Programmer's Guide to Library Functions

IBM Personal Computer XENIX™ Software Development System

Programming Family

IBM

Personal
Computer
Software

6138873

Programmer's Guide to Library Functions

IBM Personal Computer XENIX™ Software Development System

Programming Family

Personal
Computer
Software

First Edition (December 1984)

IBM Personal Computer XENIX Library Overview

The XENIX[1] System has three available products. They are the:

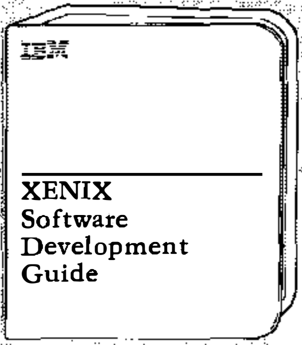
- Operating System
- Software Development System
- Text Formatting System

The following pages outline the XENIX Software Development System library.

[1] XENIX is a trademark of Microsoft Corporation.

XENIX Software Development System

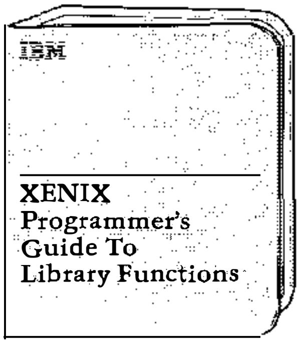
For C Language Users



- Creating language programs
- Invoking the C Compiler
- Program checkers, maintainers, and debuggers
- Using S-Files
- The C-Shell

A guide to the available programming tools in the XENIX environment.

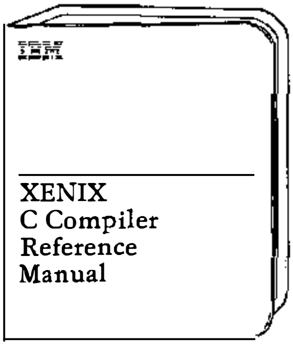
For All Users



- Using stream functions
- Screen processing
- Process controls
- Creating and using pipes
- Using signals and system resources

A reference to system calls, subroutines, and file formats. Use with the XENIX Software Command Reference.

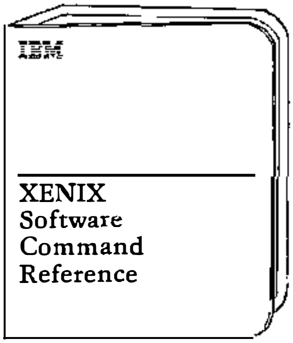
For Experienced Language Users



- Elements of the C programming language
- Preprocessor Directives
- Declarations
- Expressions and Assignments
- Description of functions and statements

A reference to the C programming language. Notational conventions are described throughout the manual.

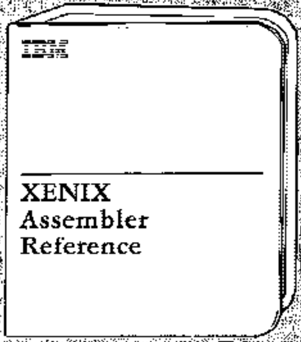
For All Users



- Software Development commands (CP)
- Command definition and syntax
- System calls and subroutines (S)
- System call and library function cross reference

A reference to Software Development System commands. Describes system services in the Operating System kernel.

For Assembler Users



- Assembly Language format
- Operands and Operators
- Directives
- Segment usage
- Machine instructions
- Assembler messages

A reference for programmers who use the IBM Personal Computer XENIX Assembler.

About This Book

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
- The screen updating and cursor movement library, called *curses.*

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.

If you have never used the C library functions before, read this manual first. Then see the IBM Personal Computer *XENIX Software Command Reference* if you need more information.

If you are familiar with the library functions, turn to the IBM Personal Computer *XENIX Software Command Reference* to see how these functions differ from the ones you already know; return to this manual for examples of the functions.

This book is organized as follows:

Chapter 1. Introduction
Gives an overview of the C language libraries and introduces the notational conventions used in this manual.

Chapter 2. Using the Standard I/O Functions
Describes the standard input and output functions. These functions enable a program to read and write to the files of a XENIX file system.

Chapter 3. Screen Processing
Describes the screen processing functions. These functions enable a program to use the screen processing facilities of a user's terminal.

Chapter 4. Character and String Processing
Describes the character and string processing functions. These functions enable a program to assign, manipulate, and compare characters and strings.

Chapter 5. Using Process Control
Describes the process control functions. These functions enable a program to execute other programs and create multiple copies of itself.

Chapter 6. Creating and Using Pipes
Describes the pipe functions. These functions enable programs to communicate with one another without resorting to the creation of temporary files.

Chapter 7. Using Signals
Describes the signal functions. These functions enable a program to process signals that are normally processed by the system.

Chapter 8. Using System Resources
Describes system resource functions. These functions enable a program to dynamically allocate memory, share memory with other programs, lock files against access by other programs, and use semaphores.

Chapter 9. Error Processing
Describes the error processing functions. These functions enable a program to process errors while accessing the file system or allocating memory.

Appendix A. Assembly Language Interface
Describes the assembly language interface with C programs and explains the calling and return value conventions of C functions.

Appendix B. XENIX System Calls
Explains how to create and use new XENIX system calls.

Appendix C. A Common Library for XENIX and DOS
Lists the XENIX library routines that form the Common C Library for the XENIX and DOS versions of the Microsoft® C Compiler.

Related XENIX Publications

- IBM Personal Computer XENIX Software Development Guide
- IBM Personal Computer XENIX C Compiler Reference
- IBM Personal Computer XENIX Software Command Reference
- IBM Personal Computer XENIX Assembler Reference
- IBM Personal Computer XENIX Installation Guide
- IBM Personal Computer XENIX Visual Shell
- IBM Personal Computer XENIX System Administration
- IBM Personal Computer XENIX Basic Operations Guide
- IBM Personal Computer XENIX Command Reference

[1] Microsoft is a registered trademark of Microsoft Corporation.

Contents

Chapter 1. Introduction

Contents

Using the C Library Functions

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 line printers, load and execute other programs, receive and process signals, communicate with other programs through pipes, share system resources, and process errors.

To use the C library functions, you must include the proper function call and definitions in the program and specify the corresponding library 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 "cc: a C Compiler" in the IBM Personal Computer *XENIX Software Development Guide*).

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 are 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

Contents

Introduction

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

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

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.c*. This file is automatically read whenever you compile a program, so no special argument is needed when you invoke the compiler.

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 special names are:

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 512) suitable for an I/O buffer supplied by the user.

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.

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 (that is, a string ending with a null character, \Ø). The first string (at *argv*[Ø]) is the program name. The argument count is never less than 1, because 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, because the system automatically removes leading and trailing whitespace characters (that is, 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 (for example, <, >, |, 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.

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. Because 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 "Using the Stream Functions" section later in this chapter.

The XENIX system lets you redirect the standard input and output using the shell's redirection symbols (> and <). 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 explain how to read from and write to the standard input and output. It also explains how to redirect the standard input and output.

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. Int type implies that the variable listed is an integer. The function normally returns the character read, but returns the end-of-file value EOF if the end of the file or an error is encountered.

The *getchar* function is 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);
}
```

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 (\n), then replaces the newline character with a null character (\Ø), 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 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".

The *gets* function cannot check the length of the string; therefore, an 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 IBM Personal Computer *XENIX Software Command Reference.*) 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 that make up a value. It then converts the characters, if necessary, into the appropriate string or number.

The *scanf* function is used whenever formatted input is required, that is, input that must be typed in a special way or that 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*. Observe 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:

```
Jeni 22
F
```

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* detects 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.

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 Standard Screen" in Chapter 3.

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 returns the value EOF if an error is encountered.

The function in a conditional loop writes a string of characters to the standard output. For example, the function:

```
write(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 can be any number of characters, but must end with a null character (\Ø). 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 used when writing full lines to the standard output. For example, the following program fragment writes one of three strings to the standard output:

```
int 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 that 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 IBM Personal Computer *XENIX Software Command Reference.*) If a string is requested, the corresponding *arg* must be a pointer. The function normally returns zero, but returns a nonzero value if an error is encountered.

The *printf* function is typically used when formatted output is required, that is, 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.

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. If the file does not exist, the shell displays an error message and stops the program.

Whenever *getchar*, *gets*, or *scanf* 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.

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* not the terminal screen as the standard output of the program *dial*. 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.

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.

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."

Program Example

This section shows how to 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. This program displays text or data files that contain characters that can 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.

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
```

To save the stripped files, 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
```

The *exit* function at the end of the program ensures that any program that executes the *ccstrip* program receives a normal termination status (0) from the program when it ends. An explanation of the *exit* function and how to execute one program under control of another is given in Chapter 5 in the section "Stopping a Program."

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.

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 refers to that file until the file is explicitly closed with the *fclose* function.

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 uses these pointers to read and write from the standard files without first using the *fopen* function to open them.

The predefined file pointers are 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.

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, in subsequent stream functions, reads from or writes 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 can 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 added 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 is encountered upon opening the file. Check for the NULL value after each call to the function to prevent reading or writing after an error.

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 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* is defined 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.

Reading a String from a File

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

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

where *s* is 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 the first newline character, whichever occurs first. The function appends a null character (\Ø) 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 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.

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 the end of the file or an error is encountered.

The function is 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);
```

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

(See "Testing for the End of a File" and "Testing for File Errors.")

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 IBM Personal Computer *XENIX Software Command Reference*.) The function normally returns the number of arguments it read but returns the value EOF if the end of the file or an error is encountered.

The function reads 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.

Writing a Single Character

The *putc* and *fputc* functions write single characters to a given file. The putc 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 returns the value EOF if an error is encountered.

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

These functions are 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 restriction of macro processing, that is, the arguments of macro functions should not contain any side effects such as function calls or increment or decrement operators.

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 copies 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 returns EOF if an error is encountered.

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 that 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 IBM Personal Computer *XENIX Software Command Reference.*) 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 writes 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.

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 the end of the file or an error is encountered.

The function writes 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.

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

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 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));
```

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 tests for errors before performing 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 an error is detected may cause undesirable results.

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 be closed. The function normally returns 0 but returns -1 if an error is encountered.

The *fclose* function frees 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);
```

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.

Program Example

This section shows how you may use the stream functions you have seen so far to perform useful tasks. The following program counts the characters, words, and lines found in one or more files, and 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 to 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;
    }
```

(Example continues on next page.)

```
      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 decimal numbers with no more than seven 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.

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.

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 delays when transferring the input and output. When a file does have a buffer, the buffer size in bytes is given by the manifest 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.

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 reading, 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.

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 creates a buffer for the standard output when it is assigned to the user's terminal, which improves 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 to 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.

Call the *setbuf* function immediately after opening a file and before reading or writing to it. Furthermore, use the *fclose* or *fflush* function to flush the buffer before terminating the program. If one of these functions is not used, some data written to the buffer may not be written to the file.

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 returns the value EOF if an error is encountered.

The function scans 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.

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.

The value EOF must never be put back in the buffer.

Flushing a File Buffer

The *fflush* function empties the buffer of a given 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 returns the value EOF if an error is encountered.

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

```
FILE *outtty;
int errflag;

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

The *fflush* function 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.

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 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.

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, unlike a file pointer, does not point to any specific structure. Instead, the descriptor is used internally by the system to access the necessary information. Because 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 can use the predefined file descriptors without explicitly opening the associated files.

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 directed to the terminal.

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 IBM Personal Computer *XENIX Software Command Reference*.) 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 opens the existing file named */usr/accounts* for reading and opens 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 a set of permissions associated with it, that is, 9 bits of protection information that control who can read, write, and execute the file. These 9 bits consist of 3 bits that determine what permissions the owner of the file has, 3 bits that determine what permissions the owner's group has, and 3 bits that determine what permissions all others have. A 3-digit octal number is the most convenient way to specify the permissions. In the example above, the octal number 0754 specifies read, write, and execute permission for the owner, read and execute permission for the group, and read for everyone else. (Permissions are described in the IBM Personal Computer *XENIX Basic Operations Guide* in the section "Using File and Directory Permissions".)

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

Reading Bytes from a File

The *read* function reads 1 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 returns 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 512, which corresponds to the physical block size on many peripheral devices.

Writing Bytes to a File

The *write* function writes 1 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.

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 returns -1 if an error is encountered.

The function closes 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);
```

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.

Program Examples

This section shows how to use the low-level functions to perform useful tasks and 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* that 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 constant 0377.

The third 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];
```

(Example continues on next page.)

```
        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 open simultaneously. Therefore, any program that intends to process many files must be prepared to reuse file descriptors by closing unneeded files.

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.

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 begins at that position. The value of *offset* must be a long integer, and *fd* and *origin* must be integers.

The function can be used to move the character pointer to the end of a file to allow appending, or it can be used to reset the indicated file back to the beginning of the file 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* allows you to treat files more or less like large arrays, but 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));
}
```

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 (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 can be used on either buffered or unbuffered files.

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.

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 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

Contents

Introduction

This chapter explains how to use the screen updating and cursor movement library named *curses*. The library provides functions to:

- Create screen windows
- Update screen windows
- Get input from the terminal in a screen-oriented way
- Optimize the motion of the cursor on the screen.

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, can be any size (within the limits of the 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. Because 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.

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 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 give 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.

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

Screen Updating

In order to update the screen optimally, it is necessary for the routines to know what the screen currently looks like and what the programmer wants it to look like next. For this purpose, a data type (structure) named WINDOW is defined that describes a window image to the routines, including its starting position on the screen (the (y,x) co-ordinates of the upper-left hand corner) and its size. One of these (called curscr for current screen) is a screen image of what the terminal currently looks like. Another screen (called stdscr for standard screen) is provided by default to make changes on.

A window is a purely internal representation. It is used to build and store a potential image of a portion of the terminal. It doesn't bear any necessary relation to what is really on the terminal screen. It is more like an array of characters on which to make changes.

When a window which describes what some part of the screen should look like, and it is not part of the stdscr, the routine refresh() or wrefresh is called. The refresh() routine makes the terminal, in the area covered by the window, look like that window. Note that changing something on a window does not change the terminal. Actual updates to the terminal screen are made only by calling refresh() or wrefresh(). This allows the programmer to maintain several different ideas of what a portion of the terminal screen should look like. Also, changes can be made to windows in any order, without regard to motion efficiency. Then, at will, the programmer can effectively say "make it look like this," and let the package worry about the best way to do this.

Naming Conventions

As hinted above, the routines can use several windows, but two are automatically given: curscr, which knows what the terminal looks like, and stdscr, which is what the programmer wants the terminal to look like next. The screen package also uses the standard I/O library, so <curses.h>, which includes <stdio.h>. It is redundant, but harmless, for the programmer to do this, too. The user should never really access curscr directly. Changes should be made to the appropriate screen, and then the routine refresh() (or wrefresh()) should be called.

Many functions are set up to deal with stdscr as a default screen. For example, to add a character to stdscr, call addch() with the desired character. If a different window is to be used, the routine waddch() (for window-specific addch()) is provided. Actually, addch() is really a "#define" macro with arguments, as are most of the "functions" that deal with stdscr as a default. This convention of prepending function names with a "w" when they are to be applied to specific windows is consistent. The only routines that do not do this are those to which a window always must be specified.

In order to move the current (y,x) co-ordinates from one point to another, the routines move() and wmove() are provided. However, it is often desirable to first move and then perform some I/O operation. In order to avoid clumsiness, most I/O routines can be preceded by the prefix "mv" and the desired (y,x) co-ordinates then can be added to the arguments to the function. For example, the calls:

```
move (y,x);
addch (ch);
```

can be replaced by

```
mvaddch (y,x,ch);
```

and

```
wmove (win, y, x);
waddch (win, ch);
```

can be replaced by

```
mvwaddch (win, y, x, ch);
```

Note that the window description pointer (win) comes before the added (y, x) co-ordinates. If such pointers are needed, they are always the first parameters passed.

Terminology

In this chapter, the following terminology is used:

Term	**Description**
window	An internal representation containing an image of what a section of the terminal screen may look like at some point in time. This subsection can either encompass the entire terminal screen, or any smaller portion down to a single character within that screen.

Term	Description
terminal	Sometimes called terminal screen. The package's idea of what the terminal's screen currently looks like; in other words, what the user sees now. This is a special screen.
screen	This is a subset of windows which are as large as the terminal screen; in other words, they start at the upper-left hand corner and encompass the lower-right hand corner. One of these, stdscr, is automatically provided for the programmer.

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 names, types, and descriptions of the variables are:

Variables

Name	Type	Description
curscr	WINDOW*	A pointer to the current version of the terminal screen.
stdscr	WINDOW*	A pointer to the default screen used for updating when no explicit screen is defined.
Def_term	char	A pointer to the default terminal type if the type cannot be determined.

Name	Type	Description
My__term	bool	The terminal type flag. If set, it causes the terminal specification in Def__term to be used, regardless of the real terminal type.
ttytype	char	A pointer to the full name of the current terminal.
LINES	int	The number of lines on the terminal.
COLS	int	The number of columns on the terminal.
ERR	boolean false value (0)	The error flag. Returned by functions on an error.
OK	boolean true value (1)	The okay flag. Returned by functions on successful operation.

The names and descriptions of the types and constants are:

Types and Constants

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

Name	Description
TRUE	The boolean true value (1).
WINDOW	A structure of short integers, characters and pointers. This represents current, maximum, and beginning x and y positions, clear, leave and scroll toggles, and three pointers to first and last characters and the y pointer.

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.

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.

Starting Up

In order to use the screen package, the routines must know about terminal characteristics, and the space for curscr and stdscr must be allocated. These functions are performed by initscr(). Since it must allocate space for the windows, it can overflow core when attempting to do so. On this rather rare occasion, initscr() returns ERR. The initscr() function must always be called before any of the routines which affect windows are used. If it is not, the program will core dump as soon as either curscr or stdscr are referenced. However, it is usually best to wait to call it until after you are sure you will need it, like after checking for startup errors. Terminal status changing routines like nl () and crmode () should be called after initscr().

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.

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. For example, HO is a string which moves the cursor to the "home" position. These names are identical to those variables used in the /etc/termcap database to describe each capability. As there are two types of variables involving ttys, there two routines. The first, gettmode (), sets some variables based upon the tty modes accessed by gtty (2). The second, setterm (), a larger task by reading in the descriptions from the /etc/termcap database. This is the way these routines are used by initscr():

```
if (isatty (0) ) {
       gettmode ();
       if (sp=getenv ("TERM") )
               setterm (sp);
}
else
       setterm (Def_term);
_puts (TI);
_puts (VS);
```

The isattyd() function checks to see if file descriptor 0 is a terminal. The isatty () function is defined in the default C library function routines. It does a gtty (2) on the descriptor and checks the return value. If it is, gettmode () sets the terminal description modes from a gtty (2) getenv () is then called to get the name of the terminal, and that value (if there is one) is passed to setterm (), which reads in the variables from /etc/termcap associated with that terminal. (The getenv () function returns a pointer to a string containing the name of the terminal, which we save in the character pointer sp.) If isatty () returns false, the default terminal Def__term is used. The TI and VS sequences initialize the terminal. The __puts () routine is a macro which uses tputs (). It is these things which endwin () undoes.

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. You must do this 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.

Capabilities from termcap

Capabilities from termcap are of three kinds: string valued options, numeric valued options, and boolean options. The string valued options are the most complicated, since they may include padding information, which we describe now.

Intelligent terminals often require padding on intelligent operations at high (and sometimes even low) speed. This is specified by a number before the string in the capability and has meaning for the capabilities that have a P at the front of their comment. This normally is a number of milliseconds to pad the operation. In the current system, which has no true programmable delays, we do this by sending a sequence of pad characters (normally nulls, but can be specified by PC).

In some cases, the pad is better computed as some number of milliseconds times the number of affected lines (to the bottom of the screen usually, except when terminals have insert modes that shift several lines). This is specified as, for example, 12*. before the capability, to say 12 milliseconds per affected whatever (currently always line). Capabilities where this makes sense say P*.

Terminal capabilities, types, and identifiers are described in detail in *termcap*(F) in the IBM Personal Computer *XENIX Command Reference*.

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" later 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 IBM Personal Computer *XENIX Command Reference*.

> **Note:** Use terminal mode functions only with other screen processing functions. Never use them alone.

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.

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.

Terminating Screen Processing

In order to do certain optimizations, and, on some terminals, to work at all, some things must be done before the screen routines start up. These functions are performed in gettmode() and setterm(), which are called by initscr(). In order to clean up after the routines, the routine endwin() is provided. It restores tty modes to what they were when initscr() was first called. Thus, anytime after the call to initscr, endwin() should be called before exiting.

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();
}
```

The *endwin* function 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.

How to Use the Screen Package

This is a description of how to actually use the screen package. In it, we assume all updating, reading, and so forth. It applies to stdscr. All instructions will work on any window, with changing the function name and parameters, as mentioned earlier in the "Naming Conventions" section.

Once the screen windows have been allocated by initscr(), you can set them up for the run. If you want to, say, allow the window to scroll, use scrollok(). If you want the cursor to be left after the last change, use leaveok(). If this isn't done, refresh() will move the cursor to the window's current (y,x) co-ordinates after updating it. New windows of your own can be created, too, by using the functions newwin() and subwin(). The delwin() function allows you to get rid of old windows. If you wish to change the official size of the terminal by hand, just set the variables LINES and COLS to be what you want, and then call initscr(). This is best done before, but can be done either before or after, the first call to initscr(), as it will always delete any existing stdscr and/or curscr before creating new ones.

Output

Now that we have set things up, we will want to actually update the terminal. The basic functions used to change what will go on a window are addch() and move(). The addch() function adds a character at the current (y,x) coordinates, returning ERR if it would cause the window to illegally scroll, that is, printing a character in the lower right-hand corner of a terminal which automatically scrolls if scrolling is not allowed. The move() function changes the current (y,x) coordinates to whatever you want them to be. It returns ERR if you try to move off the window when scrolling is not allowed. As mentioned above, you can combine the two into mvaddch() to do both things in one fell swoop.

The other output functions, such as addstr() and printw(), all call addch() to add characters to the window.

After you have put on the window what you want there, when you want the portion of the terminal covered by the window to be made to look like it, you must call refresh().

In order to optimize finding changes, refresh() assumes that any part of the window not changed since the last refresh() of that window has not been changed on the terminal, that is, that you have not refreshed a portion of the terminal with an overlapping window. If this is not the case, the routine touchwin() is provided to make it look like the entire window has been changed, thus making refresh() check the whole subsection of the terminal for changes.

If you call wrefresh() with curscr, it will make the screen look like curscr thinks it looks like. This is useful for implementing a command which would redraw the screen in case it gets messed up.

Input

Input is essentially a mirror image of output. The complementary function to addch() is getch() which, if echo is set, will call addch() to echo the character. Since the screen package needs to know what is on the terminal at all times, if characters are to be echoed, the tty must be in raw or cbreak mode. If it is not, getch() sets it to be cbreak, and then reads in the character.

Using the Standard Screen

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

The Functions

In the following definitions, "[*]" means that the "function" is really a "#define" macro with arguments. This means that it will not show up in stack traces in the debugger, or, in the case of such functions as addch(), it will show up as it's "w" counterpart.

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.

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.

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 IBM Personal Computer *XENIX Software Command Reference*.) 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 copies 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.

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 reads 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.

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 reads names and other text from the keyboard. For example, in the following program fragment *getstr* 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.

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*1, *argptr*2 . . .

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 IBM Personal Computer *XENIX Software Command Reference*.)

The function reads 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.

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.

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 inserts 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 at 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.

Inserting a Line

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

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.

Deleting a Character

The *delch* function deletes the character at the current position and shifts all the characters to the right of the deleted character 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 deletes 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-- ;
        }
```

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 deletes 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();
```

Clearing the Screen

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

The function call:

```
clear()
```

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 function call:

```
erase
```

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();
```

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 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.

Neither the *clrtobot* or *clrtoeol* functions change the current position.

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.

Creating and Using Windows

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

Creating a Window

The *newwin* function creates a window and returns a pointer that can 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 it is displayed on the terminal screen. The *win* variable must have type WINDOW.

The function is used in programs that maintain a set of windows, displaying different windows at different times or alternating between windows 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 screen.

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

Creating a Subwindow

The *subwin* function creates a subwindow and returns a pointer to the new window. A subwindow is a window that 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 it is dislayed on the terminal screen. The *swin* variable must have type WINDOW*.

The function divides a large window into separate regions. For example, in the following program fragment *subwin* creates the subwindow named cmdmenu in the lower part of the standard screen:

```
WINDOW *cmdmenu;

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

In this example, 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.

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 IBM Personal Computer *XENIX Software Command Reference*.) 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 copies 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);
```

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.

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 reads 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 reads 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 IBM Personal Computer *XENIX Software Command Reference*.)

The function reads 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.

Moving 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 newline 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.

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 edits 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 inserts 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 newline:

```
char line[80];

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

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

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 edits 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 deletes 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--;
        }
```

Clearing the Screen

The *wclear*, *werase*, *wclrtobot*, and *wclrtoeol* functions clear all or some of the characters from the given window by replacing them with spaces. The functions 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 *wrefresh* function call to clear all characters from the window. 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.

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

Refreshing from a Window

The *wrefresh* function updates the terminal screen by copying 1 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 the preceding 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, because 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 occurs, the function attempts to update as much of the screen as possible without causing the scroll.

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 builds a composite screen from overlapping windows. For example, in the following program fragment *overlay* builds the standard screen from two different windows:

```
WINDOW *info, *cmdmenu;

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

Overwriting a Screen

The *overwrite* function copies all characters, including spaces, 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 displays the contents of a temporary window in the middle of a larger window. For example, in the following program fragment, *overwrite* copies the contents of a work window to the standard screen:

```
WINDOW *work;

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

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 moves 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 an error, such as an attempt to move part of a window off the edge of the screen.

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 compare the actual contents of a window with what is assumed to be there. For example, in the following program fragment *inch* and *winch* 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();
```

Reading a character from a window does not alter the contents of the window.

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

```
touchwin(leftscreen);
```

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

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 removes temporary windows from a program to free memory space for other uses. For example, the function call:

```
delwin(midscreen);
```

removes the window named midscreen.

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.

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 distinguishes 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 leaves the corners of the box blank to prevent illegal scrolling.

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 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");
        }
```

The actual appearance 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 IBM Personal Computer *XENIX Command Reference*).

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 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();
        }
```

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 saves 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);
```

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, 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 cursor, each call to refresh automatically clears the screen, regardless of which window is given in the call.

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.

The WINDOW Structure

The WINDOW structure is defined as follows:

```
# define        WINDOW    struct_win_st

struct _win_st {
        short    _cury,_curx;
        short    _maxy,_maxx;
        short    _begy,_begx;
        short    _flags;
        bool     _clear;
        bool     _leave;
        bool     _scroll;
        char     **_y;
        short    *_firstch;
        short    *_lastch;
};

# define        _SUBWIN          01
# define        _ENDLINE         02
# define        _FULLWIN         04
# define        _SCROLLWIN       010
# define        _STANDOUT        0200
```

The _cury and _curx parameters are the current (y, x) coordinates for the window. New characters added to the screen are added at this point. The _maxy and _maxx parameters are the maximum values allowed for (_cury, _curx). The _begy and _begx parameters are the starting (y, x) coordinates on the terminal for the window, in other words, the window's home. The _cury, _curx, _maxy, and _maxx parameters are measured relative to (_begy, _begx), not the terminal's home.

The _clear parameter tells if a clear-screen sequence is to be generated on the next refresh() call. This is only meaningful for screens. The initial clear-screen for the first refresh() call is generated by initially setting clear to be TRUE for curscr, which always generates a clear-screen if set, irrelevant of the dimensions of the window involved. The _leave parameter is TRUE if the current (y, x) coordinates and the cursor are to be left after the last character changed on the terminal, or not moved if there is no change. The _scroll parameter is TRUE if scrolling is allowed.

The __y parameter is a pointer to an array of lines which describe the terminal. Thus:

```
_y[i]
```

is a pointer to the ith line, and

```
_y[i][j]
```

is the jth character on the ith line.

The __flags parameter can have one or more values or'd into it. __SUBWIN means that the window is a subwindow, which indicates to delwin() that the space for the lines is not to be freed. __ENDLINE says that the end of the line for this window is also the end of a screen. __FULLWIN says that this window is a screen. __SCROLLWIN indicates that the last character of this screen is at the lower right-hand corner of the terminal; in other words, if a character was put there, the terminal would scroll. __STANDOUT says that all characters added to the screen are in standout mode.

Examples

Here we present a few examples of how to use the package. They attempt to be representative, though not comprehensive.

Screen Updating

The following examples are intended to demonstrate the basic structure of a program using the screen updating sections of the package. Several of the programs require calculation sections that are irrelevant to the example and therefore are usually not included. However, data structure definitions are provided to help you understand what the relevant portions do. The rest is left as an exercise for you.

Twinkle

This is a moderately simple program that prints patterns on the screen. It switches between patterns of asterisks, putting them on, one-by-one in random order, and then taking them off in the same fashion. It is more efficient to write this using only the motion optimization, which is demonstrated below.

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

# define              NCOLS          80
# define              NLINES         24
# define              MAXPATTERNS     4

struct locs {
    char y, x;
};

typedef struct locs    LOCS;

LOCS  Layout[NCOLS * NLINES];  /* current board
                                       * layout */

int   Pattern,                 /* current pattern
                                * number */

      Numstars;                /* number of stars
                                * in pattern */


main() {

      char     *getenv();
      int      die();

      srand(getpid());         /* initialize random
                                * sequence */
```

(Example continues on next page.)

```
      initscr();
      signal(SIGINT, die);
      noecho();
      nonl();
      leaveok(stdscr, TRUE);
      scrollok(stdscr, FALSE);

      for (;;) {
           makeboard();        /* make the board
                                * setup */

           puton('*');         /* put on '*'s */

           puton(' ');         /* cover up
                                * with ' 's */

           }
}


/*
 * On program exit, move the cursor to the lower
 * left corner by direct addressing, since current
 * location is not guaranteed.
 */
die() {

      signal(SIGINT, SIG_IGN);
      mvcur(0, COLS-1, LINES-1, 0);
      endwin();
      exit(0);
}

/*
 * Make the current board setup.  It picks a random
 * pattern and calls ison() to determine if the
 * character is on that pattern or not.
 */
makeboard() {

      reg int y, x;
      reg LOCS *lp;
```

(Example continues on next page.)

```
	Pattern = rand() % MAXPATTERNS;
	lp = Layout;
	for (y = 0; y < NLINES; y++)
		for (x = 0; x < NCOLS; x++)
			if (ison(y, x)) {
				lp->y = y;
				lp++->x = x;
			}
	Numstars = lp - Layout;
}

/*
 * Return TRUE if (y, x) is on the current pattern.
 */
ison(y, x)
reg int y, x; {

	switch (Pattern) {
	  case 0:			/* alternating lines */
	     return !(y & 01);
	  case 1:			/* box */
	     if (x >= LINES && y >= NCOLS)
			return FALSE;
	     if (y < 3 || y >= NLINES - 3)
			return TRUE;
	     return (x < 3 || x >= NCOLS - 3);
	  case 2:			/* holy pattern! */
	     return ((x + y) & 01);
	  case 3:			/* bar across center */
	     return (y >= 9 && y <= 15);
	}

	/* NOTREACHED */
}

puton(ch)
reg char  ch; {

	reg LOCS	*lp;
	reg int		r;
	reg LOCS	*end;
	LOCS		temp;
```

(Example continues on next page.)

```
        end = &Layout[Numstars];
        for (lp = Layout; lp < end; lp++) {
             r = rand() % Numstars;
             temp = *lp;
             *lp = Layout[r];
             Layout[r] = temp;
        }

        for (lp = Layout; lp < end; lp++) {
             mvaddch(lp->y, lp->x, ch);
             refresh();
        }
}
```

Life

This program plays the famous computer pattern game of life[1] . The calculational routines create a linked list of structures defining where each piece is. Nothing here claims to be optimal, merely demonstrative. This program, however, is a very good place to use the screen updating routines, as it allows them to worry about what the last position looked like, so you don't have to. It also demonstrates some of the input routines.

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

/*
 * Run a life game.  This is a demonstration program for the
 * Screen Updating section of the -lcurses cursor package.
 */

 struct lst_st {          /* linked list element */
      int y,x;            /* (y, x) position of piece */

      struct lst_st *next, *last;  /* doubly linked */
};
```

(Example continues on the next page.)

[1] Scientific American, May, 1974

```
typedef struct lst_st          LIST;

LIST        *Head;       /* head of linked list */

main(ac, av)
int         ac;
char        *av[]; {

            int       die();

            evalargs(ac, av);
                          /* evaluate arguments */

            initscr();    /* initialize screen
                           * package */

            signal(SIGINT, die);
                          /* set to restore tty
                           * stats */


            crmode();     /* set for char-by-char */

            noecho();     /*input */

            nonl();       /* for optimization */


            getstart();   /* get starting position */
            (for ; ;){

                 prboard();
                          /* print out current board */
                 update();
                          /* update board position */
             }
}

/*
 * This is the routine that is called when rubout
 * is hit.  It resets the tty stats to their
 * original values.  This is the normal way of
 * leaving the program.
 */
 die() {
```

(Example continues on next page.)

```
           signal(SIGINT, SIG_IGN);
                        /* ignore rubouts */
           mvcur(0, COLS-1, LINES-1, 0);
                        /* go to bottom of
                         * screen */
           endwin();    /* set terminal to initial
                         * state */
           exit(0);
}

/*
 * Get the starting position from the user.  The
 * keys u, i, o, j, l, m, ,, and . are used for
 * moving their relative directions from the k key.
 * Thus, u moves diagonally up to the left, , moves
 * directly down, and so forth.  x places a piece
 * at the current position, " " takes it away.  The
 * input can also be from a file.  The list is
 * built after the board setup is ready.
 */
getstart() {

           reg char            c;
           reg int             x, y;

           box(stdscr, '|', '_');
                        /* box in the screen */
           move(1, 1);  /* move to upper left
                         * corner */

           do {
                refresh();
                        /* print current
                         * position */
                if ((c=getch()) == 'q')
                     break;
                switch (c) {
                  case 'u':
                  case 'i':
                  case 'o':
                  case 'j':
                  case 'l':
                  case 'm':
                  case ',':
```

(Example continues on next page.)

```
                        case '.':
                            adjustyx(c);
                            break;
                        case 'f':
                            mvaddstr(0, 0, "File name: ");
                            getstr(buf);
                            readfile(buf);
                            break;
                        case 'x':
                            addch('X');
                            break;
                        case ' ':
                            addch(' ');
                            break;

                    }
                }
                if (Head != NULL)
                                /* start new list */
                    dellist(Head);
                Head = malloc(sizeof (LIST));

                /*
                 * Loop through the screen looking for
                 * 'x's, and add a list element for
                 * each one.
                 */
                for (y = 1; y < LINES - 1; y++)
                    for (x = 1; x < COLS - 1; x++) {
                        move(y, x);
                        if (inch() == 'x')
                            addlist(y, x);
                    }
}

/*
 * Print out the current board position from the
 * linked list.
 */
prboard() {
```

(Example continues on next page.)

```
          reg LIST            *hp;

          erase();      /* clear out last
                         * position */
          box(stdscr, '|', '_');
                        /* box in the screen */

          /*
           * go through the list adding each
           * piece to the newly blank board
           */
          for (hp = Head; hp; hp = hp->next)
              mvaddch(hp->y, hp->x, 'X');

          refresh();
}
```

Motion Optimization

The following example shows how motion optimization is written on its own. Programs that flit from one place to another without regard for what is already there usually do not need the overhead of both space and time associated with screen updating. They should instead use motion optimization.

Twinkle

The twinkle program is a good candidate for simple motion optimization. Here is how it could be written (only the routines that have been changed are shown):

```
main() {

  reg char             *sp;
  char                 *getenv();
  int                  _putchar(), die();

  srand(getpid());  /* initialize random sequence */
```

(Example continues on next page.)

```
        if (isatty(0)) {
          gettmode();
          if (sp=getenv("TERM"))
              setterm(sp);
            signal(SIGINT, die);
          }
          else {
            printf("Need a terminal on %d\n", _tty_ch);
            exit(1);
          }
          _puts(TI);
          _puts(VS);

          noecho();
          nonl();
          tputs(CL, NLINES, _putchar);
          for (;;) {
            makeboard(); /* make the board setup */
            puton('*');  /* put on '*'s */
            puton(' ');  /* cover up with ' 's */
          }
    }

    /*
     * _putchar defined for tputs() (and_puts())
     */_putchar_putchar(c)
    reg char             c; {

        putchar(c);
    }

    puton(ch)
    char  ch; {

        static int          lasty, lastx;
        reg LOCS            *lp;
        reg int             r;
        reg LOCS            *end;
        LOCS                temp;
```

(Example continues on next page.)

```
	end = &Layout[Numstars];
	for (lp = Layout; lp < end; lp++) {
	  r = rand() % Numstars;
	  temp = *lp;
	  *lp = Layout[r];
	  Layout[r] = temp;
	}

	for (lp = Layout; lp < end; lp++)
	                        /* prevent scrolling */
	  if (!AM || (lp->y < NLINES - 1 ||
	  lp->x < NCOLS - 1)) {
	    mvcur(lasty, lastx, lp->y, lp->x);
	    putchar(ch);
	    lasty = lp->y;
	    if ((lastx = lp->x + 1) >= NCOLS)
	      if (AM) {
	        lastx = 0;
	        lasty++;
	      }
	      else
	        lastx = NCOLS - 1;
	    }
}
```

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 that can be used with the movement prefix is given in *curses*(S) in the IBM Personal Computer *XENIX Software Command Reference.*

Controlling the Terminal

The following sections explain:

- How to set the terminal modes
- How to move the cursor
- How to access other aspects of the terminal.

Use these functions only when using other screen processing functions.

Terminal Modes

COOKED (NORAW): Normal Mode (equivalent to stty sane). In this mode, lines of input are collected and input editing is done. The edited line is made available when it is completed by a newline or when EDT (Ctrl-d) is entered. Carriage return is usually synonymous with newline in this mode, and replace with a newline whenever it is typed. All tty driver functions, (input editing, interrupt generation, output processing such as delay generation and tab expansion, and so forth) are available in this mode.

CBREAK: This mode eliminates character and line editing input facilities, making the input character available to the user as it is typed. Flow control, and interrupt processing are still done in this mode. Output processing is still done.

RAW: This mode eliminates all input processing and makes all input characters available as they are typed. No output processing is done.

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 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.

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.

Moving the Terminal's Cursor

After getting gettmode() and setterm() to get the terminal descriptions,the *mvcur* function moves the terminal's cursor from one position to another in an optimal fashion. Its usage is simple; you tell it where you are now and where you want to go. 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. mvcur (0,0, LINES/2, COLS 2); would move the cursor form the home position (0,0) to the middle of the screen.

If you wish to force absolute addressing, you can just tell mvcur() that you are impossibly far away. For example, to absolutely address the lower-left hand corner of the screen from any where just claim that you are in the upper-right hand corner:

```
mvcur (0, COLS-1, LINES-1, 0);
```

> **Note:** The *mvcur* function should be used only 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.

Getting the Terminal Mode

The *gettmode* function returns the current terminal 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.

Variables Set by gettmode ()

If US and UE do not exist in the termcap entry, they are copied from SO and SE in setterm ().

Type	Name	Description
bool	NONL	Term cannot handle linefeeds doing a CR
bool	GT	Gtty indicates tabs
bool	UPPERCASE	Terminal generates only uppercase letters

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.

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 *setterm* may be used in special cases.

Variables Set by setterm()

Type	Name	Pad	Description
char *	AL	P*	Add new blank line
bool	AM		Automatic margins
char *	BC		Back cursor movement
bool	BS		Back space works
char *	BT	P	Back tab
bool	CA		Cursor addressable
char *	CD	P*	Clear to end of display
char *	CE	P	Clear to end of line
char *	CL	P*	Clear screen
char *	CM	P	Cursor motion
char *	DC	P*	Delete character

Type	Name	Pad	Description
char *	DL	P*	Delete line sequence
char *	DM		Delete mode (enter)
char *	DO		Down line sequence
char *	ED		End delete mode
bool	EO		Can erase overstrikes with ' '
char *	EI		End insert mode
char *	HO		Home cursor
bool	HZ		Hazeltine ~ braindamage
char *	IC	P	Insert character
bool	IN		Insert-null blessing
char *	IM		Enter insert mode (IC usually set, too)
char *	IP	P*	Pad after character inserted using IM+IE
char *	LL		q*ick to last line, column 0
char *	MA		Ctrl character map for cmd mode
bool	MI		Can move in insert mode
bool	NC		No cr: /r sends /r/n then eats /n
char *	ND		Non-destructive space
bool	OS		Over strike works

Type	Name	Pad	Description
char	PC		Pad character
char *	SE		Standout end (may leave space)
char *	SF	P	Scrolls forward
char *	SO		Stand out begin (may leave space)
char *	SR	P	Scrolls in reverse
char *	TA	P	Tab (not ∧ I or with padding)
char *	TE		Terminal address enable ending sequence
char *	TI		Terminal address enable initialization
char *	UC		Underline a single character
char *	UE		Underline ending sequence
bool	UL		Underlining works even though !OS
char *	UP		Upline
char *	US		Underline starting sequence
char *	VB		Visible bell
char *	VE		Visual end sequence
char *	VS		Visual start sequence
bool	XN		A newline is lost after wrap

Names starting with X are reserved for serious disturbances.

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 gets the full name of the terminal currently being used. 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

Contents

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.

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.

Testing for an ASCII Character

The *isascii* function tests for characters in the ASCII character set, that is, 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 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.

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 prepares 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.

Testing for Alphanumerics

The *isalnum* function tests for letters and decimal digits, that is, 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.

Testing for a Letter

The *isalpha* function tests for uppercase or lowercase letters, that is, 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.

Testing for Control Characters

The *iscntrl* function tests for control characters, that is, 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.

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;
```

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 hexidecimal 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.

Testing for Printable Characters

The *isprint* function tests for printable characters, that is, 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.

Testing for Punctuation

The *ispunct* function tests for punctuation characters, that is, 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.

Testing for Whitespace

The *isspace* function tests for whitespace characters, that is, the space, horizontal tab, vertical tab, 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.

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.

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 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.

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 one reads or writes 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.

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 (\Ø) to the last character in the new string. The function always returns the pointer *dst*.

The function builds 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");
```

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 (that is, 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 because the strings are identical in every way, but the function call:

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

returns -1 because the ASCII value of B is one greater than A.

The *strcmp* function continues to compare characters until a mismatch is found. If one string is shorter than the other, the function 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.

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 (\Ø) 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];
main () {
   strcpy(name, na);
}
```

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

Getting a String's Length

The *strlen* function returns the number of characters 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 the contents of inname are short enough to be stored in name:

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

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

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 3 characters in letter to the end of cover:

```
char cover[] = "cover";
char letter[] = "letter";
   main ()
   strncat(cover, letter, 3);
   }
```

This example creates the new string coverlet in cover.

Comparing Characters in Strings

The *strncmp* function compares one or more pairs of characters in two given strings and returns an integer value that 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 5 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.

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

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 returns the address of *dst*.

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

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

    strncpy(day, date, 3);
    print f("s/n",buf);}
```

In this example, day receives the string Fri.

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 IBM Personal Computer *XENIX Software Command Reference*). The function always returns the number of values read.

The function reads values from a string containing several values of different formats, or reads 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];
main () {
   sscanf(datestr,"%*3s%3s%*2s%*8s%*3s%4s",month,year);
   printf("%s,%s\n",month,year);
}
```

The first value (a 3-character string) is stored at the location pointed to by month, the second value (a 4-character string) is stored at the location pointed to by "year".

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 that 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 IBM Personal Computer *XENIX Software Command Reference*). After all values are written to the string, the function adds a null character (\Ø) to the end of the string. The function normally returns zero, but will return a nonzero value if an error is encountered.

The function builds a large string from several values of different formats. 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

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

Contents

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 programs that 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 IBM Personal Computer *XENIX Software Command Reference.*

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

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.

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 operates 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);
```

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 you use the *system* function, 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.

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 terminates 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);
```

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.

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 that 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 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 calls 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 passes 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:

```
#include <stdio.h
main (argc argv)
int argc;
char * argv [ ];
{
   int i;
   char *cmd[ ];

   cmd[0] = "cat";
   for (i=1; 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 (for example, 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.

The *execl* and *execv* functions do not expand metacharacters (for example, <, >, *, ?, 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.

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 that end with *.c*, something that the cat command cannot do by itself.

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 makes several copies of any program that must take divergent actions as a part of its normal operation, for example, 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 determines which steps the child and parent should take next. For example, in the program fragment:

```
char *cmd;

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

The child's return value, 0, causes the expression fork() ==, 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.

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 synchronizes 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 are 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 IBM Personal Computer *XENIX Software Command Reference*for a description of the various kinds of termination). The next 8 bits are 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.

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 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*.

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.\n");
        exit(1);
}
```

(Example continues on next page.)

```
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(stderr,"Bad tty name:%s/n",
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, for example, /dev/tty01. 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 Ctrl-D key is pressed at the new terminal.

Chapter 6. Creating and Using Pipes

Contents

Introduction

A pipe is a way a program can pass information to other programs without any temporary file. A pipe connects the output of one program to the input of another program. 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 operating system.

Pipes are chiefly used to pass information among 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.

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 that 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 returns the null pointer value NULL if an error is encountered.

The function is used in programs that 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 is used as input by the cat command.

Reading and Writing to a Process

The *fscanf*, *fprintf*, and other stream functions can read from or write to a pipe opened by the *popen* function. These functions have the same form as described in Chapter 2, "Using the Stream Functions."

The *fscanf* function can 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 to write to 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.

Closing a Pipe

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

pclose(*stream*)

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

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

```
FILE *pstrm;

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

Opening a Low-Level Pipe

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

pipe(*fd*)

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

```
int chan[2];

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

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

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

```
int fd[2];

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

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

Reading and Writing to a Low-Level Pipe

The *read* and *write* input and output functions can read and write characters to a low-level pipe. These functions have the same form and operation described in Chapter 2.

The *read* function can read from the read side of an open pipe. For example, in the following program fragment, *read* reads MAX characters from the read side of the pipe given by chan:

```
int chan[2];
char buf[MAX];
int number;

pipe(chan);
number = read(chan[0], buf, MAX);
```

In this example, *read* stores the characters in the array buf.

Unless the end-of-file character is encountered, a *read* call waits for the given number of characters to be read before returning.

The *write* function can write to the write side of a pipe. For example, in the following program fragment, *write* writes MAX characters from the character array buf to the writing side of the pipe given by chan:

```
int chan[2];
char buf[MAX];
int number;

pipe(chan);
number = write(chan[1], buf, MAX);
```

If the *write* function finds that a pipe is too full, it waits until some characters have been read before completing its operation.

Closing a Low-Level Pipe

The *close* function can be used to close the reading or the writing side of a pipe. The function has the same form and operation as described in Chapter 2. For example, the function call:

```
close(chan[0]);
```

closes the reading side of the pipe given by chan, and the call:

```
close(chan[1]);
```

closes the writing side.

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.

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 links 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. Because 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. Because 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);
```

(Example continues on next page.)

```
        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 hangup signals. The last set restores the signals to their original status. The *signal* function is described in detail in Chapter 7, "Using Signals".

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 a file descriptor.

Chapter 7. Using Signals

Contents

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 the system detecting an illegal operation or a user pressing the Interrupt (Del) key. The Interrupt key is the Del key on your keyboard. 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 disable a signal to prevent it from affecting the program. It can also 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.

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 (Del) key, SIGQUIT (for quit signal) caused by pressing the QUIT key, or SIGHANG (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 IBM Personal Computer *XENIX Software Command Reference*.)

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 usually terminates the program.

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

Disabling a Signal

You can disable a signal, that is, 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 prevents a signal from terminating a program executing in the background (for example, 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 Del (Interrupt) key to stop a program running in the foreground stops a program running in the background if it has not disabled that signal. For example, in the following program fragment *signal* disables 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. If the parent process is interrupted, the child process continues to execute until it reaches its normal end.

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*, SIG__DFL)

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 restores 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 read from the file given by fp.

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 lets 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 (Del) key to stop.\n");
        signal(SIGINT, catch);
        while () {
                /* Body */
        }
}

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

The *catch* function displays 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.

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 restores 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.

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)();
```

(Example continues on next page.)

```
        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);
}
```

The parent uses the while statement to wait until the child's Process IDENTIFICATION (PID) number 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.

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.

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 to 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 is 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;
```

(Example continues on next page.)

```
        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. The interrupt signal is disabled before processing the delayed signal to prevent a change to sigflag when it is being tested.

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.

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

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

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

Using Signals in Interactive Programs

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

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

The *setjmp* function has the form:

setjmp(*buffer*)

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

```
jmpbuf oldstate;

setjmp(oldstate);
```

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

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 *setjmp* 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 *setjmp* function that saved the previous execution state. For example, in the following program fragment, *setjmp* 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);
```

(Example continues on next page.)

```
        /* 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 *setjmp* function.

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 access to a signal even if that program is executing in the background or as a child to some other program.

Protecting Background Processes

Any program 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. Also, they complete their tasks without returning the prompt. Because 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 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.

Protecting Parent Processes

A program can create and wait for a child process that catches its own signals 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

Contents

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 this use with 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

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.

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 4 bytes of 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 allocates storage for a group of strings that vary in length. For example, in the following program fragment, *malloc* allocates space for 10 strings, each of different length:

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

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

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

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 used in programs that must process large arrays without knowing the size of an array in advance. For example, in the following program fragment, *calloc* allocates storage for an array of values read from the standard input:

```
char * calloc ();
int i;
char table[10 ]
unsigned inum;

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

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

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 returns a null pointer value if there is not enough space to allocate.

This function keeps storage as compact as possible. For example, in the following program fragment *realloc* removes table entries:

```
main ()
{
   char * realloc ();
   char *strings=[10]
   int i;
   unsigned inum;

   for (i=inum; i>-1;i--) {
        string[i]= 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.

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 frees space that is no longer used or frees space to be used for other purposes. For example, in the following program fragment, *free* frees the allocated space pointed to by table if the first element is equal to zero:

```
main ()
{
char table[10];

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

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, *locking*. This function locks any given section of a file, preventing all other processes that wish to use the section from gaining access. A process may lock the entire file or only a small portion of it. In any case, only the locked section is protected; all other sections can be accessed by other processes as usual.

File locking protects a file from the damage caused by several processes that try to read or write to the file at the same time. File locking 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.

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 can be used any number of times to move the character pointer to each new section to be locked. For example, the following statements locate the file pointer at the byte position 1024 from the beginning of the file *reservations* for locking:

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

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 locked, *mode* is an integer value which defines the type of lock applied to the file, *size* is a long integer value giving the size in bytes of the portion of the file section locked or unlocked. The *mode* may be LOCK for locking the given bytes, 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",O_RDWR);
      locking(fd,LOCK,100L);
}
```

The function normally returns the number of bytes locked but returns - 1 if it encounters an error.

Program Example

This section shows how to lock and unlock a small section in a file using the *locking* function. Note that a *seek* to a specified position is done immediately prior to the *locking* un*locking* of the file segment. It is always advisable to do this because the *locking* call simply takes the current file pointer position as the starting point of the segment to be locked. In the following program, a file ("data" is *opened* for reading and writing) is opened, a *seek* done to set the file pointer correctly, and then a 100 bytes of the file are locked. A *seek* is again done to the specified location and then the *locking* call is used again to unlock the the file.

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

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


fd = open("data",O_RDWR);          /* 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);   /* Unlock bytes 100-200 */

       }
}
```

Using Semaphores

The standard C library provides a group of functions, called the semaphore functions, which can 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 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.

Creating a Semaphore

The *creatsem* function creates a semaphore, returning a semaphore number that 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 that 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 used at the beginning of one process to define the semaphores it shares with other processes. For example, in the following program fragment, *creatsem* creates a semaphore named tty1 before preceding with its tasks:

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

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

The *fopen* function 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. Before resetting a semaphore, you must remove the associated semaphore file using the *unlink* function

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 can 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 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* opens 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. Although the *opensem* function does not return an error value, opening a semaphore more than once can lead to a system deadlock.

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 (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 the current controlling process relinquishes control of the semaphore before returning to the next statement.

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 used in place of *waitsem* to take control of a semaphore.

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 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.

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.

Program Example

This section shows how to use the semaphore functions to control the access of a system resource. The following program creates five processes that 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, the use of semaphores is clearly illustrated:

```
#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)
{
```

(Example continues on next page.)

```
        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.

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 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 manifest constants and other macros used by the functions.

Creating a Shared Data Segment

The *sdget* function creates a shared data segment for the current process and attaches the segment to the process's data space. 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, *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. The *flag* may be a combination of SD_CREAT for creating the segment, and SD_RDONLY for attaching the segment for reading only or SD_WRITE for attaching the segment for reading and writing. You may also use SD_UNLOCK for allowing simultaneous access by multiple processes. The values can be combined by logically ORing them. The function returns the address of the segment if it has been successfully created. The function returns -1.

The function is typically used by just one process to create a segment that it will share with several other processes. For example, in the following fragment, the program uses *sdget* to create a segment and attach it for reading and writing. The address of the new segment is assigned to *shared*

```
#include <sd.h>

main ()
{
   char *shared;

   shared=sdget("/tmp/share", SD_CREAT|SD_WRITE,
   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 severly degrade the execution performance of the program.

Attaching a Shared Data Segment

The *sdget* function can also be used to attach an existing shared data segment to a process's data space. In this case, the function call has the form:

sdget (*path,flag*)

where *path* is a character pointer to the pathname of a shared data segment created by some other process, and *flag* is an integer value which defines how the segment should be attached. The *flag* may be SD__RDONLY for attaching the segment for reading only, or SD__WRITE for attaching the segment for reading and writing. If the function is successful, it returns the address of the new segment. Otherwise, it returns -1.

The function can be used to attach any shared data segment a process may wish to access. For example, in the following fragment, the program uses *sdget* to attach the segments associated with the files */tmp/share1*and*/tmp/share2* for reading and writing. The addresses of the new segments are assigned to the pointer variables *share1* and *share2*.

```
#include<sd.h>
main()
{
char *share1,*share2;
share1=sdget("/tmp/share1",SD_
WRITE;
share2=sdget("/tmp/share2",SD_
WRITE;
}
```

The *sdget* function returns an error value to any process that attempts to access a shared data segment without the necessary permissions. The segment permissions are defined when the segments is created

Entering a Shared Data Segment

The *sdenter* function signals a process's intention to access the contents of a shared data segment. A process cannot 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 that 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. Use The *sdleave* function 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.

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 ()
{
inti=0
char c, *share;

share=sdget("/tmp/share", SD_RDONLY);

sdenter(share,SD_RDONLY);
        c= *share
sdleave(share);
```

(Example continues on next page.)

```
while (c!=0) {
        putchar(c);
        i++;
        sdenter(share,SD_RDONLY);
                c=share[i];
        sdleave(share);
}
}
```

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 chooses an action based on the current version number of the segment. For example, in the following program fragment, *sdgetv* determines whether *sdenter* should 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.

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 that defines the version number to wait for. 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 "vnum" performs its operations in the segment:

```
#include <sd.h>

main ()
{
   char *share;
   int change;

   vnum=sdgetv(share);
   i=0;
   if (sdwaitv(share,vnum)= = -1)
             fprintf(stderr, "Cannot find segment/n");

   else
             sdenter(share);
```

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

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.

Program Example

This section shows how to use the shared data functions to share a single data segment between two processes. The following program attaches a data segment named */tmp/share* and then uses it to transfer information to between the child and parent processes.

```
#include <sd.h>

main()

 char *share,message[12];
 int i, vnum;

 share = sdget("/tmp/share",SD_CREAT|SD_WRITE,12,0777);
```

(Example continues on next page.)

```
 if (fork() = = 0) {
         for (i=0; i<4;i++){
           sdenter (share, SD_WRITE);
                   strncpy(message, share, 12);
                   strncpy(share, "Shared data", 12);
                   vnum = sdgetv (share);
           sdleave (share);
           sdwaitv (share, vnum + 1);
           printf ("Child:%d - %s/n", i, message);
         }
         sdenter (shared, SD_WRITE);
          strncpy (message, share, 12);
          strncpy (share, "Shared data", 12);
         sdleave (share);
         printf ("Child: %d - %s\n", i, message);
         exit (0);

}
for (i=0; i<5; i++) {
       sdenter (share, SD_WRITE);
        strncpy (message, share, 12);
        strncpy (share,"Data shared", 12);
        vnum = sdgetv (share);
       sdleave (share);
       sdwaitv (share, vnum + 1);
       printf (Parent: %d - %s\n", i, message);
}

sdfree (share);
}
```

In this program, the child process inherits the data segment created by the parent process. Each process accesses the segment 5 times. During the access, a process copies the current contents of the segment to the variable *message* and replaces the message with one of its own. It then displays *message* and continues the loop.

To synchronize access to the segment, both the parent and child use the *sdgetv* and *sdwaitv* functions. While a process still has control of the segment, it uses *sdgetv* to assign the current version number to the variable *vnum*. It then uses this number in a call to *sdwaitv* to force itself to wait until the other process has accesed the segment. Note that the argument to *sdwaitv* is actually "vnum +1". Since *vnum* was assigned before the *sdleave* call, it is exactly one less than the version number after the *sdleave* call. It is assigned before the *sdleave* call to ensure that the other process does modify the current version number before the current process has a chance to assign it to *vnum*

The last time the child process accesses the segment, it displays the message and exits without calling the *sdwaitv* function. This is to prevent the process from waiting forever, since the parent has already exited and can no longer modify the current version number.

Chapter 9. Error Processing

Contents

Introduction

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

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 occurs, such as an attempt to access unavailable or protected memory that results in a violation of memory space.

Using the Standard Error File

The standard error file is a special output file to display error messages. The standard error file is one of three standard files (standard input, standard output, and standard 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 file and standard output file, has predefined file pointer and file descriptor values. The file pointer stderr is used in stream functions to copy data to the error file. The file descriptor **2** is 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).

Using the errno Variable

The errno variable is a predefined external variable that 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 can be used in subsequent statements to process the error.

The errno variable is used immediately after a system function has returned an error. In the following program fragment, errno determines 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 defined in errno.h), 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 ends.

To use the errno variable in a program, you must explicitly define it as an external variable with int type. The file *errno.h* contains manifest constant definitions for each error number. These constants can 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 Introduction(S) in the IBM Personal Computer *XENIX Software Command Reference.*

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.

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 IBM Personal Computer *XENIX Software Command Reference.*

A program can 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.

Encountering System Errors

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 IBM Personal Computer *XENIX Command Reference*.

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.

Most system errors occur during calls to system functions. If the system error is recoverable, the system returns an error value to the program and sets the error 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 one 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 are 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, the error is not reported.

Appendixes

Contents

Appendix A. Assembly Language Interface

Introduction

This appendix explains how to use 8086/80286 assembly language routines with C language programs and functions. In particular, it explains how to call assembly language routines from C language programs and how to call C language functions from an assembly language routine.

This assembly language interface is especially useful for assembly language programmers who wish to see the functions of the standard C library and other C libraries.

C Calling Sequence

To receive values from C language function calls or to pass values to C functions, assembly language routines must follow the C argument passing conventions. C language function calls pass their arguments to the given functions by pushing the value of each argument onto the stack. The call pushes the value of the last argument first and the first argument last. If an argument is an expression, the call computes the expression's value before pushing it onto the stack. The same number of bytes the variable occupied in the main program is pushed onto the stack.

Arguments with char, int, or unsigned type occupy a single word (16 bits) on the stack. Arguments with long type occupy a doubleword (32 bits) with the value's high-order word occupying the first word. Arguments with float type are converted to double type (64 bits). Note that char type arguments are zero-extended to int type before being pushed on the stack.

If the name of an array is used as an argument, the value passed to the function is the address of the beginning of the array.

If an argument is a structure, the function call pushes the last word of the structure first and each successive word in turn until the first word is pushed. Note that when you want the called routine to modify an argument (call-by-reference), you simply pass the address of the argument by preceding it with an "&".

After a function returns control to a routine, the calling routine is responsible for removing arguments from the stack.

Entering an Assembly Routine

Assembly language routines that receive control from C function calls should preserve the contents of the bp, si, and di registers and set the bp register to the current sp register value before proceeding with their tasks. The following example illustrates the recommended instruction sequence for entry to an assembly language routine:

```
entry:
        push    bp
        mov     bp,sp
        push    di
        push    si
```

This is the same sequence used by the C compiler.

If this sequence is used, the last argument passed by the function call (which is also the first argument given in the call's argument list) is at address 4(bp). Subsequent arguments begin at address 6(bp) or 8(bp), depending on the size of the first argument.

This sequence is strongly recommended even if the si and di registers are not modified, because it allows backtracing with the *adb* program during program debugging.

Return Values

Assembly language routines that wish to return values to a C language program or receive return values from C functions must follow the C return value conventions. C functions place return values that have int, char, or unsigned type in the ax register. They place values with long type in the ax and dx registers, with the high-order word in dx.

To return a structure or a floating point value, C functions place the address of the given value in the ax register. The structure or floating-point value must be in a static area in memory. Long addresses are returned in the ax and dx registers, with the segment selector in dx.

Exiting a Routine

Assembly language routines that return control to C programs should restore the values of the bp, si, and di registers before returning control. The following example illustrates the recommended instruction sequence for exiting a routine:

```
pop     si
pop     di
mov     sp, bp
pop     bp
ret
```

This sequence does not change the ax, bx, cx, or dx registers or any of the segment registers. The sequence does not remove arguments from the stack. This is the responsibility of the calling routine.

Program Example

To illustrate the assembly language interface, consider the following example of a C function:

```
add (i,j)
int  i,j;
{
return(i+j);
}
```

If written as an assembly language routine, this function must:

- Save the proper registers
- Retrieve the arguments from the stack
- Add the arguments
- Place the return value in the ax register
- Restore registers
- Return control

The following is a example of how the routine can be written:

```
_add:
        push    bp
        mov     bp,sp
        push    di
        push    si

        mov     ax,*4(bp)
        add     ax,*6(bp)

        pop     si
        pop     di
        mov     sp, bp
        pop     bp
        ret
```

If this C function:

```
add (i,j)
int  i,j;
{
return (i+j);
}
```

is to be called by an assembly language routine, the routine must contain instructions that push the arguments on the stack in the proper order, call the function, and clear the stack. It may then use the return value in the ax register. The following is an example of the instructions that can do this:

```
push     <j value>
push     <i value>
call     _add
add      sp,*4
```

Note that the order in which you push and pop parameters on stack is critical.

Appendix B. XENIX System Calls

Introduction

This appendix lists some of the differences among XENIX, XENIX 2.3, 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 the XENIX system.

Executable File Format

Both XENIX and UNIX System 3.0 execute only 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 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 IBM Personal Computer *XENIX Command Reference.*

Revised System Calls

Some system calls in XENIX 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 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 3.0 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 3.0. The *clocal* function provides access to all calls unique to an Original Equipment Manufacturer (OEM).

Version 7 Additions

XENIX maintains a number of UNIX V7 features that were dropped from System 3.0. In particular, XENIX 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.

Changes to the ioctl Function

XENIX and UNIX System 3.0 have a full set of XENIX 2.3-compatible *ioctl* calls. Furthermore, XENIX 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.

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 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 also allows the ".." symbol, but restricts its use to the super-user.

Using the mount and chown Functions

Both XENIX 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.

Super-Block Format

Both UNIX System 3.0 and UNIX System 5.0 have new super-block formats. XENIX uses the UNIX System 5.0 format but uses a different number for each revision. The XENIX super-block has an additional field at the end, which can be used to distinguish between XENIX 2.3 and XENIX super-blocks. XENIX checks this number at start time and during a mount. If a XENIX 2.3 super-block is read, XENIX converts it to the new format internally. Similarly, if a XENIX 2.3 super-block is written, XENIX 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.

Separate Version Libraries

XENIX 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.

Appendix C. A Common Library for XENIX and DOS

Introduction

This appendix lists the XENIX library routines that form the Common C Library for the IBM Personal Computer XENIX and IBM Personal Computer Disk Operating System (DOS) versions of the Microsoft® C Compiler. These routines can be used by programmers who wish to develop C programs for both the XENIX and DOS environments. The routines provide an identical interface to a set of operations that are useful on both XENIX and DOS.

The following table contains the common routines:

abort *	execl *	freopen *	islower	puts	strncat
abs	execle *	frexp	isprint	putw	strncmp
access *	execlp *	fscanf	ispunct	rand	strncpy
acos	execv *	fseek *	isspace	read *	strpbrk
asctime	execve *	fstat *	isupper	realloc	strrchr
asin	execvp *	ftell	isxdigit	rewind	strspn
assert	exit *	ftime	j0	sbrk *	strtok
atan	exp	fwrite *	j1	scanf	swab
atan2	fabs	gcvt	jn	setbuf	system *
atof	fclose	getc	ldexp	setjmp	tan
atoi	fcvt	getchar	localtime	signal *	tanh
atol	fdopen *	getcwd	log	sin	time
calloc	feof	getenv	log10	sinh	toascii
ceil	ferror	getpid *	longjmp	sprintf	tolower
chdir *	fflush	gets	lseek *	sqrt	toupper
chmod *	fgetc	getw	malloc	srand	tolower
chsize *	fgets	gmtime	mktemp *	sscanf	toupper
clearerr	fileno	hypot	strcat *	stat *	umask *
close	floor	isalnum	modf	strchr	ungetc

(Table continues on next page.)

cos	fmod	isalpha	open *	strcmp	unlink *
cosh	fopen *	isascii	perror	pow	utime
creat *	fprintf	isatty *	strcspn	strcpy	write *
ctime	fputc	iscntrl	printf	strdup	y0
dup	fputs	isdigit	putc	strlen	y1
dup2	fread *	isgraph	putchar		yn
ecvt	free				

Routines marked by an asterisk (*) have a slightly different operation or meaning in the DOS environment than they do under XENIX. These differences are fully described in the following sections. Routines which are not marked function exactly the same in DOS as they do in XENIX. Complete descriptions are given in section S of the IBM Personal Computer *XENIX Software Command Reference.*

Common Include Files

Structure definitions, return value types, and manifest constants used in the descriptions of some of the common routines may vary from environment to environment and are therefore fully defined in a set of include files for each environment. There are the following include files:

```
errno.h
math.h
stat.h
stdio.h
types.h
```

The *errno.h* file contains definitions of the error values returned in the global variable, **errno**. Whenever a library routine or system call detects an error, a general error indicator is returned from the call. The indicator is defined to be some otherwise illegal return value, usually -1. This method is used to avoid possible conflicts between an error return and a legitimate return value. When an error return is detected, the actual error value can be determined by looking at the value of **errno**. The value of **errno** is undefined if the function returned a non-error result.

The *math.h* file defines some of the floating point math routine interfaces and some standard constants.

The *stat.h* file defines the format, fields, and constant values for the file status structure returned by the *stat* and *fstat* routines.

The *stdio.h* file contains the definitions of the basic system file structure, FILE, some of the basic operations available of files, such as the *putc*, *getc*, *putchar*, and *getchar* routines, as well as the standard pointer constant NULL.

The *types.h* file defines some of the types used in defining system structures, such as the time, date, and file status structures.

Differences between Common Routines

The following sections explain how the DOS routines of the common library differ from their XENIX counterparts. These descriptions are intended to be used in conjunction with the more detailed descriptions of XENIX functions provided in section S of the IBM Personal Computer *XENIX Software Command Reference.*

abort

The *abort* routine terminates the process and returns control to the operating system *without* creating a core file.

access

The *access* routine checks the access to a given file. Access does not depend on real and effective IDs as it does in the XENIX environment. Under DOS, the real and effective IDs are ignored. The *amode* parameter can be any combination of the values:

```
04  Read
02  Write
00  Check for existence
```

The "Execute" access mode (01) is not allowed.

The EROFS and ETXTBSY error values are not used.

chdir

The *chdir* routine causes the named directory to become the current working directory just as it does in the XENIX environment. The only difference is that the directory pathname under DOS must use the backslash separator (\) and not the slash (/).

chmod

The *chmod* routine can set the "owner" access permissions for a given file, but all other permissions settings are ignored. The *mode* parameter can be:

```
00400  Read by owner
00200  Write by owner
00600  Read and write by owner
```

If write permission is not given, the file is treated as a read-only file.

The *chmod* routine under DOS is not affected by real or effective IDs.

The EPERM and ETXTBSY error values are not used.

chsize

The *chsize* routine changes the size of the given file just as it does in the XENIX environment. However, the maximum size of a file is not affected by the limit defined by the *ulimit*(S) routine. There is no *ulimit* routine for the DOS environment.

creat

The *creat* routine creates a new file or prepares an existing file for writing. If the file is created, the access permissions are set as defined by *mode*. Only "owner" permissions are allowed (see "chmod" above). Ownership of the file is not affected by the real and effective user and group IDs. (These are ignored under DOS).

The EROFS and ETXTBSY error values are not used by *creat* under DOS.

As in the XENIX environment, use of the *open* routine is preferred over *creat* when creating or opening files in the DOS environment.

exec

The *execl*, *execle*, *execlp*, *execv*, *execve*, and *execvp* routines do not overlay the calling process as in the XENIX environment. Instead they cause the named process file to be copied into whatever memory is currently available. If there is not enough memory for the new process, the exec routine will fail and return to the calling process. Otherwise, the new process begins execution.

Under DOS, the exec routines *do not*:

- Use the close-on-exec flag to determine open files for the new process.
- Use the set user and group ID access permissions of the new process file to determine effective user and group IDs.
- Set up signal processing for the new process.
- Disable profiling for the new process (profiling is not allowed under DOS).
- Give the new process attributes inherited from the calling process.
- Use the ETXTBSY error value.

The combined size of all arguments in an exec routine under DOS must not exceed 128 bytes.

exit

The *exit* function terminates the current process and makes the low order byte of *status* available to the parent process. When the process is terminated, all buffers are flushed and released. Also, all open files in the calling process are closed. If the parent process was not waiting on the current process, the *status* value is lost.

fopen, fdopen, freopen

The *fopen*, *fdopen*, and *freopen* routines open stream files just as they do in the XENIX environment. However, there are the following additional values for the *type* string:

t	Opens the file in **text** mode. Opening a file in this mode causes the low-level I/O routines to translate carriage return/linefeed (CR-LF) character combinations into a single linefeed (LF) on input. Similarly on output, linefeeds are translated into CR-LF combinations.
b	Opens the file in **binary** mode. This mode suppresses translation.

For example, the call:

```
fopen("test.dat", "rt");
```

opens a file for reading in **text** mode.

If "t" or "b" is not given in the *type* string, then the mode is defined by the default mode variable *__fmode*. If *__fmode* is 0, the default mode is **text**. If the higher order bit of *__fmode* is 1, the default mode is **binary**.

fread

The *fread* routine removes carriage returns (CR) in all CR-LF pairs read from the input stream if the stream was opened in **text** mode. See "fopen" above.

fseek

The *fseek* routine moves the file pointer to the given position just as in the XENIX environment. However, since DOS uses the carriage return/linefeed (CR-LF) character combinations for newline characters (XENIX uses only an LF), an *fseek* call which moves the file pointer a specific number of bytes past newline characters will not move the pointer to same place in the DOS file as it does in the XENIX file. For example, if a file contains the characters:

```
abcdef\n09123
```

(where **\n** is the newline character) and the file pointer is currently at the letter "a," then the call:

```
fseek(stream, 8, 1);
```

moves the pointer to the digit "0," if the file is a DOS file, or to the digit "9," if the file is a XENIX file.

Note that some *fseek* calls treat DOS and XENIX files identically. For example:

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

always moves the file pointer to the end of the file.

fwrite

The *fwrite* routine replaces every linefeed (LF) character written to the output stream with a carriage return/linefeed (CR-LF) pair if the stream was opened in **text** mode. See "fopen" above.

getpid

The *getpid* routine returns a unique number. Although the number may be used to uniquely identify the process, it does not have the same meaning as the process ID returned by *getpid* in the XENIX environment.

isatty

The *isatty* routine indicates whether or not the given file descriptor is associated with any character device not just a terminal. A character devices can be a console, printer, or serial port.

lseek

The *lseek* routine is similar to the *fseek* routine under DOS whenever the given file descriptor has been opened in **text** mode. In other words, *lseek* must move the file pointer one additional byte for each newline character in the DOS file in order to move the file pointer to the same position in the XENIX file. See "fseek" for more details.

mktemp

The *mktemp* routine creates a temporary filename, using a unique number instead of a process ID. The number is the same as returned by *getpid* (see "getpid" above).

open

The *open* routine opens a file descriptor for a named file, just as in the XENIX environment. However, there is one additional *oflag* value, O_BINARY, and two values, O_NDELAY and O_SYNCW, have been removed.

The O__BINARY flag causes the file to be opened in the opposite mode specified by the *__fmode* variable (see "fopen" above). For example, if the default mode is **text** (*__fmode* is 0), then using O__BINARY opens the file in **binary** mode. If not used, the file is opened in **text** mode.

The EISDIR, EROFS, ETXTBSY, and ENXIO error values are not used.

read

The *read* routine reads characters from the file given by a file descriptor just as in the XENIX environment. However, if the file has been opened in **text** mode (see "Open" above), *read* will replace each CR-LF pair read from the file with a single LF character. The number of bytes returned is the number of bytes remaining after the the CR-LF pairs have been replaced. Thus, the return value may not always correspond with the actual number of bytes read. This is considered normal and has no implications as far as detecting the end of the file.

sbrk

The *sbrk* routine performs the same task as in the XENIX environment. However, *sbrk* is not affected by the limits imposed by *ulimit*(S), since no *ulimit* routine exists for DOS.

signal

The *signal* routine can only handle the SIGINT signal. In DOS, SIGINT is defined to be INT 23H (the Ctrl-C signal).

stat, fstat

The *stat* and *fstat* routines return a structure defining the current status of given the file. The structure members have the following names and meaning:

st_mode	User read/write/execute bits are set. The execute bit is inferred from the filename extension. These are copied into the group and other bits. All files have either the S_IFREG or S_IFDIR bits set. See *stat.h* in "Common Include Files" above.
st_ino	Not used.
st_dev	Drive number of the disk containing the file.
st_rdev	Drive number of the disk containing the file.
st_nlink	Always 1.
st_uid	Not used.
st_gid	Not used.
st_size	Size of the file in bytes.
st_atime	Time of last modification of file.
st_mtime	Time of last modification of file.
st_ctime	Time of last modification of file.
st_dosattr	Contains DOS file attributes.

The *fstat* routine returns less useful information since DOS does not make as much information available for file descriptors as it does full pathnames. The *fstat* routine can detect device files, but it must not be used with directories. The structure returned by *fstat* has the following members:

st_mode	Always read and write for everyone. The S_IFCHR flag is set if this is a device. Otherwise, the S_IFREG bit is set. See *stat.h* in "Common Include Files" above.
st_ino	Not used.
st_dev	Either drive number of the disk containing the file, or file descriptor if this file is a device.
st_rdev	Either drive number of the disk containing the file, or file descriptor if this file is a device.
st_nlink	Always 1.
st_uid	Not used.
st_gid	Not used.
st_size	Size of the file in bytes.
st_atime	Time of last modification of file.
st_mtime	Time of last modification of file.
st_ctime	Time of last modification of file.
st_dosattr	Contains DOS file attributes.

system

The *system* routine passes the given string the the operating system for execution. In order to execute this string, the full pathname of the directory containing the DOS "COMMAND.COM" program must be assigned to the COMSPC environment variable, or assigned to the PATH environment variable. The call will return an error if "COMMAND.COM" cannot be found using these variables.

umask

The *umask* routine can set a mask for "owner" read and write access permissions only. All other permissions are ignored.

unlink

The *unlink* routine always deletes the given file. Since DOS does not allow multiple "links" to the same file, unlinking a file is the same as deleting it.

The EBUSY, ETXTBSY, and EROFS error values are not used.

write

The *write* routine writes a specified number of characters to the file named by the given file descriptor just as in the XENIX environment. However, if the file has been opened in **text** mode (see "Open" above), every LF character in the output is replaced by a CR-LF pair before being written. This does not affect the return value.

Differences in Definitions

Many of the special definitions given in Introduction(S) in the IBM Personal Computer *XENIX Software Command Reference* do not apply to the common routines when used in the DOS environment. The following is a list of the differences.

The *process* ID is still a unique integer, but does not have the same meaning as in the XENIX environment.

The *parent process*, *process group*, *tty group*, *real user*, *real group*, *effective user* and *effective group* IDs are not used by the common routines when run under DOS. Furthermore, there is no *super-user* or *special processes* in the DOS environment.

Filenames in DOS have two parts: a filename and a filename extension. Filenames may be any combination of upto eight letters or digits. Filename extensions may be any combination of upto three letters or digits, preceded by a period (.).

Pathnames in DOS may be any combination of directory names separated by a backslash (\). The slash (/) used in the XENIX environment is not allowed. Directory names may be any combination of upto eight letters or digits. The special names "." and ".." refer to the current directory and the parent directory, respectively.

Drive names may be used at the begin of a pathname to specify a specific disk drive or device. Drives names are generally a letter or combination of letters and digits followed by a colon (:).

Access permissions in DOS are restricted to read and write by the owner of the file. Since all users own all files in DOS, access permissions do little more than define whether or not the file is a read-only file or can be modified. Execution permission and other permissions defined for files in the XENIX environment do not apply the files in the DOS environment.

DOS-specific Routines

The DOS-specific routines are intended for programs being compiled in the XENIX environment, but which are to be executed in the DOS environment only. These routines are not available for use in the XENIX environment. The following sections describe the routines in detail.

eof

```
int eof(fildes)
int fildes;
```

The *eof* function returns the value 1 if the current position of the file associated with *fildes* is at the end-of-file, otherwise the function returns 0. The return value -1 indicates an error.

fcloseall

```
int fcloseall( )
```

The *fcloseall* function closes all currently open streams, except **stdin**, **stdout**, and **stderr**. The function flushes all file buffers before closing, and although it releases system-allocated buffers, it does not release buffers allocated using *setbuf*.

The *fcloseall* function returns the total number of streams closed. The return value -1 indicates an error.

fgetchar

```
#include <stdio.h>

int fgetchar( )
```

The *fgetchar* function reads a single character from the standard input stream **stdin**. The *fgetchar* function is the function version of the macro *getchar*.

The *fgetchar* function returns the character read, or EOF when end-of-file is reached.

filelength

```
long filelength(fildes)
int fildes;
```

The *filelength* function returns the length, in bytes, of the file associated with *fildes*. The return value -1 indicates an error.

flushall

```
int flushall()
```

The *flushall* function flushes the buffers of all currently open output streams. All streams remain open after the call.

The *flushall* function returns the total number of open streams (both input and output streams). There is no error return.

Note that buffers are automatically flushed when they are full, when the associated files are closed, or when a program terminates without closing the files.

fputchar

```
#include <stdio.h>

int fputchar(c)
char c;
```

The *fputc* function writes the single character *c* to the output stream **stdout**. The *fputchar* function is the function version of the macro *putchar*.

The *fputchar* function returns the character written. The return value EOF indicates an error.

itoa, ltoa, and ultoa

```
char *itoa(value, string, radix)
int value;
char *string;
int radix;

char *ltoa(value, string, radix)
long value;
char *string;
int radix;

char *ultoa(value, string, radix)
unsigned long value;
char *string;
int radix;
```

The *itoa*, *ltoa*, and *toa* functions convert the given *value* to a character string that represents that value. The resulting string is stored in *string*, and consists of one or more digits from the numeric base given by *radix*.

The *itoa* function converts type **int** values into strings, *ltoa* converts type **long** values, and *toa* converts type **unsigned long** values. The *radix* can be any in the range 2-36. If *radix* equals 10 and *value* is negative, the first character of the stored string is the minus sign (-).

All functions return a pointer to the new string. There is no error return, and no overflow checking is performed.

labs

```
long labs(value)
long value;
```

The *labs* function returns the absolute value of the type **long** number given by *value* . There is no error return.

mkdir

```
int mkdir(pathname)
char *pathname;
```

The *mkdir* function creates a new directory with the specified *pathname*. The last component of *pathname* names the new directory; the preceding components must identify an existing directory.

The *mkdir* function returns the value 0 if the new directory was created. The return value -1 indicates an error.

rmdir

```
int rmdir(pathname)
char *pathname;
```

The *rmdir* function deletes the directory specified by *pathname*. The directory must be empty, and it must not be the current working directory or the root directory.

The *rmdir* function returns the value 0 if the directory is successfully deleted. The return value -1 indicates an error.

spawn

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

int spawnl(modeflag, pathname, arg0,  . . . , argn, NULL)
int modeflag;
char *pathname, *arg0,  . . . , *argn;

int spawnle(modeflag, pathname, arg0,  . . . , argn, NULL, envp)
int modeflag;
char *pathname, *arg0,  . . . , *argn, *envp[\ ];
```

(Example continues on next page.)

```
int spawnlp(modeflag, filename, arg0,  . . . , argn, NULL)
int modeflag;
char *filename, *arg0,  . . . , *argn;

int spawnv(modeflag, pathname, argv)
int modeflag;
char *pathname, *argv[\ ];

int spawnve(modeflag, pathname, argv, envp)
int modeflag;
char *pathname, *argv[\ ], *envp[\ ];

int spawnvp(modeflag, filename, argv)
int modeflag;
char *filename, *arv[\ ];
```

The *spawn* functions load and execute new child processes. The *pathname* or *filename* argument names the executable file to be loaded. The *arg n* or *argv* arguments contain pointers to character strings to be passed to the new process. The *modeflag* argument defines the execution of the parent process after placing a call to a *spawn* function. The *envp* argument allows the user to alter the environment for the child process by passing a list of environment settings. The *spawnl*, *spawnle* and *spawnlp* functions are typically used in cases where the number of arguments is known in advance. The *spawnv*, *spawnve*, and *spawnvp* functions are useful when the number of arguments to the new process is variable. Pointers to the arguments are passed as an array, *argv*, which accommodates any number of elements.

The *modeflag* values are defined in the include file *spawn.h*. The following lists the meaning of each value:

Modeflag	**Meaning**
P__WAIT	Suspend parent process until execution of child process is complete.
P__NOWAIT	Continue to execute parent process concurrently with child process.
P__OVERLAY	Overlay parent process with child process.

When P__WAIT or P__NOWAIT is specified, there must be sufficient memory available for loading and executing the child process. If P__OVERLAY is specified, the parent process is destroyed and control cannot be returned to it. This is similar to the effect of the *exec* routines. Only P__WAIT and P__OVERLAY may be used under DOS 2.0. P__NOWAIT is reserved for future implementations, and use of this flag with DOS 2.0 will produce an error.

The *pathname* argument must be the full directory pathname for the file to be loaded. The *filename* argument (in the *spawnlp* and *spawnvp* functions) may be just the filename or a partial pathname for the file; the current value of the environment variable PATH is used to determine which directories are searched for this file.

The *arg n* arguments in the *spawnl*, *spawnle*, and *spawnlp* functions must be pointers to null-terminated character strings. These strings form the argument list for the child process. Their combined length must not exceed 128 bytes. (Terminating null characters (\0) are not counted.) Thus, any number of *arg n* arguments may be given, as long as the character count of the corresponding strings does not exceed 128. The NULL pointer value must mark the end of the *arg n* argument list.

The *argv* arguments in the *spawnv*, *spawnve*, and *spawnvp* functions must be pointers to a single array of pointers to the character strings. The combined length of the strings must not exceed 128 bytes. The NULL pointer value must be placed in the array element immediately following the element containing the last character string.

By convention, the *arg0* and *argv [0]* arguments should be a copy of the *pathname* or *filename* argument. A different value will not produce an error.

The *envp* argument in the *spawnle* and *spawnve* functions must be an array of character pointers, each element of which points to a null-terminated string defining an environment variable. An environment setting has the following form:

name=value

where *name* is the name of an environment variable and *value* is the string value to which that variable is set. Notice that *value* is not enclosed in double quotes. When *envp* is NULL, the child process inherits the environment settings of the parent process.

Files that are open when a call to a *spawn* function is made remain open in the new process. In the *spawnl*, *spawnlp*, *spawnv*, and *spawnvp* functions, the child process inherits the environment of the parent.

Return Values

If the P_WAIT is specified, the return value is the exit status of the child process. The exit status is 0 if the process terminated normally. A positive exit status indicates an abnormal exit through an *abort* function call or an interrupt. The exit status may also be set to a non-zero value if the child process specifically calls the *exit* function with a non-zero argument.

If P_OVERLAY is specified and the child is successfully loaded, the routine never returns a value.

The return value -1 indicates an error (the child process is not started). The value -1 is also returned when P_NOWAIT is specified under DOS 2.0.

strlwr and strupr

```
char *strlwr(string)
char *string;

char *strupr(string)
char *string;
```

The *strlwr* function converts any uppercase letters in the given *string* to lowercase.

The *strupr* function converts any lowercase letters in the given *string* to uppercase.

The *strlwr* and *strupr* functions return a pointer to the converted *string*. There is no error return.

strset and strnset

```
char *strset(string, c)
char *string, c;

char *strnset(string, c, n)
char *string, c;
unsigned int n;
```

The *strset* function sets all characters in the given *string* (except the terminating null character) to the character *c* and returns a pointer to the altered string.

The *strnset* function sets the first *n* characters of *string* to the character *c* and returns a pointer to the altered *string*. If *n* is greater than the length of a given string, the string length is used instead.

strrev

```
char *strrev(string)
char *string;
```

The *strrev* function reverses the order of the characters in the given *string*. The terminating null character (\0) remains in place.

The *strrev* function returns a pointer to the altered *string*. There is no error return.

tell

```
long tell(fildes)
int fildes;
```

The *tell* function returns the current position of the file associated with *filedes.* The position is the number of bytes from the beginning of the file. The return value of -1 indicates an error.

Index

IBM

The Personal Computer
Programming Family

Reader's Comment Form

XENIX™ 6138873
Programmer's Guide to
Library Functions

Your comments assist us in improving the usefulness of our publication; they are an important part of the input used for revisions.

IBM may use and distribute any of the information you supply in any way it believes appropriate without incurring any obligation whatever. You may, of course, continue to use the information you supply.

Please do not use this form for technical questions regarding the IBM Personal Computer or programs for the IBM Personal Computer, or for requests for additional publications; this only delays the response. Instead, direct your inquiries or request to your authorized IBM Personal Computer dealer.

Comments:

NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES

BUSINESS REPLY MAIL

FIRST CLASS **PERMIT NO. 321** **BOCA RATON, FLORIDA 33432**

POSTAGE WILL BE PAID BY ADDRESSEE

IBM PERSONAL COMPUTER
SALES & SERVICE
P.O. BOX 1328-C
BOCA RATON, FLORIDA 33432

Fold here

Tape

Please do not staple

Tape