of 2 to the power of 30 bytes) is locked or unlocked. *Size* may extend beyond the end of the file, in which case only the process issuing the lock call may access or add information to the file within the boundary defined by size.

The potential for a deadlock occurs when a process controlling a locked area is put to sleep by accessing another process' locked area. Thus calls to *locking*, *read*, or *write* scan for a deadlock prior to sleeping on a locked region. An error return is made if sleeping on the locked region would cause a deadlock.

Lock requests may, in whole or part, contain or be contained by a previously locked region for the same process. When this occurs, or when adjacent regions are locked, the regions are combined into a single area if the mode of the lock is the same (i.e.; either read permitted or regular lock). If the mode of the overlapping locks differ, the locked areas will be assigned assuming that the *most recent request* must be satisfied. Thus if a read only lock is applied to a region, or part of a region, that had been previously locked by the same process against both reading and writing, the area of the file specified by the new lock will be locked for read only, while the remaining region, if any, will remain locked against reading and writing. There is no arbitrary limit to the number of regions which may be locked in a file. There is however a system-wide limit on the total number of locked regions. This limit is 200 for XENIX systems.

Unlock requests may, in whole or part, release one or more locked regions controlled by the process. When regions are not fully released, the remaining areas are still locked by the process. Release of the center section of a locked area requires an additional locked element to hold the separated section. If the lock table is full, an error is returned, and the requested region is not released. Only the process which locked the file region may unlock it. An unlock request for a region that the process does not have locked, or that is already unlocked, has no effect. When a process terminates, all locked regions controlled by that process are unlocked.

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

logname – Finds login name of user.

Syntax

```
char *logname( );
```

Description

Logname returns a pointer to the null-terminated login name. It uses the string found in the LOGNAME variable from the user's environment.

Files

/etc/profile

See Also

env(C), login(M), profile(M), environ(M)

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)

Name

lseek – Moves read/write file pointer.

Syntax

```
long lseek (fildes, offset, whence)
int fildes;
long offset;
int whence;
```

Description

Fildes is a file descriptor returned from a *creat*, *open*, *dup*, or *fcntl* system call. *Lseek* sets the file pointer associated with *fildes* as follows:

If *whence* is 0, the pointer is set to *offset* bytes.

If *whence* is 1, the pointer is set to its current location plus *offset*.

If *whence* is 2, the pointer is set to the size of the file plus *offset*.

Upon successful completion, the resulting pointer location as measured in bytes from the beginning of the file is returned.

Lseek will fail and the file pointer will remain unchanged if one or more of the following are true:

Fildes is not an open file descriptor. [EBADF]

Fildes is associated with a pipe or fifo. [ESPIPE]

Whence is not 0, 1 or 2. [EINVAL and SIGSYS signal]

The resulting file pointer would be negative. [EINVAL]

Some devices are incapable of seeking. The value of the file pointer associated with such a device is undefined.

Return Value

Upon successful completion, a nonnegative integer indicating the file pointer value is returned. Otherwise, a value of – 1 is returned and *errno* is set to indicate the error.

See Also

creat(S), dup(S), fcntl(S), open(S)

Name

malloc, free, realloc, calloc – Allocates main memory.

Syntax

char *malloc (size) unsigned size;

free (ptr)
char *ptr;

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

char *calloc (nelem, elsize)
unsigned elem, elsize;

Description

Malloc and *free* provide a simple general-purpose memory allocation package. *Malloc* returns a pointer to a block of at least *size* bytes beginning on a word boundary.

The argument to *free* is a pointer to a block previously allocated by *malloc*; this space is made available for further allocation, but its contents are left undisturbed.

Needless to say, grave disorder will result if the space assigned by *malloc* is overrun or if some random number is handed to *free*.

Malloc allocates the first contiguous reach of free space found in a circular search from the last block allocated or freed, coalescing adjacent free blocks as it searches. It calls *sbrk* (see *sbrk*(S)) to get more memory from the system when there is no suitable space already free.

Realloc changes the size of the block pointed to by *ptr* to *size* bytes and returns a pointer to the (possibly moved) block. The contents will be unchanged up to the lesser of the new and old sizes.

Realloc also works if *ptr* points to a block freed since the last call of *malloc*, *realloc*, or *calloc*; thus sequences of *free*, *malloc* and *realloc* can exploit the search strategy of *malloc* to do storage compaction.

Calloc allocates space for an array of *nelem* elements of size *elsize*. The space is initialized to zeros.

Each of the allocation routines returns a pointer to space suitably aligned (after possible pointer coercion) for storage of any type of 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.

Name

mknod – Makes a directory, or a special or ordinary file.

Syntax

int mknod (path, mode, dev)
char *path;
int mode, dev;

Description

Mknod creates a new file named by the pathname pointed to by *path*. The mode of the new file is initialized from *mode*. Where the value of *mode* is interpreted as follows:

0170000 File type; one of the following:
- 0010000 Named pipe special
- 0020000 Character special
- 0040000 Directory
- 0050000 Name special file
- 0060000 Block special
- 0100000 or 0000000 Ordinary file

0004000 Set user ID on execution

0002000 Set group ID on execution

0001000 Save text image after execution

0000777 Access permissions; constructed from the following
- 0000400 Read by owner
- 0000200 Write by owner
- 0000100 Execute (search on directory) by owner
- 0000070 Read, write, execute (search) by group
- 0000007 Read, write, execute (search) by others

Values of *mode* other than those above are undefined and should not be used.

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.

The low-order 9 bits of *mode* are modified by the process' file mode creation mask: all bits set in the process' file mode creation mask are cleared. See *umask*(S). If *mode* indicates a block, character, or name special file, then *dev* is a configuration dependent specification of a character or block I/O device. If *mode* does not indicate a block, character, or name special file, then *dev* is ignored. For block and character special files, *dev* is the special file's device number. For name special files, *dev* is the type of the name file, either a 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

mktemp – Makes a unique filename.

Syntax

char *mktemp(template)
char *template;

Description

Mktemp replaces *template* with a unique filename, and returns a pointer to the name. The template should look like a filename with six trailing X's, which will be replaced with the current process ID preceded by a zero.

See Also

getpid(S)

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)

Notes

An executable program created by cc – p automatically includes calls for *monitor* with default parameters; *monitor* needn't be called explicitly except to gain fine control over profiling.

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]

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

mount(C), umount(S)

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

nice – Changes priority of a process.

Syntax

```
int nice (incr)
int incr;
```

Description

Nice adds the value of *incr* to the nice value of the calling process. A process' nice value is a positive number for which a higher value results in lower CPU priority.

A maximum nice value of 39 and a minimum nice value of 0 are imposed by the system. Requests for values above or below these limits result in the nice value being set to the corresponding limit.

Nice will not change the nice value if *incr* is negative and the effective user ID of the calling process is not super-user. [EPERM]

Return Value

Upon successful completion, *nice* returns the new nice value minus 20. Note that *nice* is unusual in the way return codes are handled. It differs from most other system calls in two ways: the value – 1 is a valid return code (in the case where the new nice value is 19), and the system call either works or ignores the request; there is never an error.

See Also

nice(C), exec(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.

Name

open – Opens file for reading or writing.

Syntax

```
#include <fcntl.h>
int open (path, oflag[, mode])
char *path;
int oflag, mode;
```

Description

Path points to a pathname naming a file. *Open* opens a file descriptor for the named file and sets the file status flags according to the value of *oflag*. *Oflag* values are constructed by or-ing flags from the following list (only one of the first three flags below may be used):

O_RDONLY Open for reading only.

O_WRONLY Open for writing only.

O_RDWR Open for reading and writing.

O_NDELAY This flag may affect subsequent reads and writes. See *read*(S) and *write*(S).

When opening a FIFO with O_RDONLY or O_WRONLY set:

If O_NDELAY is set:

An *open* for reading-only will return without delay. An *open* for writing-only will return an error if no process currently has the file open for reading.

If O_NDELAY is clear:

An *open* for reading-only will block until a process opens the file for writing. An *open* for writing-only will block until a process opens the file for reading.

When opening a file associated with a communication line:

If O_NDELAY is set:

The open will return without waiting for carrier.

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]

Oflag permission is denied for the named file. [EACCES]

The named file is a directory and *oflag* is write or read/write. [EISDIR]

The named file resides on a read-only file system and *oflag* is write or read/write. [EROFS]

Twenty file descriptors are currently open. [EMFILE]

The named file is a character special or block special file, and the device associated with this special file does not exist. [ENXIO]

The file is a pure procedure (shared text) file that is being executed and *oflag* is write or read/write. [ETXTBSY]

Path points outside the process' allocated address space. [EFAULT]

O_CREAT and O_EXCL are set, and the named file exists. [EEXIST]

O_NDELAY is set, the named file is a FIFO, O_WRONLY is set, and no process has the file open for reading. [ENXIO]

Return Value

Upon successful completion, a nonnegative integer, namely a file descriptor, is returned. Otherwise, a value of -1 is returned and *errno* is set to indicate the error.

See Also

close(S), creat(S), dup(S), fcntl(S), lseek(S), read(S), write(S)

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*. *Creatsem* 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

pause – Suspends a process until a signal occurs.

Syntax

int pause ( );

Description

Pause suspends the calling process until it receives a signal. The signal must be one that is not currently set to be ignored by the calling process.

If the signal causes termination of the calling process, *pause* will not return.

If the signal is *caught* by the calling process and control is returned from the signal catching function (see *signal*(S)), the calling process resumes execution from the point of suspension; with a return value of – 1 from *pause* and *errno* set to EINTR.

See Also

alarm(S), kill(S), signal(S), wait(S)

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

pipe – Creates an interprocess pipe.

Syntax

```
int pipe (fildes)
int fildes[2];
```

Description

Pipe creates an I/O mechanism called a pipe and returns two file descriptors in the array *fildes*. *Fildes*[0] is opened for reading and *fildes*[1] is opened for writing. The descriptors remain open across *fork*(S) system calls, making communication between parent and child possible.

Writes up to 5120 bytes of data are buffered by the pipe before the writing process is blocked. A read on file descriptor *fildes*[0] accesses the data written to *fildes*[1] on a first-in-first-out basis.

No process may have more than 20 file descriptors open simultaneously.

Pipe will fail if 19 or more file descriptors are currently open. [EMFILE]

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

sh(C), read(S), write(S), fork(S), popen(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).

Name

printf, fprintf, sprintf – Formats output.

Syntax

```
#include <stdio.h>

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

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

int sprintf (s, format [ , arg ] ... )
char *s, format;
```

Description

Printf places output on the standard output stream stdout. *Fprintf* places output on the named output *stream*. *Sprintf* places output, followed by the null character (\0) in consecutive bytes starting at *s*; it is the user's responsibility to ensure that enough storage is available. Each function returns the number of characters placed (not including the \0 in the case of *sprintf*), or a negative value if an output error was encountered.

Each of these functions converts, formats, and prints its *args* under control of the *format*. The *format* is a character string that contains two types of objects: plain characters, which are simply copied to the output stream, and conversion specifications, each of which results in fetching of zero or more *args*. The results are undefined if there are insufficient *args* for the format. If the format is exhausted while *args* remain, the excess *args* are simply ignored.

Each conversion specification is introduced by the character % After the % the following appear in sequence:

Zero or more *flags*, which modify the meaning of the conversion specification.

An optional decimal digit string specifying a minimum *field width*. If the converted value has fewer characters than the field width, it will be padded on the left (or right, if the left-adjustment flag described below has been given) to the field width.

A *precision* that gives the minimum number of digits to appear for the d, o, u, x, or X conversions, the number of digits to appear after the decimal point for the e and f conversions, the 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 o, x, or X *and* the # flag is present).

f The float or double *arg* is converted to decimal notation in the style "[–]ddd.ddd", where the number of digits after the decimal point is equal to the precision specification. If the precision is missing, six digits are output; if the precision is explicitly 0, no decimal point appears.

e,E The float or double *arg* is converted in the style "[–]d.ddde± dd", where there is one digit before the decimal point and the number of digits after it is equal to the precision; when the precision is missing, 6 digits are produced; if the precision is zero, no decimal point appears. The E format code will produce a number with E instead of e introducing the exponent. The exponent always contains exactly two digits.

g,G The float or double *arg* is printed in style f or e (or in style E in the case of a G format code), with the precision specifying the number of significant digits. The style used depends on the value converted: style e will be used only if the exponent resulting from the conversion is less than – 4 or greater than the precision. Trailing zeroes are removed from the result; a decimal point appears only if it is followed by a digit.

c The character *arg* is printed.

s The *arg* is taken to be a string (character pointer) and characters from the string are printed until a null character (\0) is encountered or the number of characters indicated by the precision specification is reached. If the precision is missing, it is taken to be infinite, so all characters up to the first null character are printed.

% Print a %; no argument is converted.

In no case does a nonexistent or small field width cause truncation of a field; if the result of a conversion is wider than the field width, the field is simply expanded to contain the conversion result. Characters generated by *printf* and *fprintf* are printed as if *putchar* had been called (see *putc*(S)).

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

profil – Creates an execution time profile.

Syntax

```
profil (buff, bufsiz, offset, scale)
char *buff;
int bufsiz, offset, scale;
```

Description

Buff points to an area of core whose length (in bytes) is given by *bufsiz*. After this call, the user's program counter is examined each clock tick, where a clock tick is some fraction of a second given in *machine*(M). *Offset* is subtracted from it, and the result multiplied by *scale*. If the resulting number corresponds to a word inside *buff*, that word is incremented.

The scale is interpreted as an unsigned, fixed-point fraction with binary point at the left: 0177777 (octal) gives a 1-1 mapping of pc's to words in *buff*; 077777 (octal) maps each pair of instruction words together. 02(octal) maps all instructions onto the beginning of *buff* (producing a noninterrupting core clock).

Profiling is turned off by giving a *scale* of 0 or 1. It is rendered ineffective by giving a *bufsiz* of 0. Profiling is turned off when an *exec* is executed, but remains on in child and parent both after a *fork*. Profiling will be turned off if an update in *buff* would cause a memory fault.

See Also

prof(CP), monitor(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

not defined for this request. Peculiar results will ensue if the parent does not expect to trace the child.

The remainder of the requests can only be used by the parent process. For each, *pid* is the process ID of the child. The child must be in a stopped state before these requests are made.

1, 2 The word at location *addr* in the address space of the child is returned to the parent process. If I and D space are separated, request 1 returns a word from I space, and request 2 returns a word from D space. If I and D space are not separated, either request 1 or request 2 may be used with equal results. The *data* argument is ignored. These two requests will fail if *addr* is not the start address of a word, in which case a value of − 1 is returned to the parent process and the parent's *errno* is set to EIO.

3 With this request, the word at location *addr* in the child's USER area in the system's address space (see <sys/user.h>) is returned to the parent process. The *data* argument is ignored. This request will fail if *addr* is not the start address of a word or is outside the USER area, in which case a value of − 1 is returned to the parent process and the parent's *errno* is set to EIO.

4, 5 With these requests, the value given by the *data* argument is written into the address space of the child at location *addr*. If I and D space are separated, request 4 writes a word into I space, and request 5 writes a word into D space. If I and D space are not separated, either request 4 or request 5 may be used with equal results. Upon successful completion, the value written into the address space of the child is returned to the parent. These two requests will fail if *addr* is a location in a pure procedure space and another process is executing in that space, or *addr* is not the start address of a word. Upon failure a value of − 1 is returned to the parent process and the parent's *errno* is set to EIO.

6 With this request, a few entries in the child's USER area can be written. *Data* gives the value that is to be written and *addr* is the location of the entry. The few entries that can be written follow:

- The general registers
- Any floating-point status registers
- Certain bits of the processor status

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 portable across all implementations without some change. Please note that IBM-PC performs no memory mapping.

System calls cannot be single-stepped. If a *ptrace* call requests a single step through a system call, the trace bit is cleared, and the user program will run to completion or until it encounters an explicitly set breakpoint.

See Also

adb(CP), exec(S), signal(S), wait(S), machine(M)

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)

Diagnostics

These functions return the constant EOF upon error. Since this is a valid integer, *ferror*(S) should be used to detect *putw* errors.

Notes

Because *putc* is implemented as a macro, the *stream* argument with side effects is not treated correctly.

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

puts, fputs – Puts a string on a stream.

Syntax

```
#include <stdio.h>

int puts (s)
char *s;

int fputs (s, stream)
char *s;
FILE *stream;
```

Description

Puts copies the null-terminated string *s* to the standard output stream *stdout* and appends a newline character.

Fputs copies the null-terminated string *s* to the named output *stream*.

Neither routine copies the terminating null character.

Diagnostics

Both routines return EOF on error.

See Also

ferror(S), fopen(S), fread(S), gets(S), printf(S), putc(S)

Notes

Puts appends a newline, *fputs* does not.

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

rand, srand – Generates a random number.

Syntax

```
srand (seed)
unsigned seed;

int rand ( )
```

Description

Rand uses a multiplicative congruential random number generator with period 2^{32} to return successive pseudo-random numbers in the range from 0 to $2^{15} - 1$.

The generator is reinitialized by calling *srand* with 1 as argument. It can be set to a random starting point by calling *srand* with an unsigned integer in argument *seed*.

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.

Name

read – Reads from a file.

Syntax

```
int read (fildes, buf, nbyte)
int fildes;
char *buf;
unsigned nbyte;
```

Description

Fildes is a file descriptor obtained from a *creat*, *open*, *dup*, *fcntl*, or *pipe* system call.

Read attempts to read *nbyte* bytes from the file associated with *fildes* into the buffer pointed to by *buf*.

On devices capable of seeking, the *read* starts at a position in the file given by the file pointer associated with *fildes*. Upon return from *read*, the file pointer is incremented by the number of bytes actually read.

Devices that are incapable of seeking always read from the current position. The value of a file pointer associated with such a file is undefined.

Upon successful completion, *read* returns the number of bytes actually read and placed in the buffer; this number may be less than *nbyte* if the file is associated with a communication line (see *ioctl*(S) and *tty*(M)), or if the number of bytes left in the file is less than *nbyte* bytes. A value of 0 is returned when an end-of-file has been reached.

When attempting to read from an empty pipe (or FIFO):

> If O_NDELAY is set, the read will return a 0.
>
> If O_NDELAY is clear, the read will block until data is written to the file or the file is no longer open for writing.

When attempting to read a file associated with a tty that has no data currently available:

> If O_NDELAY is set, the read will return a 0.
>
> If O_NDELAY is clear, the read will block until data becomes available.

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.

Name

regex, regcmp – Compiles and executes regular expressions.

Syntax

```
char *regcmp(string1[,string2, ...],0);
char *string1, *string2, ...;

char *regex(re,subject[,ret0, ...]);
char *re, *subject, *ret0, ...;
```

Description

Regcmp compiles a regular expression and returns a pointer to the compiled form. *Malloc*(S) is used to create space for the compiled expression. It is the user's responsibility to free unneeded space so allocated. A zero return from *regcmp* indicates an incorrect argument. *Regcmp*(CP) has been written to generally preclude the need for this routine at execution time.

Regex executes a compiled pattern against the subject string. Additional arguments are passed to receive values back. *Regex* returns zero on failure or a pointer to the next unmatched character on success. A global character pointer *_loc1* points to where the match began. Although *regcmp* and *regex* were derived from the editor, *ed*(C), the syntax and semantics have been changed slightly. The following are the valid symbols and their associated meanings.

[] *.^ These symbols retain their current meaning.

$ Matches the end of the string, \n matches the newline.

– Within brackets the minus means *through*. For example, [a– z] is equivalent to [abcd...xyz]. The – can appear as itself only if used as the last or first character. For example, the character class expression []–] matches the characters] and – .

+ A regular expression followed by + means "one or more times". For example, [0– 9]+ is equivalent to [0– 9][0– 9]*.

{m} {m,} {m,u}
Integer values enclosed in {} indicate the number of times the preceding regular expression is to be applied. *m* is the minimum number and *u* is a number, less than 256, which is the maximum. If only *m* is present (e.g., {m}), it indicates the exact number of times the regular expression is

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*.

See Also

ed(C), regcmp(CP), malloc(S)

Notes

The user program may run out of memory if *regcmp* is called iteratively without freeing the vectors no longer required. The following user-supplied replacement for *malloc*(S) reuses the same vector saving time and space:

```
/* user's program */
        ...
malloc(n)
{
        static int rebuf[256];
        return &rebuf;
}
```

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), *sed*(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( )).

UNGETC(*c*)	Cause the argument *c* to be returned by the next call to GETC( ) (and PEEKC( )). No more that one character of pushback is ever needed and this character is guaranteed to be the last character read by GETC( ). The value of the macro UNGETC(*c*) is always ignored.
RETURN(*pointer*)	This macro is used on normal exit of the *compile* routine. The value of the argument *pointer* is a pointer to the character after the last character of the compiled regular expression. This is useful to programs which have memory allocation to manage.
ERROR(*val*)	This is the abnormal return from the *compile* routine. The argument *val* is an error number (see table below for meanings). This call should never return.

Error	*Meaning*
11	Range endpoint too large
16	Bad number
25	"\digit" out of range
36	Illegal or missing delimiter
41	No remembered search string
42	\(\) imbalance
43	Too many \(
44	More than 2 numbers given in \{ \}
45	} expected after \
46	First number exceeds second in \{ \}
49	[ ] imbalance
50	Regular expression overflow

The syntax of the *compile* routine is as follows:

```
compile(instring, expbuf, endbuf, eof)
```

The first parameter *instring* is never used explicitly by the *compile* routine but is useful for program that pass down different pointers to input characters. It is sometimes used in the INIT declaration (see below). Programs which call functions to input characters or have characters in an external array can pass down a value of ((char *) 0) for this parameter.

The next parameter *expbuf* is a character pointer. It points to the place where the compiled regular expression will be placed.

The parameter *endbuf* is one more that the highest address that the compiled regular expression may be placed. If the compiled expression cannot fit in (*endbuf*– *expbuf*) bytes, a call to ERROR(50) is made.

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 far as possible and will recursively call itself trying to match the rest of the string to the rest of the regular expression. As long as there is no match, *advance* will back up along the string until it finds a match or reaches the point in the string that initially matched the * or **\{ \}**. It is sometimes desirable to stop this backing up before the initial point in the string is reached. If the external character pointer *locs* is equal to the point in the string at sometime during the backing up process, *advance* will break out of the loop that backs up and will return zero. This is used be *ed*(C) and *sed*(C) for substitutions done globally (not just the first occurrence, but the whole line) so, for example, expressions like **s/y*//g** do not loop forever.

The routines *ecmp* and *getrange* are simple and are called by the routines previously mentioned.

Examples

The following is an example of how the regular expression macros and calls look from *grep*(C):

```
#define INIT            register char *sp = instring;
#define GETC()          (*sp++)
#define PEEKC()         (*sp)
#define UNGETC(c)       (--sp)
#define RETURN(c)       return;
#define ERROR(c)        regerr()

#include <regexp.h>
...
                compile(*argv, expbuf, &expbuf[ESIZE], '\0');
...
                if(step(linebuf, expbuf))
                                succeed();
```

Files

/usr/include/regexp.h

See Also

ed(C), grep(C), sed(C).

Notes

The handling of *circf* is kludgy.

The routine *ecmp* is equivalent to the standard I/O routine *strncmp* and should be replaced by that routine.

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)

Name

scanf, fscanf, sscanf – Converts and formats input.

Syntax

#include <stdio.h>

int scanf (format [, pointer] ...)
char *format;

int fscanf (stream, format [, pointer] ...)
FILE *stream;
char *format;

int sscanf (s, format [, pointer] ...)
char *s, *format;

Description

Scanf reads from the standard input stream *stdin*. *Fscanf* reads from the named input *stream*. *Sscanf* reads from the character string *s*. Each function reads characters, interprets them according to a format, and stores the results in its arguments. Each expects, as arguments, a control string *format* described below, and a set of *pointer* arguments indicating where the converted input should be stored.

The control string usually contains conversion specifications, which are used to direct interpretation of input sequences. The control string may contain:

1. Blanks, tabs, or newlines, which cause input to be read up to the next nonwhitespace character.

2. An ordinary character (not %), which must match the next character of the input stream.

3. Conversion specifications, consisting of the character %, an optional assignment suppressing character *, an optional numerical maximum field width, and a conversion character.

A conversion specification directs the conversion of the next input field; the result is placed in the variable pointed to by the corresponding argument, unless assignment suppression was indicated by *. An input field is defined as a string of nonspace characters; it extends to the next inappropriate character or until the field width, if specified, is exhausted.

The conversion character indicates the interpretation of the input field; the corresponding pointer argument must usually be of a restricted type. The following conversion characters are allowed:

% 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* *latter case, the offending character* is left unread in the input stream. This is very important to remember, because subtle errors can occur when not taking this into account.

Scanf returns the number of successfully matched and assigned input items; this number can be zero in the event of an early conflict between an input character and the control string. If the input ends before the first conflict or conversion, EOF is returned.

Examples

The call:

```
int i; float x; char name[50];
scanf ("%d%f%s", &i, &x, name);
```

with the input line:

```
25 54.32E-1 thompson
```

will assign to *i* the value 25, to *x* the value 5.432, and *name* will contain thompson\0. Or:

```
int i; float x; char name[50];
scanf ("%2d%f%*d%[1234567890]", &i, &x, name);
```

with input:

```
56789 0123 56a72
```

will assign 56 to *i*, 789.0 to *x*, skip 0123, and place the string 56\0 in *name*. The next call to *getchar* (see *getc*(S)) will return a.

See Also

atof(S), getc(S), printf(S)

Diagnostics

These functions return EOF on end of input and a short count for missing or illegal data items.

Notes

The success of literal matches and suppressed assignments is not directly determinable.

Trailing whitespace (including a newline) is left unread unless matched in the control string.

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.

Return Value

Successful calls return 0. Unsuccessful calls return −1, and *errno* is set to indicate the error.

See Also

sdget(S), sdgetv(S)

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*.

Return Value

On successful completion, the address at which the segment was attached is returned. Otherwise, -1 is returned, and *errno* is set to indicate the error.

Notes

Use of the SD_UNLOCK flag on systems without hardware support for shared data may cause severe performance degradation.

See Also

sdenter(S), sdgetv(S)

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

setbuf – Assigns buffering to a stream.

Syntax

```
#include <stdio.h>

setbuf (stream, buf)
FILE *stream;
char *buf;
```

Description

Setbuf is used after a stream has been opened but before it is read or written. It causes the character array *buf* to be used instead of an automatically allocated buffer. If *buf* is the constant pointer NULL, input/output will be completely unbuffered.

A manifest constant BUFSIZ tells how big an array is needed:

```
char buf[BUFSIZ];
```

A buffer is normally obtained from *malloc*(S) upon the first *getc*(S) or *putc*(S) on the file, except that output streams directed to terminals, and the standard error stream stderr are normally not buffered.

A common source of error is allocation of buffer space as an "automatic" variable in a code block, and then failing to close the stream in the same block.

See Also

fopen(S), getc(S), malloc(S), putc(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

setpgrp – Sets process group ID.

Syntax

int setpgrp ( )

Description

Setpgrp sets the process group ID of the calling process to the process ID of the calling process and returns the new process group ID.

Return Value

Setpgrp returns the value of the new process group ID.

See Also

exec(S), fork(S), getpid(S), intro(S), kill(S), 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

shutdn – Flushes block I/O and halts the CPU.

Syntax

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

shutdn (sblk)
struct filsys *sblk;
```

Description

Shutdn causes all information in core memory that should be on disk to be written out. This includes modified super-blocks, modified inodes, and delayed block I/O. The super-blocks of all writable file systems are flagged 'clean', so that they can be remounted without cleaning when XENIX is rebooted. *Shutdn* then prints "Normal System Shutdown" on the console and halts the CPU.

If *sblk* is nonzero, it specifies the address of a super-block which will be written to the root device as the last I/O before the halt. This facility is provided to allow file system repair programs to supercede the system's copy of the root super-block with one of their own.

Shutdn locks out all other processes while it is doing its work. However, it is recommended that user processes be killed off (see *kill*(S)) before calling *shutdn* as some types of disk activity could cause file systems to not be flagged "clean".

The caller must be the super-user.

See Also

fsck(C), haltsys(C), shutdown(C), mount(S), kill(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:

1. All of the receiving process' open file descriptors will be closed.

2. If the parent process of the receiving process is executing a *wait*, it will be notified of the termination of the receiving process and the terminating signal's number will be made available to the parent process; see *wait*(S).

3. If the parent process of the receiving process is not executing a *wait*, the receiving process will be transformed into a zombie process (see *exit*(S) for definition of zombie process).

4. The parent process ID of each of the receiving process' existing child processes and zombie processes will be set to 1. This means the initialization process (see *intro*(S)) inherits each of these processes.

5. An accounting record will be written on the accounting file if the system's accounting routine is enabled; see *acct*(S).

6. If the receiving process' process ID, tty group ID, and process group ID are equal, the signal **SIGHUP** will be sent to all of the processes that have a process group ID equal to the process group ID of the receiving process.

7. A "core image" will be made in the current working directory of the receiving process if *sig* is one for which an asterisk appears in the above list *and* the following conditions are met:

   - The effective user ID and the real user ID of the receiving process are equal.

   - An ordinary filenamed **core** exists and is writable or can be created. If the file must be created, it will have a mode of 0666 modified by the file creation mask (see *umask*(S)), a file owner ID that is the same as the effective user ID of the receiving process, a file group ID that is the same as the effective group ID of the receiving process

The SIG_IGN value causes the process to ignore a signal. The signal *sig* is to be ignored. Note that the signal **SIGKILL** cannot be ignored.

A *function address* value causes to process to catch a signal. Upon receipt of the signal *sig*, the receiving process is to execute the signal-catching function pointed to by *func*. The signal number *sig* will be passed as the only argument to the signal-catching function. There are the following consequences:

1. Upon return from the signal-catching function, the receiving process will resume execution at the point it was interrupted and the value of *func* for the caught signal will be set to SIG_DFL unless the signal is **SIGILL**, **SIGTRAP**, **SIGCLD**, or **SIGPWR**.

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.

SIG_IGN - ignore signal

The signal is to be ignored. Also, if *sig* is SIGCLD, the calling process' child processes will not create zombie processes when they terminate; see *exit*(S).

function address - catch signal

If the signal is SIGPWR, the action to be taken is the same as that described above for *func* equal to *function address*. The same is true if the signal is SIGCLD except, that while the process is executing the signal-catching function any received SIGCLD signals will be queued and the signal-catching function will be continually reentered until the queue is empty.

The SIGCLD affects two other system calls (*wait*(S), and *exit*(S)) in the following ways:

wait If the *func* value of SIGCLD is set to SIG_IGN and a *wait* is executed, the *wait* will block until all of the calling process' child processes terminate; it will then return a value of − 1 with *errno* set to ECHILD.

exit If in the exiting process' parent process the *func* value of SIGCLD is set to SIG_IGN, the exiting process will not create a zombie process.

When processing a pipeline, the shell makes the last process in the pipeline the parent of the proceeding processes. A process that may be piped into in this manner (and thus become the parent of other processes) should take care not to set SIGCLD to be caught.

Notes

The defined constant NSIG in signal.h standing for the number of signals is always at least one greater than the actual number.

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

sinh, cosh, tanh – Performs hyperbolic functions.

Syntax

```
#include <math.h>

double sinh (x)
double x;

double cosh (x)
double x;

double tanh (x)
double x;
```

Description

These functions compute the designated hyperbolic functions for real arguments.

Diagnostics

Sinh and *cosh* return a huge value of appropriate sign when the correct value would overflow.

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)

Name

ssignal, gsignal – Implements software signals.

Syntax

```
#include <signal.h>

int (*ssignal (sig, action))( )
int sig, (*action)( );

int gsignal (sig)
int sig;
```

Description

Ssignal and *gsignal* implement a software facility similar to *signal*(S). This facility is used by the standard C library to enable the user to indicate the disposition of error conditions, and is also made available to the user for his own purposes.

Software signals made available to users are associated with integers in the inclusive range 1 through 15. An *action* for a software signal is *established* by a call to *ssignal*, and a software signal is *raised* by a call to *gsignal*. Raising a software signal causes the action established for that signal to be *taken*.

The first argument to *ssignal* is a number identifying the type of signal for which an action is to be established. The second argument defines the action; it is either the name of a (user defined) *action function* or one of the manifest constants SIG_DFL (default) or SIG_IGN (ignore). *Ssignal* returns the action previously established for that signal type; if no action has been established or the signal number is illegal, *ssignal* returns SIG_DFL.

Gsignal raises the signal identified by its argument, *sig*:

> If an action function has been established for *sig*, then that action is reset to SIG_DFL and the action function is entered with argument *sig*. *Gsignal* returns the value returned to it by the action function.
>
> If the action for *sig* is SIG_IGN , *gsignal* returns the value 1 and takes no other action.
>
> If the action for *sig* is SIG_DFL , *gsignal* returns the value 0 and takes no other action.
>
> If *sig* has an illegal value or no action was ever specified for *sig*, *gsignal* returns the value 0 and takes no other action.

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.

Name

stat, fstat – Gets file status.

Syntax

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

int stat (path, buf)
char *path;
struct stat *buf;

int fstat (fildes, buf)
int fildes;
struct stat *buf;
```

Description

Path points to a pathname naming a file. Read, write or execute permission of the named file is not required, but all directories listed in the pathname leading to the file must be searchable. *Stat* obtains information about the named file.

Similarly, *fstat* obtains information about an open file known by the file descriptor *fildes*, obtained from a successful *open*, *creat*, *dup*, *fcntl*, or *pipe* system call.

Buf is a pointer to a *stat* structure into which information is placed concerning the file.

The contents of the structure pointed to by *buf* include the following members:

```
ushort  st_mode;    /* File mode; see mknod(S) */
ino_t   st_ino;     /* Inode number */
dev_t   st_dev;     /* ID of device containing */
                    /* a directory entry for this file */
dev_t   st_rdev;    /* ID of device */
                    /* This entry is defined only for */
                    /* special files */
short   st_nlink;   /* Number of links */
ushort  st_uid;     /* User ID of the file's owner */
ushort  st_gid;     /* Group ID of the file's group */
off_t   st_size;    /* File size in bytes */
time_t  st_atime;   /* Time of last access */
time_t  st_mtime;   /* Time of last data modification */
time_t  st_ctime;   /* Time of last file status change */
                    /* Times measured in seconds since */
                    /* 00:00:00 GMT, Jan. 1, 1970 */
```

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)

Name

stdio – Performs standard buffered input and output.

Syntax

```
#include <stdio.h>
FILE *stdin, *stdout, *stderr;
```

Description

The *stdio* library contains an efficient, user-level I/O buffering scheme. The in-line macros *getc*(S) and *putc*(S) handle characters quickly. The macros *getchar*, *putchar*, and the higher-level routines *fgetc*, *fgets*, *fprintf*, *fputc*, *fputs*, *fread*, *fscanf*, *fwrite*, *gets*, *getw*, *printf*, *puts*, *putw*, and *scanf* all use *getc* and *putc*; they can be freely intermixed.

A file with associated buffering is called a "stream" and is declared to be a pointer to a defined type FILE . *Fopen*(S) creates certain descriptive data for a stream and returns a pointer to designate the stream in all further transactions. Normally, there are three open streams with constant pointers declared in the "include" file and associated with the standard open files:

stdin	Standard input file
stdout	Standard output file
stderr	Standard error file

A constant "pointer" NULL designates the null stream.

An integer constant EOF is returned upon end-of-file or error by most integer functions that deal with streams (see the individual descriptions for details).

Any program that uses this package must include the header file of pertinent macro definitions, as follows:

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

Most of the functions and constants mentioned in this section of the manual are declared in that "include" file and are described elsewhere. The constants and the following "functions" are implemented as macros (redeclaration of these names is perilous): *getc*, *getchar*, *putc*, *putchar*, *feof*, *ferror*, and *fileno*.

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

stime – Sets the time.

Syntax

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

time_t stime (tp)
long *tp;
```

Description

Stime sets the system's idea of the time and date. *Tp* points to the value of time as measured in seconds from 00:00:00 GMT January 1, 1970.

Stime will fail if the effective user ID of the calling process 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

time(S)

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;
```

Description

These functions operate on null-terminated strings. They do not check for overflow of any receiving string.

Strcat appends a copy of string *s2* to the end of string *s1*. *Strncat* copies at most *n* characters. Both return a pointer to the null-terminated result.

Strcmp compares its arguments and returns an integer greater than, equal to, or less than 0, according as *s1* is lexicographically greater than, equal to, or less than *s2*. *Strncmp* makes the same comparison but looks at at most *n* characters.

Strcpy copies string *s2* to *s1*, stopping after the null character has been moved. *Strncpy* copies exactly *n* characters, truncating or null-padding *s2*; the target may not be null-terminated if the length of *s2* is *n* or more. Both return *s1*.

Strlen returns the number of nonnull characters in *s*.

Strchr (*strrchr*) returns a pointer to the first (last) occurrence of character *c* in string *s*, or NULL if *c* does not occur in the string. The null character terminating a string is considered to be part of the string.

Strpbrk returns a pointer to the first occurrence in string *s1* of any character from string *s2*, or NULL if no character from *s2* exists in *s1*.

Strspn (*strcspn*) returns the length of the initial segment of string *s1* which consists entirely of characters from (not from) string *s2*.

Strtok considers the string *s1* to consist of a sequence of zero or more text tokens separated by spans of one or more characters from the separator string *s2*. The first call (with pointer *s1* specified) returns a pointer to the first character of the first token, and will have written a NULL character into *s1* immediately following the returned token. Subsequent calls with zero for the first argument, will work through the string *s1* in this way until no tokens remain. The separator string *s2* may be different from call to call. When no token remains in *s1*, a NULL is returned.

Strdup returns a pointer to a duplicate copy of the string pointed to by *s*. The duplicate string is automatically allocated storage using a *malloc*(S) system call. This call allocates the exact number of bytes needed to store the string and its terminating null character.

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

swab – Swaps bytes.

Syntax

```
swab (from, to, nbytes)
char *from, *to;
int nbytes;
```

Description

Swab copies *nbytes* pointed to by *from* to the position pointed to by *to*, exchanging adjacent even and odd bytes. It is useful for transporting binary data between machines that differ in the ordering of bytes. *Nbytes* should be even.

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

system – Executes a shell command.

Syntax

#include <stdio.h>

int system (string)
char *string;

Description

System passes the *string* to a new invocation of a shell (see *sh*(C)). The shell reads and executes the *string* as if it had been typed as a command at a terminal, then returns the exit status of the command to the calling process. The calling process waits until the shell has returned a status before proceeding with execution.

See Also

sh(C), exec(S)

Diagnostics

System stops if it can't execute *sh*(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 write the file */etc/termcap*.

Tgetnum gets the numeric value of capability *id*, returning - 1 if is not given for the terminal. *Tgetflag* returns 1 if the specified capability is present in the terminal's entry, 0 if it is not. *Tgetstr* gets the string value of capability *id*, placing it in the buffer at *area*, advancing the *area* pointer. It decodes the abbreviations for this field described in *termcap*(M), except for cursor addressing and padding information.

Tgoto returns a cursor addressing string decoded from *cm* to go to column *destcol* in line *destline*. It uses the external variables UP (from the up capability) and BC (if bc is given rather than bs) if necessary to avoid placing \n, CNTRL-D or NULL in the returned string. (Programs which call *tgoto* should be sure to turn off the TAB3 bit (see *tty*(M)), since *tgoto* may now output a tab. Note that programs using termcap should in general turn off TAB3 anyway since some terminals use CNTRL-I for other functions, such as nondestructive space.) If a %sequence is given which is not understood, then *tgoto* returns "OOPS".

Tputs decodes the leading padding information of the string *cp; affcnt* gives the number of lines affected by the operation, or 1 if this is not applicable, *outc* is a routine which is called with each character in turn. The external variable *ospeed* should contain the output speed of the terminal as encoded by *stty*(C). The external variable PC should contain a pad character to be used (from the pc capability) if a NULL is inappropriate.

Files

/usr/lib/libtermcap.a - ltermcap library
/etc/termcap data base

See Also

curses(S), termcap(M), tty(M)

Credit

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

Notes

These routines can be linked by using the linker option - ltermcap.

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;
};
```

The structure contains the time since the epoch in seconds, up to 1000 milliseconds of more-precise interval, the local time zone (measured in minutes of time westward from Greenwich), and a flag that, if nonzero, indicates that Daylight Saving time applies locally during the appropriate part of the year.

See Also

date(C), stime(S), ctime(S)

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

tmpfile – Creates a temporary file.

Syntax

#include <stdio.h>

FILE *tmpfile ( )

Description

Tmpfile creates a temporary file and returns a corresponding FILE pointer. Arrangements are made so that the file will automatically be deleted when the process using it terminates. The file is opened for update.

See Also

creat(S), unlink(S), fopen(S), mktemp(S), tmpnam(S)

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

sin, cos, tan, asin, acos, atan, atan2 – Performs trigonometric functions.

Syntax

```
#include <math.h>

double sin (x)
double x;

double cos (x)
double x;

double asin (x)
double x;

double acos (x)
double x;

double atan (x)
double x;

double atan2 (y, x)
double x, y;
```

Description

Sin, *cos* and *tan* return trigonometric functions of radian arguments. The magnitude of the argument should be checked by the caller to make sure the result is meaningful.

Asin returns the arc sin in the range $-\pi/2$ to $\pi/2$.

Acos returns the arc cosine in the range 0 to π.

Atan returns the arc tangent of *x* in the range $-\pi/2$ to $\pi/2$.

Atan2 returns the arc tangent of *y/x* in the range $-\pi$ to π.

Diagnostics

Arguments of magnitude greater than 1 cause *asin* and *acos* to return value 0.

Notes

These routines can be linked with the linker option – lm.

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

ulimit – Gets and sets user limits.

Syntax

long ulimit (cmd, newlimit)
int cmd;
long newlimit;

Description

This function provides for control over process limits. The *cmd* values available are:

1 Gets the process' file size limit. The limit is in units of disk blocks and is inherited by child processes. Files of any size can be read.

2 Sets the process' file size limit to the value of *newlimit*. Any process may decrease this limit, but only a process with an effective user ID of super-user may increase the limit. *Ulimit* will fail and the limit will be unchanged if a process with an effective user ID other than super-user attempts to increase its file size limit. [EPERM]

3 Gets the maximum possible break value. See *sbrk*(S).

Return Value

Upon successful completion, a nonnegative value is returned. Otherwise, a value of – 1 is returned and *errno* is set to indicate the error.

See Also

sbrk(S), chsize(S), write(S)

Notes

The file limit is only enforced on writes to regular files. Tapes, disks, and other devices of any size can be written.

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

umount – Unmounts a file system.

Syntax

```
int umount (spec)
char *spec;
```

Description

Umount requests that a previously mounted file system contained on the block special device identified by *spec* be unmounted. *Spec* is a pointer to a pathname. After unmounting the file system, the directory upon which the file system was mounted reverts to its ordinary interpretation.

Umount may be invoked only by the super-user.

Umount will fail if one or more of the following are true:

The process' effective user ID is not super-user. [EPERM]

Spec does not exist. [ENXIO]

Spec is not a block special device. [ENOTBLK]

Spec is not mounted. [EINVAL]

A file on *spec* is busy. [EBUSY]

Spec 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

mount(C), mount(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 *sysname*. 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.

See Also

uname(C)

Notes

Not all fields may be set on a particular system.

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.

Name

unlink – Removes directory entry.

Syntax

```
int unlink (path)
char *path;
```

Description

Unlink removes the directory entry named by the pathname pointed to by *path.*

The named file is unlinked unless 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]

Write permission is denied on the directory containing the link to be removed. [EACCES]

The named file is a directory and the effective user ID of the process is not super-user. [EACCES]

The entry to be unlinked is the mount point for a mounted file system. [EBUSY]

The entry to be unlinked is "." or ".." in the root directory of a mounted filesystem. [EBUSY]

The entry to be unlinked is the last link to a pure procedure (shared text) file that is being executed. [ETXTBSY]

The directory entry to be unlinked is part of a read-only file system. [EROFS]

Path points outside the process' allocated address space. [EFAULT]

When all links to a file have been removed and no process has the file open, the space occupied by the file is freed and the file ceases to exist. If one or more processes have the file open when the last link is removed, the removal is postponed until all references to the file have been closed.

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

ustat – Gets file system statistics.

Syntax

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

int ustat (dev, buf)
int dev;
struct ustat *buf;
```

Description

Ustat returns information about a mounted file system. *Dev* is a device number identifying a device containing a mounted file system. *Buf* is a pointer to a *ustat* structure that includes the following elements:

```
daddr_t f_tfree;          /* Total free blocks */
ino_t   f_tinode;         /* Number of free inodes */
char    f_fname[6];       /* Filsys name */
char    f_fpack[6];       /* Filsys pack name */
```

Ustat will fail if one or more of the following are true:

Dev is not the device number of a device containing a mounted file system. [EINVAL]

Buf 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

stat(S), filesystem(F)

Notes

When using file systems from previous versions of XENIX, *fsck*(C) must be run on the file system before mounting. Otherwise the *ustat* system call will not work correctly. This only needs to be done once.

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]

Times is not NULL and points outside the process' allocated address space. [EFAULT]

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

stat(S)

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).

Wait will fail and return immediately if one or more of the following are true:

The calling process has no existing unwaited-for child processes. [ECHILD]

Stat_loc points to an illegal address. [EFAULT]

Return Value

If *wait* returns due to the receipt of a signal, a value of – 1 is returned to the calling process and *errno* is set to EINTR. If *wait* returns due to a stopped or terminated child process, the process ID of the child is returned to the calling process. Otherwise, a value of – 1 is returned and *errno* is set to indicate the error.

See Also

exec(S), exit(S), fork(S), pause(S), signal(S)

Warning

See *Warning* in *signal*(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.

Name

write – Writes to a file.

Syntax

```
int write (fildes, buf, nbyte)
int fildes;
char *buf;
unsigned nbyte;
```

Description

Fildes is a file descriptor obtained from a *creat*, *open*, *dup*, *fcntl*, or *pipe* system call.

Write attempts to write *nbyte* bytes from the buffer pointed to by *buf* to the file associated with the *fildes*.

On devices capable of seeking, the actual writing of data proceeds from the position in the file indicated by the file pointer. Upon return from *write*, the file pointer is incremented by the number of bytes actually written.

On devices incapable of seeking, writing always takes place starting at the current position. The value of a file pointer associated with such a device is undefined.

If the O_APPEND flag of the file status flags is set, the file pointer will be set to the end of the file prior to each write.

Write will fail and the file pointer will remain unchanged if one or more of the following are true:

Fildes is not a valid file descriptor open for writing. [EBADF]

An attempt is made to write to a pipe that is not open for reading by any process. [EPIPE and SIGPIPE signal]

An attempt was made to write a file that exceeds the process' file size limit or the maximum file size. See *ulimit*(S). [EFBIG]

Buf points outside the process' allocated address space. [EFAULT]

If a *write* requests that more bytes be written than there is room for (e.g., the *ulimit* (see *ulimit*(S)) or the physical end of a medium), only as many bytes as there is room for will be written. For example, suppose there is space for 20 bytes more in a file before reaching a limit. A write of 512 bytes will return 20. The next write of a nonzero number of bytes will give a failure return (except as noted 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.

Name

xlist, fxlist – Gets name list entries from files.

Syntax

```
#include <a.out.h>
xlist(filename, xl)
char *filename;
struct xlist xl[ ];

#include <a.out.h>
#include <stdio.h>
fxlist(fp, xl)
FILE *fp;
struct xlist xl[ ];
```

Description

Fxlist performs the same function as *xlist*, except that *fxlist* accepts a pointer to a previously opened file intead of a filename.

Xlist examines the name list in the given executable output file and selectively extracts a list of values. The name list structure *xl* consists of an array of *xlist* structures containing names, types, values, and segment values (if applicable). The list is terminated by either a pointer to a null name or a null pointer. 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 into the next two fields. The segment value (if it exists) is inserted in the third field. If the name is not found, both entries are set to zero. See *a.out*(F) for a discussion of the xlist structure.

X.out and a.out formats are understood, as well as 8086 relocatable and x.out segmented formats.

If the symbol table is in a.out format, and if the symbol name given to *xlist* is longer than eight characters, only the first eight characters are used for comparison. In all other cases, the name given to *xlist* must be the same length as a name list entry in order to match.

If two or more symbols happen to match the name given to *xlist,* then the type and value used will be those of the last symbol found.

See Also

a.out(F)

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

a.out – Format of assembler and link editor output.

Description

A.out is the output file of the assembler *as* and the link editor *ld*. Both programs will make a.out executable if there were no errors in assembling or linking, and no unresolved external references.

The format of a.out, called the x.out or segmented x.out format, is defined by the files */usr/include/a.out.h* and */usr/include/sys/relsym.h*. The a.out file has the following general layout:

1. Header.

2. Extended header.

3. File segment table (for segmented formats).

4. Segments (Text, Data, Symbol, and Relocation).

In the segmented format, there may be several text and data segments, depending on the memory model of the program. Segments within the file begin on boundaries which are multiplies of 512 bytes as defined by the file's pagesize.

See Also

as(CP), ld(CP), nm(CP), strip(CP).

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

ar – Archive file format.

Description

The archive command *ar* is used to combine several files into one. Archives are used mainly as libraries to be searched by the link editor *ld*(C).

A file produced by *ar* has a magic number at the start, followed by the constituent files, each preceded by a file header. The magic number is 0177545 octal (or 0xff65 hexadecimal). The header of each file is declared in **/usr/include/ar.h**.

Each file begins on a word boundary; a null byte is inserted between files if necessary. Nevertheless the size given reflects the actual size of the file exclusive of padding.

Notice there is no provision for empty areas in an archive file.

See Also

ar(CP), ld(CP)

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

core – Format of core image file.

Description

XENIX writes out a core image of a terminated process when any of various errors occur. See *signal*(S) for the list of reasons; the most common are memory violations, illegal instructions, bus errors, and user-generated quit signals. The core image is called **core** and is written in the process' working directory (provided it can be; normal access controls apply). A process with an effective user ID different from the real user ID will not produce a core image.

The first section of the core image is a copy of the system's per-user data for the process, including the registers as they were at the time of the fault. The size of this section depends on the parameter *usize*, which is defined in **/usr/include/sys/param.h**. The remainder represents the actual contents of the user's core area when the core image was written. If the text segment is read-only and shared, or separated from data space, it is not dumped.

The format of the information in the first section is described by the *user* structure of the system, defined in **/usr/include/sys/user.h**. The locations of registers, are outlined in **/usr/include/sys/reg.h.**

See Also

adb(CP), setuid(S), signal(S)

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%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

dir – Format of a directory.

Syntax

#include <sys/dir.h>

Description

A directory behaves exactly like an ordinary file, except that no user may write into a directory. The fact that a file is a directory is indicated by a bit in the flag word of its inode entry (see ***filesystem***(F)). The structure of a directory is given in the include file /usr/include/sys/dir.h.

By convention, the first two entries in each directory are "dot" (.) and "dotdot" (..). The first is an entry for the directory itself. The second is for the parent directory. The meaning of dotdot is modified for the root directory of the master file system; there is no parent, so dotdot has the same meaning as dot.

See Also

filesystem(F)

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.

The fields of the header structure are as follows:

c_type	The type of the header.
c_date	The date the dump was taken.
c_ddate	The date the file system was dumped from.
c_volume	The current volume number of the dump.
c_tapea	The current block number of this record. This is counting 512 byte blocks.
c_inumber	The number of the inode being dumped if this is of type TS_INODE.
c_magic	This contains the value MAGIC above, truncated as needed.
c_checksum	This contains whatever value is needed to make the block sum to CHECKSUM.
c_dinode	This is a copy of the inode as it appears on the file system.
c_count	This is the count of characters following that describe the file. A character is zero if the block associated with that character was not present on the file system, otherwise the character is nonzero. If the block was not present on the file system no block was dumped and it is replaced as a hole in the file. If there is not sufficient space in this block to describe all of the blocks in a file, TS_ADDR blocks will be scattered through the file, each one picking up where the last left off.
c_addr	This is the array of characters that is used as described above.

Each volume except the last ends with a tapemark (read as an end of file). The last volume ends with a TS_END block and then the tapemark.

The structure idates describes an entry of the file where dump history is kept.

See Also

dump(C), restor(C), filesystem(F)

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 try again. To free an inode, provided *s_ninode* is less than 100, place its number into *s_inode*[*s_ninode*] and increment *s_ninode*. If *s_ninode* is already 100, do not bother to enter the freed inode into any table. This list of inodes only speeds up the allocation process. The information about whether the inode is really free is maintained in the inode itself.

S_tinode is the total free inodes available in the file system.

S_flock and *s_ilock* are flags maintained in the core copy of the file system while it is mounted and their values on disk are immaterial. The value of *s_fmod* on disk is also immaterial, and is used as a flag to indicate that the super-block has changed and should be copied to the disk during the next periodic update of file system information.

S_ronly is a read-only flag to indicate write-protection.

S_time is the last time the super-block of the file system was changed, and is a double-precision representation of the number of seconds that have elapsed since 00:00 Jan. 1, 1970 (GMT). During a reboot, the *s_time* of the super-block for the root file system is used to set the system's idea of the time.

I-numbers begin at 1, and the storage for inodes begins in block 2. Also, inodes are 64 bytes long, so 8 of them fit into a block. Therefore, inode i is located in block $(i+15)/8$, and begins $64 \times ((i+15) \pmod 8)$ bytes from its start. Inode 1 is reserved for future use. Inode 2 is reserved for the root directory of the file system, but no other i-number has a built-in meaning. Each inode represents one file. For the format of an inode and its flags, see *inode*(F).

Files

/usr/include/sys/filsys.h

/usr/include/sys/stat.h

See Also

fsck(C), mkfs(C), inode(F)

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)

Name

master – master device information table

Description

This file is used by the *config*(CP) program to obtain device information that enables it to generate the configuration files. The file consists of 4 parts, each separated by a line with a dollar sign ($) in column 1. Part 1 contains device information; part 2 contains the line discipline table; part 3 contains names of devices that have aliases; part 4 contains tunable parameter information. Any line with an asterisk (*) in column 1 is treated as a comment.

Part 1 contains lines consisting of 14 fields with the fields delimited by tabs and/or blanks:

Field 1: device name (8 chars. maximum).
Field 2: interrupt vector size (decimal, in bytes).
Field 3: device mask (octal)– each "on" bit indicates that the driver has the corresponding handler or structure:

- 000400 tty structure
- 000200 stop handler
- 000100 not used
- 000040 not used
- 000020 open handler
- 000010 close handler
- 000004 read handler
- 000002 write handler
- 000001 ioctl handler.

Field 4: device type indicator (octal):

- 000200 allow only one of these devices
- 000100 not used
- 000040 not used
- 000020 required device
- 000010 block device
- 000004 character device
- 000002 not used
- 000001 not used.

Field 5: handler prefix (4 chars. maximum).
Field 6: not used.
Field 7: major device number for block-type device.
Field 8: major device number for character-type device.
Field 9: maximum number of devices per controller (decimal).
Field 10: not used.
Fields 11-14: maximum of four interrupt vector addresses. Each address is followed by a unique letter or a blank.

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

mnttab – Format of mounted file system table.

Syntax

#include <stdio.h>
#include <mnttab.h>

Description

The */etc/mnttab* file contains a table of devices mounted by the *mount*(C) command.

Each table entry contains the pathname of the directory on which the device is mounted, the name of the device special file, the read/write permissions of the special file, and the date on which the device was mounted.

The maximum number of entries in *mnttab* is based on the system parameter NMOUNT located in /usr/sys/conf/c.c, which defines the number of allowable mounted special files.

See Also

mount(C)

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
```

The first line (@s) contains the number of lines inserted/deleted/unchanged respectively. The second line (@d) contains the type of the delta (currently, normal: D, and removed: R), the SCCS ID of the delta, the date and time of creation of the delta, the login name corresponding to the real user ID at the time the delta was created, and the serial numbers of the delta and its predecessor, respectively.

The @i, @x, and @g lines contain the serial numbers of deltas included, excluded, and ignored, respectively. These lines are optional.

The @m lines (optional) each contain one MR number associated with the delta; the @c lines contain comments associated with the delta.

The @e line ends the delta table entry.

User Names

The list of login names and/or numerical group IDs of users who may add deltas to the file, separated by new-lines. The lines containing these login names and/or numerical group IDs are surrounded by the bracketing lines @u and @U. An empty list allows anyone to make a delta.

Flags

Keywords used internally (see *admin*(CP) for more information on their use). Each flag line takes the form:

@f <flag> <optional text>

The following flags are defined:

@f t	<type of program>
@f v	<program name>
@f i	
@f b	
@f m	<module name>
@f f	<floor>
@f c	<ceiling>
@f d	<default-sid>
@f n	
@f j	
@f l	<lock-releases>
@f q	<user defined>

The t flag defines the replacement for the identification keyword. The v flag controls prompting for MR numbers in addition to comments; if the optional text is present it defines an MR number 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

types - Primitive system data types.

Syntax

#include <sys/types.h>

Description

The data types defined in the include file <sys/types.h> are used in XENIX system code; some data of these types are accessible to user code.

The form *daddr_t* is used for disk addresses except in an inode on disk, see *filesystem*(F). Times are encoded in seconds since 00:00:00 GMT, January 1, 1970. The major and minor parts of a device code specify kind and unit number of a device and are installation-dependent. Offsets are measured in bytes from the beginning of a file. The *label_t* variables are used to save the processor state while another process is running.

See Also

filesystem(F)

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 is not loaded.

The layout of a symbol table entry, and the principal flag values that distinguish symbol types, are given in the include file. If a symbol's type is undefined external, and the value field is non-zero, the symbol is interpreted by the loader, **ld**, as the name of a common region whose size is indicated by the value of the symbol. The value of a word in the text or data portions, which is not a reference to an undefined external symbol, is exactly the value that will appear in core when the file is executed. If a word in the text or data portion involves a reference to an undefined external symbol, as indicated by the relocation information for that word, then the value of the word as stored in the file is an offset from the associated external symbol.

When the file is processed by the loader and the external symbol becomes defined, the value of the symbol will be added into the word in the file. If relocation information is present, it amounts to one word per word of program text or initialized data.

Files

/usr/include/a.out.h

Notes

See also **as(CP), ld(CP), nm(CP),** /usr/include/a.out.h.