2. PROGRAMMING BASICS

Introduction

The information in this chapter is for anyone just learning to write programs to run in a SYSTEM V/68 environment. In Chapter 1 we identified one group of users as single-user programmers. People in that category, particularly those who are not deeply interested in programming, may find this chapter (plus related reference manuals) tells them as much as they need to know about coding and running programs in the SYSTEM V/68 environment.

Programmers whose interest runs deeper, who are part of an application development project, or who are producing programs on one computer that are being ported to another should view this chapter as a starter package.

Choosing a Programming Language

How do you decide which programming language to use in a given situation? One answer could be, "I always code in HAIRBOL, because that's the language I know best." Actually, in some circumstances that's a legitimate answer. But assuming more than one programming language is available to you, that different programming languages have their strengths and weaknesses, and that once you've learned to use one programming language it becomes relatively easy to learn to use another, you might approach the problem of language selection by asking yourself questions like the following:

- What is the nature of the job this program is to do?

  Does the task call for the development of a complex algorithm, or is this a simple procedure that has to be done on many records?

- Does the programming task have many separate parts?

  Can the program be subdivided into separately compilable functions, or is it one module?

- How soon does the program have to be available?

  Is it needed right now, or do I have enough time to work out the most efficient process possible?

- What is the scope of its use?

  Am I the only person who will use this program, or is it going to be distributed to the whole world?

- Is there a possibility the program will be ported to other systems?
- What is the life expectancy of the program?

  Is it going to be used just a few times, or will it still be going strong five years from now?

Supported Languages in an Operating System Environment

By "supported languages," we mean those offered for use on VME-based computers running the SYSTEM V/68 operating system Release 3. Since these are separately purchasable items, not all of them will necessarily be installed on your machine. On the other hand, you may have languages available on your machine that came from another source and are not mentioned in this discussion. In this section and the one to follow we give brief descriptions of the nature of a) six full-scale programming languages and b) several special-purpose languages. (As an adjunct to these supported languages, of course, don't overlook the considerable capabilities of shell procedures.)

C Language

The C language is intimately associated with the operating system. If you need to use many function calls for low-level I/O, memory or device management, or inter-process communication, C language is a logical first choice. Since most programs don't require such direct interfaces with the operating system, however, the decision to choose C might better be based on one or more of the following characteristics:

- a variety of data types: character, integer, long integer, float, and double
- low-level constructs (most of the operating system kernel is written in C)
- derived data types, such as arrays, functions, pointers, structures, and unions
- multi-dimensional arrays
- scaled pointers and the ability to do pointer arithmetic
- bit-wise operators

- a variety of flow-of-control statements: if, if-else, switch, while, do-while, and for
- a high degree of portability

C is a language that lends itself readily to structured programming. It is natural in C to think in terms of functions. The next logical step is to view each function as a separately compilable unit. This approach (coding a program in small pieces) eases the job of making changes and/or improvements. As you create functions for one program you will surely find that many can be picked up, or quickly revised, for another program.

A difficulty with C is that it takes a fairly concentrated use of the language over a period of several months to reach your full potential as a C programmer. If you are a casual programmer, you might want to choose a less demanding language.

FORTRAN

The oldest of the high-level programming languages, FORTRAN is still highly prized for its variety of mathematical functions. If you are writing a program for statistical analysis or other scientific applications, FORTRAN is a good choice. An original design objective was to produce a language with good operating efficiency. This has been achieved at the expense of some flexibility in the area of type definition and data abstraction. There is, for example, only a single form of the iteration statement. FORTRAN also requires using a somewhat rigid format for input of lines of source code. This shortcoming may be overcome by using one of the tools designed to make FORTRAN more flexible.

Pascal

Originally designed as a teaching tool for block-structured programming, Pascal has gained a wide acceptance because of its straightforward style. Pascal is highly structured and allows system-level calls (characteristics it shares with C). Since the intent of the developers, however, was to produce a language to teach people about programming, it is perhaps best suited to small projects. Among its inconveniences are its lack of facilities for specifying initial values for variables and limited file-processing capability.

COBOL

Probably more programmers are familiar with COBOL than with any other single programming language. It is frequently used in business applications because its strengths lie in the management of input/output and in defining record layouts.

It is somewhat cumbersome to use COBOL for complex algorithms, but it works well in cases where many records have to be passed through a simple process (a payroll withholding tax calculation, for example). It is a rather tedious language to work with, because each program requires a lengthy amount of text merely to describe record layouts, processing environment, and variables used in the code. The COBOL language is wordy, so the compilation process is often complex. Once written and put into production, COBOL programs have a way of staying in use for years; and what might be thought of by some as wordiness comes to be considered self-documentation. The investment in programmer time often makes them resistant to change.

BASIC

The most commonly heard comment about BASIC is that it is easy to learn. With the spread of personal microcomputers many people have learned BASIC because it is simple to produce runnable programs in very little time. It is difficult, however, to use BASIC for large programming projects. It lacks the provision for structured flow-of-control, requires that every variable used be defined for the entire program, and has no way of transferring values between functions and calling programs. Most versions of BASIC run as interpreted rather than compiled code. That makes for slower-running programs. Despite its limitations, however, it is useful for getting simple procedures into operation quickly.

Assembly Language

The closest approach to machine language, assembly language is specific to the particular computer on which your program is to run. High-level languages are translated into the assembly language for a specific processor as one step of the compilation. The most common need to work in assembly language arises when you want to do some task that is not within the scope of a high-level language. Since assembly language is machine-specific, programs written in it are not portable.

Special-Purpose Languages

In addition to the above formal programming languages, the operating system may offer one or more of the special-purpose languages described below.

NOTE

Since SYSTEM V/68 utilities and commands are packaged in functional groupings, it is possible that not all the facilities mentioned will be available on all systems.

awk

awk (its name is an acronym constructed from the initials of its developers) scans an input file for lines that match pattern(s) described in a specification file. On finding a line that matches a pattern, **awk** performs actions also described in the specification. It is not uncommon that an **awk** program can be written in a couple of lines to do functions that would take a couple of pages to describe in a programming language like FORTRAN or C. For example, consider a case where you have a set of records that consist of a key field and a second field that represents a quantity. You have sorted the records by the key field, and you now want to add the quantities for records with duplicate keys and output a file in which no keys are duplicated. The pseudo-code for such a program might look like this:

```
Read the first record into a hold area;
Read additional records until EOF;
 {
 If the key matches the key of the record in the hold area,
  add the quantity to the quantity field of the held record;
 If the key does not match the key of the held record,
  write the held record,
  move the new record to the hold area;
 }
At EOF, write out the last record from the hold area.
```

An **awk** program to accomplish this task would look like this:

```
          { qty[$1] += $2 }
        END { for (key in qty) print key, qty[key] }
```

This illustrates only one characteristic of **awk**: its ability to work with associative arrays. With **awk**, the input file does not have to be sorted, which is a requirement of the pseudo-program.

lex

lex is a lexical analyzer that can be added to C or FORTRAN programs. A lexical analyzer is interested in the vocabulary of a language rather than its grammar, which is a system of rules defining the structure of a language. **lex** can produce C language subroutines that recognize regular expressions specified by the user, take some action when a regular expression is recognized, and pass the output stream on to the next program.

yacc

yacc (Yet Another Compiler Compiler) is a tool for describing an input language to a computer program. **yacc** produces a C language subroutine that parses an input stream according to rules laid down in a specification file. The **yacc** specification file establishes a set of grammar rules together with actions to be taken when tokens in the input match the rules. **lex** may be used with **yacc** to control the input process and pass tokens to the parser that applies the grammar rules.

M4

M4 is a macro processor that can be used as a preprocessor for assembly language and C programs. It is described in Section (1) of the *Programmer's Reference Manual.*

bc and dc

bc enables you to use a computer terminal as you would a programmable calculator. You can edit a file of mathematical computations and call **bc** to execute them. The **bc** program uses **dc**. You can use **dc** directly, if you want to; but it takes a little getting used to since it works with reverse Polish notation. That means you enter numbers into a stack followed by the operator. Both **bc** and **dc** are described in Section (1) of the *User's Reference Manual.*

curses

Actually a library of C functions, **curses** is included in this list because the set of functions just about amounts to a sub-language for dealing with terminal screens. If you are writing programs that include interactive user screens, you will want to become familiar with this group of functions.

After Your Code Is Written

The last two steps in most compilation systems in the SYSTEM V/68 environment are the assembler and the link editor. The compilation system produces assembly language code. The assembler translates that code into the machine language of the computer the program is to run on. The link editor resolves all undefined references and makes the object module executable. With most languages on SYSTEM V/68, the assembler and link editor produce files in what is known as the Common Object File Format (COFF). A common format makes it easier for utilities that depend on information in the object file to work on different machines running different versions of the operating system.

In the Common Object File Format an object file contains:

- a file header
- optional secondary header
- a table of section headers
- data corresponding to the section headers
- relocation information
- line numbers
- a symbol table
- a string table

An object file is made up of sections. Usually, there are at least two: **.text**, and **.data**. Some object files contain a section called **.bss**. (**.bss** is an assembly language pseudo-op that originally stood for "block started by symbol.") The **.bss** section, when present, holds uninitialized data. Options of the compilers cause different items of information to be included in the Common Object File Format. For example, compiling a program with the **-g** option adds line numbers and other symbolic information that is needed for the **sdb** (Symbolic Debugger) command to be fully effective. You can spend many years programming without having to worry too much about the contents and organization of the Common Object File Format, so we are not going into any further depth of detail at this point. Detailed information is available in Chapter 11 of this guide.

Compiling and Link Editing

The command used for compiling depends on the language used;

- for C programs, **cc** both compiles and link edits

- for FORTRAN programs, **f77** both compiles and link edits

Compiling C Programs

To use the C compilation system you must have your source code in a file with a filename that ends in the characters **.c**, as in **mycode.c**. The command to invoke the compiler is:

```
cc mycode.c
```

If the compilation is successful, the process proceeds through the link edit stage. The result will be an executable file named **a.out**.

Several options to the **cc** command are available to control its operation. The most used options are:

-c	causes the compilation system to suppress the link edit phase. This produces an object file (**mycode.o**) that can be link edited at a later time with a **cc** command without the **-c** option.
-g	causes the compilation system to generate special information about variables and language statements used by the symbolic debugger **sdb**. If you are going through the stage of debugging your program, use this option.
-O	causes the inclusion of an additional optimization phase. This option is logically incompatible with the **-g** option. You would normally use **-O** after the program has been debugged, to reduce the size of the object file and increase execution speed.
-p	causes the compilation system to produce code that works with the **prof**(1) command to produce a runtime profile of where the program is spending its time. Useful in identifying which routines are candidates for improved code.
-o *outfile*	tells **cc** to tell the link editor to use the specified name for the executable file rather than the default **a.out**.

Other options can be used with **cc**. Check the *Programmer's Reference Manual.*

If you enter the **cc** command using a file name that ends in **.s**, the compilation system treats it as assembly language source code and bypasses all the steps ahead of the assembly step.

Compiling FORTRAN Programs

The **f77** command invokes the FORTRAN compilation system. The operation of the command is similar to that of the **cc** command, except the source code files must have a **.f** suffix. The **f77** command compiles your source code and calls in the link editor to produce an executable file named **a.out**.

The following command line options have the same meaning as they do for the **cc** command:

> **-c**, **-p**, **-O**, **-g**, and **-o** *outfile*

Loading and Running BASIC Programs

BASIC programs can be invoked in two ways:

- With the command

  basic bscpgm.b

  where **bscpgm.b** is the name of the file that holds your BASIC statements. This tells the operating system to load and run the program. If the program includes a **run** statement naming another program, you will chain from one to the other. Variables specified in the first can be preserved for the second with the **common** statement.

- By setting up a shell script.

Compiler Diagnostic Messages

The C compiler generates error messages for statements that don't compile. The messages are generally quite understandable, but in common with most language compilers they sometimes point several statements beyond where the error occurred. For example, if you inadvertently put an extra ; at the end of an if statement, a subsequent else will be flagged as a syntax error. In the case where a block of several statements follows the if, the line number of the syntax error caused by the else will start you looking for the error well past where it is. Unbalanced curly braces, { }, are another common producer of syntax errors.

Link Editing

The **ld** command invokes the link editor directly. The typical user, however, seldom invokes **ld** directly. A more common practice is to use a language compilation control command (such as **cc**) that invokes **ld**. The link editor combines several object files into one, performs relocation, resolves external symbols, incorporates startup routines, and supports symbol table information used by **sdb**. You may, of course, start with a single object file rather than several. The resulting executable module is left in a file named **a.out**.

Any file named on the **ld** command line that is not an object file (typically, a name ending in **o**) is assumed to be an archive library or a file of link editor directives. The **ld** command has some 16 options. We are going to describe four of them. These options should be fed to the link editor by specifying them on the **cc** command line if you are doing both jobs with the single command, which is the usual case.

-o *outfile* provides a name to be used to replace **a.out** as the name of the output file. Obviously, the name **a.out** is of only temporary usefulness. If you know the name you want use to invoke your program, you can provide it here. Of course, it may be equally convenient to do this:

```
mv a.out progname
```

when you want to give your program a less-temporary name.

-l*x* directs the link editor to search a library **lib***x***.a**, where *x* is up to nine characters. For C programs, **libc.a** is automatically searched if the **cc** command is used. The **-l***x* option is used to bring in libraries not normally in the search path, such as **libm.a**, the math library. The **-l***x* option can occur more than once on a command line, with different values for the *x*. A library is searched when its name is encountered, so the placement of the option on the command line is important. The safest place to put it is at the end of the command line. The **-l***x* option is related to the **-L** option.

-L *dir* changes the **lib***x***.a** search sequence to search in the specified directory before looking in the default library directories, usually **/lib** or **/usr/lib**. This is useful if you have different versions of a library and you want to point the link editor to the correct one. It works on the assumption that once a library has been found no further searching for that library is necessary. Because **-L** diverts the search for the libraries specified by **-l***x* options, it must precede such options on the command line.

-u *symname* enters *symname* as an undefined symbol in the symbol table. This is useful if you are loading entirely from an archive library, because initially the symbol table is empty and needs an unresolved reference to force the loading of the first routine.

When the link editor is called through **cc**, a startup routine (typically **/lib/crt0.o** for C programs) is linked with your program. This routine calls **exit**(2) after execution of the main program.

The link editor accepts a file containing link editor directives. The details of the link editor command language can be found in Chapter 12.

The Operating System/Programming Language Interface

When a program is run in a computer it depends on the operating system for a variety of services. Some of the services such as bringing the program into main memory and starting the execution are completely transparent to the program. They are, in effect, arranged for in advance by the link editor when it marks an object module as executable. As a programmer you seldom need to be concerned about such matters.

Other services, however, such as input/output, file management, and storage allocation do require work on the part of the programmer. These connections between a program and the operating system are what is meant by the term "operating system/language interface." The topics included in this section are:

- How arguments are passed to a program
- System calls and subroutines
- Header files and libraries
- Input/Output
- Processes
- Error Handling, Signals, and Interrupts

Why C Is Used to Illustrate the Interface

Throughout this section, C programs are used to illustrate the interface between the operating system and programming languages because C programs make more use of the interface mechanisms than other high-level languages. What is really being covered in this section, then, is the operating system/C Language interface. The way that other languages deal with these topics is described in the user's guides for those languages.

How Arguments Are Passed to a Program

Information or control data can be passed to a C program as arguments on the command line. When the program is run as a command, arguments on the command line are made available to the function **main** in two parameters, an argument count and an array of pointers to character strings. (Every C program is required to have an entry module by the name of **main**.) Since the argument count is always given, the program does not have to know in advance how many arguments to expect. The character strings pointed at by elements of the array of pointers contain the argument information.

The arguments are presented to the program traditionally as **argc** and **argv**, although any names you choose will work. **argc** is an integer that gives the count of the number of arguments. Since the command itself is considered to be the first argument, **argv[0]**, the count is always at least one. **argv** is an array of pointers to character strings (arrays of characters terminated by the null character \0).

If you plan to pass runtime parameters to your program, you need to include code to deal with the information. Two possible uses of runtime parameters are:

- as control data. Use the information to set internal flags that control the operation of the program.
- to provide a variable filename to the program.

Figures 2-1 and 2-2 show program fragments that illustrate these uses.

```
#include <stdio.h>

main(argc, argv)
  int argc;
  char *argv[];
{
               void exit();
               int oflag = FALSE;
               int pflag = FALSE; /* Function Flags */
               int rflag = FALSE;
               int ch;

               while ((ch = getopt(argc,argv, "opr")) != EOF)
               {
                  /* For options present, set flag to TRUE */
                  /* If no options present, print error message */
               switch (ch)
               {
               case 'o':
               oflag = 1;
               break;
               case 'p':
               pflag = 1;
               break;
               case 'r':
               rflag = 1;
               break;
               default:
               (void)fprintf(stderr,
               "Usage: %s [-opr]\n", argv[0]);
               exit(2);
               }
               }
               .
               .
               .
}
```

Figure 2-1. Using Command Line Arguments to Set Flags

```
#include <stdio.h>

main(argc, argv)
  int argc;
  char *argv[];
{
               FILE *fopen(), *fin;
               void perror(), exit();

               if (argc > 1)
               {
               if ((fin = fopen(argv[1], "r")) == NULL)
               {
                  /* First string (%s) is program name (argv[0]) */
                  /* Second string (%s) is name of file that could */
                  /* not be opened (argv[1]) */

               (void)fprintf(stderr,
                 "%s: cannot open %s: ",
                 argv[0], argv[1]);
               perror("");
               exit(2);
               }
               }
               .
               .
               .
}
```

Figure 2-2. Using **argv**[*n*] Pointers to Pass a Filename

The shell, which makes arguments available to your program, considers an argument to be any nonblank character string separated by blanks or tabs from any adjacent nonblank character strings. Characters enclosed in double quotes ("abc def") are passed to the program as one argument, even if blanks or tabs are among the characters. You are responsible for error checking and otherwise making sure the argument received is what your program expects it to be.

A third argument is also present, in addition to **argc** and **argv**. The third argument, known as **envp**, is an array of pointers to environment variables. You can find more information on **envp** in the *Programmer's Reference Manual* under **exec**(2) and **environ**(5).

System Calls and Subroutines

System calls are requests from a program for an action to be performed by the operating system kernel. Subroutines are precoded modules used to supplement the functionality of a programming language.

Both system calls and subroutines look like functions such as those you might code for the individual parts of your program. There are, however, differences between them:

- At link edit time, the code for subroutines is copied into the object file for your program; the code invoked by a system call remains in the kernel.
- At execution time, subroutine code is executed as if it were code you had written yourself; a system function call is executed by switching from your process area to the kernel.

This means that, while subroutines make your executable object file larger, runtime overhead for context switching may be less and execution may be faster.

Categories of System Calls and Subroutines

System calls divide fairly neatly into the following categories:

- file access
- file and directory manipulation
- process control
- environment control and status information

You can generally tell the category of a subroutine by the section of the *Programmer's Reference Manual* in which you find its manual page. However, the first part of Section 3 (3C and 3S) covers such a variety of subroutines it might be helpful to classify them further.

- The subroutines of subclass 3S constitute the operating system/C Language standard I/O, an efficient I/O buffering scheme for C.
- The subroutines of subclass 3C perform a variety of tasks. They have in common the fact that their object code is stored in **libc.a**. They can be divided into the following categories:
  - — string manipulation
  - — character conversion
  - — character classification

— environment management

— memory management

Table 2-1 lists the functions that compose the standard I/O subroutines in 3S. Since a manual page often describes several related functions, the leftmost column in a row shows the function name that appears at the top of the manual page; any other names in the row are related functions described on the same page. (For all functions: #include <stdio.h>.)

Table 2-2 lists string-handling functions that are grouped under the heading **string**(3C) in the *Programmer's Reference Manual*. Extern definitions of the string functions are provided by **string.h**. (For all functions: #include <string.h>.)

Table 2-3 lists macros that classify ASCII character-coded integer values. These macros are described under the heading **ctype**(3C) in Section 3 of the *Programmer's Reference Manual*. Nonzero return == true; zero return == false. (For all functions: #include <ctype.h>.)

Table 2-4 lists functions that are used to convert integers or strings from one representation to another. Since a manual page often describes several related functions, the leftmost column in a row shows the function name that appears at the top of the manual page; any other names in the row are related functions described on the same page.

Table 2-5 lists functions and macros that are used to translate characters (see **conv**(3C)). For the two macros: #include <ctype.h>.

TABLE 2-1. C Language Standard I/O Subroutines

Function Names				Purpose
fclose	**fflush**			Close or flush a stream.
ferror	**feof**	**clearerr**	**fileno**	Stream status inquiries.
fopen	**freopen**	**fdopen**		Open a stream.
fread	**fwrite**			Binary input/output.
fseek	**rewind**	**ftell**		Reposition a file pointer in a stream.
getc	**getchar**	**fgetc**	**getw**	Get a character or word from a stream.
gets	**fgets**			Get a string from a stream.
popen	**pclose**			Begin or end a pipe to/from a process.
printf	**fprintf**	**sprintf**		Print formatted output.
putc	**putchar**	**fputc**	**putw**	Put a character or word on a stream.
puts	**fputs**			Put a string on a stream.
scanf	**fscanf**	**sscanf**		Convert formatted input.
setbuf	**setvbuf**			Assign buffering to a stream.
system				Issue a command through the shell.
tmpfile				Create a temporary file.
tmpnam	**tempnam**			Create a name for a temporary file.
ungetc				Push character back into input stream.
vprintf	**vfprintf**	**vsprintf**		Print formatted output of a varargs argument list.

TABLE 2-2. String Operations

Function Name	Operation
strcat(s1, s2)	Append a copy of s2 to the end of s1.
strncat(s1, s2, n)	Append n characters from s2 to the end of s1.
strcmp(s1, s2)	Compare two strings. Return an integer less than, greater than, or equal to 0 to show that s1 is lexicographically less than, greater than, or equal to s2.
strncmp(s1, s2, n)	Compare n characters from two strings. Results are otherwise identical to strcmp.
strcpy(s1, s2)	Copy s2 to s1, stopping after the null character (\0) has been copied.
strncpy(s1, s2, n)	Copy n characters from s2 to s1. Truncate s2 if it is longer than n, or pad it with null characters if it is shorter than n.
strdup(s)	Return a pointer to a new string that is a duplicate of the string pointed to by s.
strchr(s, c)	Return a pointer to the first occurrence of character c in string s, or return a NULL pointer if c is not in s.
strrchr(s, c)	Return a pointer to the last occurrence of character c in string s, or return a NULL pointer if c is not in s.
strlen(s)	Return the number of characters in s up to the first null character.
strpbrk(s1, s2)	Return a pointer to the first occurrence in s1 of any character from s2, or return a NULL pointer if no character from s2 occurs in s1.
strspn(s1, s2)	Return the length of the initial segment of s1, which consists entirely of characters from s2.
strcspn(s1, s2)	Return the length of the initial segment of s1, which consists entirely of characters not from s2.
strtok(s1, s2)	Look for occurrences of s2 within s1.

TABLE 2-3. Classifying ASCII Character-Coded Integer Values

Macro Name	Class Defined
isalpha(c)	Is *c* a letter?
isupper(c)	Is *c* an upper-case letter?
islower(c)	Is *c* a lower-case letter?
isdigit(c)	Is *c* a digit [0-9]?
isxdigit(c)	Is *c* a hexadecimal digit [0-9], [A-F] or [a-f]?
isalnum(c)	Is *c* an alphanumeric (letter or digit)?
isspace(c)	Is *c* a space, tab, carriage return, new line, vertical tab or form feed?
ispunct(c)	Is *c* a punctuation character (neither control nor alphanumeric)?
isprint(c)	Is *c* a printing character, code 040 (space) through 0176 (tilde)?
isgraph(c)	Same as isprint except false for 040 (space)?
iscntrl(c)	Is *c* a control character (less than 040) or a delete character? (0177)
isascii(c)	Is *c* an ASCII character (code less than 0200)?

TABLE 2-4. Conversion Functions for Integers and Strings

Function Names			Conversion
a64l	**l64a**		Between long integer and base-64 ASCII string.
ecvt	**fcvt**	**gcvt**	Floating-point number to string.
l3tol	**ltol3**		Between 3-byte integer and long integer.
strtod	**atof**		String to double-precision number.
strtol	**atol**	**atoi**	String to integer.

TABLE 2-5. Character Translation Functions and Macros

Function or Macro	Translation
toupper	Lowercase to uppercase.
_toupper	Macro version of toupper.
tolower	Uppercase to lowercase.
_tolower	Macro version of tolower.
toascii	Other system to ASCII. (Turns off all bits that are not part of a standard ASCII character; intended for compatibility with other systems.)

Where to Find Manual Pages

System calls are listed alphabetically in Section 2 of the *Programmer's Reference Manual*. Subroutines are listed in Section 3.

We have described above what is in the first subsection of Section 3. The remaining subsections of Section 3 are:

- 3M—functions that make up the Math Library, **libm**
- 3X—various specialized functions
- 3F—the FORTRAN intrinsic function library, **libF77**
- 3N—Networking Support Utilities

Using System Calls and Subroutines in C Programs

Information about the proper way to use a system call or subroutine is given on its manual page, but you have to know what you are looking for before it begins to make sense. To illustrate, a typical manual page (for **gets**(3S)) is shown in Figure 2-3.

NAME

gets, fgets - get a string from a stream

SYNOPSIS

```
#include <stdio.h>

char   *gets (s)
char   *s;

char   *fgets (s, n, stream)
char   *s;
int    n;
FILE    *stream;
```

DESCRIPTION

Gets reads characters from the standard input stream, *stdin,* into the array pointed to by *s,* until a new-line character is read or an end-of-file condition is encountered. The new-line character is discarded and the string is terminated with a null character.

Fgets reads characters from the *stream* into the array pointed to by *s,* until *n*-1 characters are read, or a new-line character is read and transferred to *s,* or an end-of-file condition is encountered. The string is then terminated with a null character.

SEE ALSO

ferror(3S), fopen(3S), fread(3S), getc(3S), scanf(3S).

DIAGNOSTICS

If end-of-file is encountered and no characters have been read, no characters are transferred to *s* and a NULL pointer is returned. If a read error occurs, such as trying to use these functions on a file that has not been opened for reading, a NULL pointer is returned. Otherwise *s* is returned.

Figure 2-3. Manual Page for **gets**(3S)

As you can see from the illustration, two related functions are described on this page: **gets** and **fgets**. Each function gets a string from a stream in a slightly different way. The DESCRIPTION section tells how each operates.

It is the SYNOPSIS section, however, that contains the critical information about how the function (or macro) is used in your program. Notice that the first line in the SYNOPSIS is:

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

This means that to use **gets** or **fgets** you must bring the standard I/O header file into your program (generally right at the top of the file). There is something in **stdio.h** that is needed when you use the described functions. Figure 2-4 shows a version of **stdio.h**. Check it to see if you can understand what **gets** or **fgets** uses.

The next thing shown in the SYNOPSIS section of a page for system calls or subroutines is the formal declaration of the function. The formal declaration tells you:

- **the type of object returned by the function**

  In our example, both **gets** and **fgets** return a character pointer.

- **the object or objects the function expects to receive when called**

  These are the things enclosed in the parentheses of the function. **gets** expects a character pointer. (The DESCRIPTION section sheds light on what the tokens of the formal declaration stand for.)

- **how the function is going to treat those objects**

  The declaration:

  ```
  char *s;
  ```

  in **gets** means that the token **s** enclosed in the parentheses will be considered to be a pointer to a character string. Bear in mind that in C language the name of an array, when passed as an argument, is converted to a pointer to the beginning of the array.

We have chosen a simple example here in **gets**. If you want to test yourself on something a little more complex, try working out the meaning of the elements of the **fgets** declaration.

While we're on the subject of **fgets**, there is another piece of C esoterica that we'll explain. Notice that the third parameter in the **fgets** declaration is referred to as **stream**. A **stream**, in this context, is a file with its associated buffering. It is declared to be a pointer to a defined type FILE. Where is FILE defined? Right! In **stdio.h**. (See Figure 2-4.)

```
#ifndef _NFILE
#define _NFILE 20

#define BUFSIZ 1024
#define _SBFSIZ 8

typedef struct {
                 int           _cnt;
                 unsigned char *_ptr;
                 unsigned char *_base;
                 char          _flag;
                 char          _file;
} FILE;

#define _IOFBF      0000  /* _IOLBF means that a file's output  */
#define _IOREAD     0001  /* will be buffered line by line.     */
#define _IOWRT      0002  /* In addition to being flags, _IONBF,*/
#define _IONBF      0004  /* _IOLBF and IOFBF are possible      */
#define _IOMYBUF    0010  /* values for "type" in setvbuf.      */
#define _IOEOF      0020
#define _IOERR      0040
#define _IOLBF      0100
#define _IORW       0200

#ifndef NULL
#define NULL        0
#endif
#ifndef EOF
#define EOF         (-1)
#endif
```

Figure 2-4. A Version of **stdio.h** (part 1 of 2)

```
#define stdin       (&_iob[0])
#define stdout      (&_iob[1])
#define stderr      (&_iob[2])

#define _bufend(p)  _bufendtab[(p)->_file]
#define _bufsiz(p)  (_bufend(p) - (p)->_base)

#ifndef lint
#define getc(p)     (--(p)->_cnt < 0 ? _filbuf(p) : (int) *(p)->_ptr++)
#define putc(x, p)  (--(p)->_cnt < 0 ?
                        _flsbuf((unsigned char) (x), (p)) :
                        (int) (*(p)->_ptr++ = (unsigned char) (x)))
#define getchar()   getc(stdin)
#define putchar(x)  putc((x), stdout)
#define clearerr(p) ((void) ((p)->_flag &=  (_IOERR | _IOEOF)))
#define feof(p)     ((p)->_flag & _IOEOF)
#define ferror(p)   ((p)->_flag & _IOERR)
#define fileno(p)   (p)->_file
#endif

extern FILE _iob[_NFILE];
extern FILE *fopen(), *fdopen(), *freopen(), *popen(), *tmpfile();
extern long ftell();
extern void rewind(), setbuf();
extern char *ctermid(), *cuserid(), *fgets(), *gets(), *tempnam(), *tmpnam();
extern unsigned char *_bufendtab[];

#define L_ctermid   9
#define L_cuserid   9
#define P_tmpdir    "/usr/tmp/"
#define L_tmpnam    (sizeof(P_tmpdir) + 15)
#endif
```

Figure 2-4. A Version of **stdio.h** (part 2 of 2)

To finish off this discussion of the way you use functions described in the *Programmer's Reference Manual* in your own code, Figure 2-5 shows a program fragment in which **gets** is used.

```
#include <stdio.h>

main()
{
     char sarray[80];

     for(;;)
     {
      if (gets(sarray) != NULL)
      .
      .       /* Do something with the string */
      .
     }
}
```

Figure 2-5. How **gets** Is Used in a Program

You might ask, "Where is **gets** reading from?" The answer is, "From the standard input." That generally means 1) from something being keyed in from the terminal where the command was entered to get the program running, or 2) from output of another command that was piped to **gets**. How do we know that? The DESCRIPTION section of the **gets** manual page says, "*Gets* reads characters from the standard input...." Where is the standard input defined? In **stdio.h**.

Header Files and Object File Libraries

The following paragraphs discuss header files and object file libraries.

Header Files

In the earlier parts of this chapter there have been frequent references to **stdio.h**, and a version of the file itself is shown in Figure 2-4. This is the most commonly used header file in the C environment, but there are many others.

Header files carry definitions and declarations that are used by more than one function. Header file names traditionally have the suffix **.h** and are brought into a program at compile time by the C-preprocessor. The preprocessor does this because it interprets the **#include** statement in your program as a directive, as indeed it is. All keywords preceded by a pound sign (#) at the beginning of the line are treated as preprocessor directives. The two most commonly used directives are **#include** and **#define**. We have already seen that the **#include** directive is used to call in (and process) the contents of the named file. The

#define directive is used to replace a name with a token-string. For example,

```
#define _NFILE    20
```

sets to 20 the number of files a program can have open at one time. See **cpp**(1) for the complete list.

In the pages of the *Programmer's Reference Manual* there are about 45 different **.h** files named. The format of the **#include** statement for all these shows the file name enclosed in angle brackets (<>), as in:

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

The angle brackets tell the C preprocessor to look in the standard places for the file. In most systems the standard place is in the **/usr/include** directory. If you have some definitions or external declarations that you want to make available in several files, you can create a **.h** file with any editor, store it in a convenient directory and make it the subject of a **#include** statement such as the following:

```
#include "../defs/rec.h"
```

It is necessary, in this case, to provide the relative pathname of the file and enclose it in quotation marks (" "). Fully-qualified pathnames (those that begin with /) can create portability and organizational problems. An alternative to long or fully-qualified pathnames is to use the **-I***dir* preprocessor option when you compile the program. This option directs the preprocessor to search for **#include** files whose names are enclosed in quotation marks, first in the directory of the file being compiled, then in the directories named in the **-I** options, and finally in directories on the standard list. In addition, all **#include** files whose names are enclosed in angle brackets are first searched for in the list of directories named in the **-I** option and finally in the directories on the standard list.

Object File Libraries

It is common practice in the SYSTEM V/68 operating system to keep modules of compiled code (object files) in archives that are, by convention, designated by a **.a** suffix. System calls from Section 2 and the subroutines in subsections 3C and 3S of the *Programmer's Reference Manual* that are functions (as distinct from macros) are kept in archive file **libc.a**. In most systems, **libc.a** is found in the directory **/lib**. Many systems also have a directory **/usr/lib**. Where both **/lib** and **/usr/lib** occur, **/usr/lib** is apt to be used to hold archives that are related to specific applications.

During the link edit phase of the compilation and link edit process, copies of some of the object modules in an archive file are loaded with your executable code. By default the **cc** command that invokes the C compilation system causes the link editor to search **libc.a**. If you need to point the link editor to other libraries that are not searched by default, you do it by naming them explicitly on the command line with the **-l** option. The format of the **-l** option is **-l***x*, where *x* is the library name (9 characters maximum). For example, if your program includes functions from the **curses** screen control package, the option:

-lcurses

will cause the link editor to search for **/lib/libcurses.a** or **/usr/lib/libcurses.a** and use the first one it finds to resolve references in your program.

When you want to direct the order in which archive libraries are searched, you may use the **-L** *dir* option. Assuming the **-L** option appears on the command line ahead of the **-l** option, it directs the link editor to search the named directory for **lib***x***.a** before looking in **/lib** and **/usr/lib**. This is particularly useful if you are testing out a new version of a function that already exists in an archive in a standard directory. Its success is because, once having resolved a reference, the link editor stops looking. That's why the **-L** option, if used, should appear on the command line ahead of any **-l** specification.

Input/Output

We talked some about I/O earlier in this chapter in connection with system calls and subroutines. A whole set of subroutines constitutes the C language standard I/O package, and there are several system calls that deal with the same area. In this section we want to get into the subject in a little more detail and describe for you how to deal with input and output concerns in your C programs. First, let's briefly define I/O functions:

- creating and sometimes removing files
- opening and closing files used by your program
- transferring information from a file to your program (reading)
- transferring information from your program to a file (writing)

In this section we will describe some of the subroutines you might choose for transferring information, but the heaviest emphasis will be on dealing with files.

Three Files You Always Have

Programs are permitted to have several files open simultaneously. The number may vary from system to system; the most common maximum is 20. _NFILE in **stdio.h** specifies the number of standard I/O FILEs a program is permitted to have open.

Any program automatically starts off with three files. If you look again at Figure 2-4, about midway through you will see that **stdio.h** contains three **#define** directives that equate **stdin, stdout,** and **stderr** to the address of _iob[0], _iob[1], and _iob[2], respectively. The array _iob holds information dealing with the way standard I/O handles streams. It is a representation of the open file table in the control block for your program. The position in the array is a digit that is also known as the file descriptor. The default in SYSTEM V/68 is to associate all three of these files with your terminal.

The real significance is that functions and macros that deal with **stdin** or **stdout** can be used in your program with no further need to open or close files. For example, **gets**, cited above, reads a string from **stdin**; **puts** writes a null-terminated string to **stdout**. There are others that do the same (in slightly different ways: character at a time, formatted, etc.). You can specify that output be directed to **stderr** by using a function such as **fprintf**. This function works the same as **printf** except that it delivers its formatted output to a named stream, such as **stderr**. You can use the shell's redirection feature on the command line to read from or write into a named file. If you want to separate error messages from ordinary output being sent to **stdout** and thence possibly piped by the shell to a succeeding program, you can do it by using a function to handle the ordinary output and using a variation of the same function, one that names the stream, to handle error messages.

Named Files

Any files other than **stdin, stdout,** and **stderr** that are to be used by your program must be explicitly connected by you before the file can be read from or written to. This can be done using the standard library routine **fopen**. **fopen** takes a pathname (which is the name by which the file is known to the file system), asks the system to keep track of the connection, and returns a pointer that you then use in functions that do the reads and writes.

A structure is defined in **stdio.h** with a type of FILE. In your program you need to have a declaration such as:

```
FILE *fin;
```

The declaration says that **fin** is a pointer to a FILE. You can then assign the name of a particular file to the pointer with a statement in your program like this:

```
fin = fopen("filename", "r");
```

where **filename** is the pathname to open. The "**r**" means that the file is to be opened for reading. This argument is known as the **mode**. As you might suspect, there are modes for reading, writing, and both reading and writing.

Actually, the file open function is often included in an **if** statement such as:

```
if ((fin = fopen("filename", "r")) == NULL)
  (void)fprintf(stderr,"%s: Unable to open input file %s\n",
    argv[0],"filename");
```

that takes advantage of the fact that **fopen** returns a NULL pointer if it can't open the file.

Once the file has been successfully opened, the pointer **fin** is used in functions (or macros) to refer to the file. For example:

```
int c;
c = getc(fin);
```

brings in a character at a time from the file into an integer variable called **c**. The variable **c** is declared as an integer, even though we are reading characters, because the function **getc()** returns an integer. Getting a character is often incorporated into some flow-of-control mechanism such as:

```
while ((c = getc(fin)) != EOF)
     .
     .
     .
```

that reads through the file until EOF is returned. EOF, NULL, and the macro **getc** are all defined in **stdio.h**. The **getc** macro and others that make up the standard I/O package keep advancing a pointer through the buffer associated with the file; the operating system and the standard I/O subroutines are responsible for seeing that the buffer is refilled (or written to the output file if you are producing output) when the pointer reaches the end of the buffer. All these mechanics are mercifully invisible to the program and the programmer.

The function **fclose** is used to break the connection between the pointer in your program and the pathname. The pointer may then be associated with another file by another call to **fopen**. This reuse of a file descriptor for a different stream may be necessary if your program has many files to open. For output files it is good to issue an **fclose** call, because the call makes sure that all output has been sent from the output buffer before disconnecting the file. The system call **exit** closes all open files for you. It also gets you completely out of your process, however; so it is safe to use only when you are sure you are completely finished.

Low-level I/O and Why You Shouldn't Use It

The term "low-level I/O" is used to refer to the process of using system calls from Section 2 of the *Programmer's Reference Manual* rather than the functions and subroutines of the standard I/O package. We are going to postpone until Chapter 3 any discussion of when this might be advantageous. If you find as you go through the information in this chapter that it is a good fit with the objectives you have as a programmer, it is a safe assumption that you can work with C language programs in SYSTEM V/68 for a good many years without ever having a real need to use system calls to handle your I/O and file-accessing problems. Using low-level I/O is perilous because it is more system-dependent. Your programs are less portable and probably no more efficient.

System Calls for Environment or Status Information

Under some circumstances you might want to be able to monitor or control the environment in your computer. There are system calls that can be used for this purpose. Some of them are shown in Table 2-6. Since a manual page sometimes describes several related functions, the leftmost column in a row shows the function name that appears at the top of the manual page; any other names in the row are related functions described on the same page.

As you can see, many of the functions shown in Table 2-6 have equivalent shell commands. Shell commands can easily be incorporated into shell scripts to accomplish the monitoring and control tasks you may need to do. The functions are available, however, and may be used in C programs as part of the operating system/C Language interface. They are documented in Section 2 of the *Programmers' Reference Manual.*

TABLE 2-6. Environment and Status System Calls

Function Names			Purpose
chdir			Change working directory.
chmod			Change access permission of a file.
chown			Change owner and group of a file.
getpid	**getpgrp**	**getppid**	Get process IDs.
getuid	**geteuid**	**getgid**	Get user IDs.
ioctl			Control device.
link	**unlink**		Add or remove a directory entry.
mount	**umount**		Mount or unmount a file system.
nice			Change priority of a process.
stat	**fstat**		Get file status.
time			Get time.
ulimit			Get and set user limits.
uname			Get name of current operating system.

Processes

Whenever you execute an operating system command, you are initiating a process that is numbered and tracked by the operating system. A flexible feature of SYSTEM V/68 is that processes can be generated by other processes. This happens more often than you might be aware of. For example, when you log in to your system you are running a process, probably the shell. If you then use an editor such as **vi**, take the option of invoking the shell from **vi**, and execute the **ps** command, you will see a display something like that in Figure 2-6 (which shows the results of a **ps -f** command):

```
UID     PID     PPID    C    STIME      TTY     TIME    COMMAND
abc     24210   1       0    06:13:14   tty29   0:05    -sh
abc     24631   24210   0    06:59:07   tty29   0:13    vi c2.uli
abc     28441   28358   80   09:17:22   tty29   0:01    ps -f
abc     28358   24631   2    09:15:14   tty29   0:01    sh -i
```

Figure 2-6. Process Status Display

As you can see, user abc (who went through the steps described above) now has four processes active. It is an interesting exercise to trace the chain that is shown in the Process ID (PID) and Parent Process ID (PPID) columns. The shell that was started when user abc logged on is Process 24210; its parent is the initialization process (Process ID 1). Process 24210 is the parent of Process 24631, and so on.

The four processes in the example above are all shell-level commands, but you can spawn new processes from your own program. (Actually, when you issue the command from your terminal to execute a program you are asking the shell to start another process: your executable object module with all the functions and subroutines that were made a part of it by the link editor.)

You might think, "Well, it's one thing to switch from one program to another when I'm at my terminal working interactively with the computer; but why would a program want to run other programs; and if one does, why wouldn't I just put everything together into one big executable module?"

Overlooking the case where your program is itself an interactive application with diverse choices for the user, your program may need to run one or more other programs based on conditions it encounters in its own processing (making a trial balance at the end of the month, for example). The usual reasons why it might not be practical to create one monster executable are:

- The load module may get too big to fit in the maximum process size for your system.
- You may not have control over the object code of all the other modules you want to include.

Thus, there are legitimate reasons why new processes might need to be created. There are three ways to do it (described in the following subsections):

- **system**(3S)—request the shell to execute a command
- **exec**(2)—stop this process and start another
- **fork**(2)—start an additional copy of this process

system(3S)

The formal declaration of the **system** function looks like this:

```
#include <stdio.h>

int system(string)
char *string;
```

The function asks the shell to treat the string as a command line. The string can therefore be the name and arguments of any executable program or the operating system shell command. If the exact arguments vary from one execution to the next, you may want to use **sprintf** to format the string before issuing the **system** command. When the command has finished running, **system** returns the shell exit status to your program. Execution of your program waits for the completion of the command initiated by **system** and then picks up again at the next executable statement.

exec(2)

The name **exec** refers to a family of functions that includes **execv, execle, execve, execlp**, and **execvp**. They all have the function of transforming the calling process into a new process. The reason for the variety is to provide different ways of pulling together and presenting the arguments of the function. An example of one version (**execl**) follows:

```
execl("/bin/prog2", "prog", progarg1, progarg2, (char *)0);
```

For **execl**, the argument list is:

/bin/prog2	path name of new process file
prog	name the new process gets in its argv[0]
progarg1, progarg2	arguments to *prog2* as char *'s
(char *)0	null char pointer to mark end of arguments

Check the manual page in the *Programmer's Reference Manual* for the rest of the details. The key point of the **exec** family is that there is no return from a successful execution: the calling process is finished, the new process overlays the old. The new process also takes over the Process ID and other attributes of the old process. If the call to **exec** is unsuccessful, control is returned to your program with a return value of -1. You can check **errno** to learn why it failed. (See "Error Handling" later in this chapter.)

fork(2)

The **fork** system call creates a new process that is an exact copy of the calling process. The new process is known as the child process; the caller is known as the parent process. The one major difference between the two processes is that the child gets its unique process ID. When the **fork** process has completed successfully, it returns a 0 to the child process and the child's process ID to the parent. If the idea of having two identical processes seems strange, consider this:

- Because the return value is different between the child process and the parent, the program can contain the logic to determine different paths.
- The child process could say, "Okay, I'm the child. I'm supposed to issue an **exec** for an entirely different program."
- The parent process could say, "My child is going to be **exec**ing a new process. I'll issue a **wait** until I get word that the new process is finished."

To take this out of the storybook world where programs talk like people and into the world of C programming (where people talk like programs), your code might include statements like those shown in Figure 2-7.

```
#include <errno.h>

int ch_stat, ch_pid, status;
char *progarg1;
char *progarg2;
void exit();
extern int errno;

     if ((ch_pid = fork()) < 0)
     {
      /* Could not fork...
         check errno
      */
     }
     else if (ch_pid == 0)           /* child */
     {
      (void)execl("/bin/prog2","prog",progarg1,progarg2,(char *)0);
      exit(2); /* execl() failed */
     }
     else              /* parent */
     {
      while ((status = wait(&ch_stat)) != ch_pid)
      {
          if (status < 0 && errno == ECHILD)
         break;
          errno = 0;
      }
     }
```

Figure 2-7. Example of **fork**

Because the child process ID is taken over by the new **exec**'d process, the parent knows the ID. What this boils down to is a way of leaving one program to run another, returning to the point in the first program where processing ceased. This is exactly what the **system**(3S) function does, using this same procedure of **fork**ing and **exec**ing, with a **wait** in the parent.

Keep in mind that the fragment of code above includes a minimum amount of checking for error conditions. There is also potential confusion about open files and which program is writing to a file. Leaving out the possibility of named files, the new process created by the **fork** or **exec** has the three standard files that are automatically opened: **stdin**, **stdout**, and **stderr**. If the parent has buffered output that should appear before output from the child, the buffers must be flushed before the fork. Also, if the parent and the child process both read input from a stream, whatever is read by one process will be lost to the other. That is, once something has been delivered from the input buffer to a process the pointer has moved on.

Pipes

The idea of using pipes, a connection between the output of one program and the input of another, when working with commands executed by the shell is common in the SYSTEM V/68 operating system environment. For example, to learn the number of archive files in your system you might enter a command like:

```
echo /lib/*.a /usr/lib/*.a | wc -w
```

that first echoes all the files in **/lib** and **/usr/lib** that end in **.a**, then pipes the results to the **wc** command, which counts their number.

A feature of the operating system/C Language interface is the ability to establish pipe connections between your process and a command to be executed by the shell or between two cooperating processes. The first uses the **popen**(3S) subroutine that is part of the standard I/O package; the second requires the system call **pipe**(2).

popen is similar in concept to the **system** subroutine in that it causes the shell to execute a command. The difference is that once having invoked **popen** from your program, you have established an open line to a concurrently running process through a stream. You can send characters or strings to this stream with standard I/O subroutines just as you would to **stdout** or to a named file. The connection remains open until your program invokes the companion **pclose** subroutine. A common application of this technique is a pipe to a printer spooler, as shown in Figure 2-8.

```
#include <stdio.h>

main()
{
     FILE *pptr;
     char *outstring;

     if ((pptr = popen("lp","w")) != NULL)
     {
      for(;;)
      {     .
            .     /* Organize output */
            .
            (void)fprintf(pptr, "%s\n", outstring);
            .
            .
            .

      }
     .
     .
     .
     pclose(pptr);
     }
     .
     .
     .
}
```

Figure 2-8. Example of a **popen** Pipe

Error Handling

Within your C programs you must determine the appropriate level of checking for valid data and for acceptable return codes from functions and subroutines. If you use any of the system calls described in Section 2 of the *Programmer's Reference Manual,* you have a way in which you can find out the probable cause of a bad return value.

Operating system calls that are not able to complete successfully almost always return a value of -1 to your program. (If you look through the system calls in Section 2, you will see that there are a few calls for which no return value is defined; but they are the exceptions.) In addition to the -1 that is returned to the program, the unsuccessful system call places an integer in an externally declared variable, **errno**. You can determine the value in **errno** if your program contains the statement:

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

The value in **errno** is not cleared on successful calls, so your program should check it only if the system call returned a -1. The errors are described in **intro**(2) of the *Programmer's Reference Manual.*

The subroutine **perror**(3C) can be used to print an error message (on **stderr**) based on the value of **errno**.

Signals and Interrupts

Signals and interrupts are two words for the same thing. Both words refer to messages passed by the operating system to running processes. Generally, the effect is to cause the process to stop running. Some signals are generated if the process attempts to do something illegal; others can be initiated by a user against his or her own processes, or by the superuser against any process.

You can include the system call **kill** in your program to send signals to other processes running under your user-id. The format for the **kill** call is:

```
    kill(pid, sig)
```

where **pid** is the process number against which the call is directed, and **sig** is an integer from 1 to 19 that shows the intent of the message. The name "kill" is something of an overstatement; not all the messages have a "drop dead" meaning. Some of the available signals are shown in Figure 2-9 as they are defined in **<sys/signal.h>**.

```
#define SIGHUP  1  /* hangup */
#define SIGINT  2  /* interrupt (rubout) */
#define SIGQUIT 3  /* quit (ASCII FS) */
#define SIGILL  4  /* illegal instruction (not reset when caught)*/
#define SIGTRAP 5  /* trace trap (not reset when caught) */
#define SIGIOT  6  /* IOT instruction */
#define SIGABRT 6  /* used by abort, replace SIGIOT in the future */
#define SIGEMT  7  /* EMT instruction */
#define SIGFPE  8  /* floating point exception */
#define SIGKILL 9  /* kill (cannot be caught or ignored) */
#define SIGBUS  10 /* bus error */
#define SIGSEGV 11 /* segmentation violation */
#define SIGSYS  12 /* bad argument to system call */
#define SIGPIPE 13 /* write on a pipe with no one to read it */
#define SIGALRM 14 /* alarm clock */
#define SIGTERM 15 /* software termination signal from kill */
#define SIGUSR1 16 /* user defined signal 1 */
#define SIGUSR2 17 /* user defined signal 2 */
#define SIGCLD  18 /* death of a child */
#define SIGPWR  19 /* power-fail restart */

                    /* SIGWIND and SIGPHONE only used in SYSTEM V/68/PC */
/*#define SIGWIND 20*/ /* window change */
/*#define SIGPHONE 21*/ /* handset, line status change */

#define SIGPOLL 22 /* pollable event occurred */

#define NSIG    23 /* The valid signal number is from 1 to NSIG-1 */
#define MAXSIG  32 /* size of u_signal[], NSIG-1 <= MAXSIG*/
                   /* MAXSIG is larger than we need now. */
                   /* In the future, we can add more signal */
                   /* number without changing user.h */
```

Figure 2-9. Signal Numbers Defined in **/usr/include/sys/signal.h**

The **signal**(2) system call is designed to let you code methods of dealing with incoming signals. You have a three-way choice. You can a) accept whatever the default action is for the signal, b) have your program ignore the signal, or c) write a function of your own to deal with it.

Analysis/Debugging

SYSTEM V/68 provides several commands designed to help you discover the causes of problems in programs and to learn about potential problems.

To illustrate how these commands are used and the type of output they produce, we have constructed a sample program that opens and reads an input file and performs one to three subroutines according to options specified on the command line. This program does not do anything you couldn't do easily on your pocket calculator, but it does serve to illustrate some points. The source code is shown in Figure 2-10. The header file, **recdef.h**, is shown at the end of the source code.

The output produced by the various analysis and debugging tools illustrated in this section may vary slightly from one installation to another. The *Programmer's Reference Manual* is a good source of additional information about the contents of the reports produced.

```
       /* Main module -- restate.c */

#include <stdio.h>
#include "recdef.h"

#define TRUE     1
#define FALSE    0

main(argc, argv)
int argc;
char *argv[];
{
     FILE *fopen(), *fin;
     void exit();
     int getopt();
     int oflag = FALSE;
     int pflag = FALSE;
     int rflag = FALSE;
     int ch;
     struct rec first;
     extern int opterr;
     extern float oppty(), pft(), rfe();

           /* restate.c is continued on the next page */
```

Figure 2-10. Source Code for Sample Program (part 1 of 4)

```
                /* restate.c continued */

if (argc < 2)
{
        (void) fprintf(stderr, "%s: Must specify option\n",argv[0]);
        (void) fprintf(stderr, "Usage: %s -rpo\n", argv[0]);
        exit(2);
}

opterr = FALSE;
while ((ch = getopt(argc,argv,"opr")) != EOF)
{
        switch(ch)
        {
        case 'o':
                oflag = TRUE;
                break;
        case 'p':
                pflag = TRUE;
                break;
        case 'r':
                rflag = TRUE;
                break;
        default:
                (void) fprintf(stderr, "Usage: %s -rpo\n",argv[0]);
                exit(2);
        }
}
if ((fin = fopen("info","r")) == NULL)
{
(void) fprintf(stderr, "%s: cannot open input file %s\n",argv[0],"info");
exit(2);
}
```

Figure 2-10. Source Code for Sample Program (part 2 of 4)

```
                  /* restate.c continued */

     if (fscanf(fin, "%s%f%f%f%f%f%f",first.pname,&first.ppx,
     &first.dp,&first.i,&first.c,&first.t,&first.spx) != 7)
     {
       (void) fprintf(stderr,"%s: cannot read first record from %s\n",
          argv[0],"info");
       exit(2);
     }

     printf("Property: %s\n",first.pname);

     if(oflag)
          printf("  Opportunity Cost: $%#5.2f\n",oppty(&first));

     if(pflag)
          printf("  Anticipated Profit(loss): $%#7.2f\n",pft(&first));

     if(rflag)
          printf("  Return on Funds Employed: %#3.2f%%\n",rfe(&first));
}
                  /* End of Main Module -- restate.c */

                  /* Opportunity Cost -- oppty.c */
#include "recdef.h"

float
oppty(ps)
struct rec *ps;
{
     return(ps->i/12 * ps->t * ps->dp);
}
```

Figure 2-10. Source Code for Sample Program (part 3 of 4)

```
                    /* Profit -- pft.c */

#include "recdef.h"

float
pft(ps)
struct rec *ps;
{
     return(ps->spx - ps->ppx + ps->c);
}

               /* Return on Funds Employed -- rfe.c */

#include "recdef.h"

float
rfe(ps)
struct rec *ps;
{
     return(100 * (ps->spx - ps->c) / ps->spx);
}

                    /* Header File -- recdef.h */

struct rec {         /* To hold input  */
     char pname[25];
     float ppx;
     float dp;
     float i;
     float c;
     float t;
     float spx;
 } ;
```

Figure 2-10. Source Code for Sample Program (part 4 of 4)

Discussions of the analysis/debugging commands follow, using examples based on the program shown in Figure 2-10.

The cflow Command

The **cflow** command produces a chart of the external references in C, **yacc**, **lex**, and assembly language files. Using the modules of our sample program, the command:

cflow restate.c oppty.c pft.c rfe.c

produces the output shown in Figure 2-11.

```
1 main: int(), <restate.c 11>
2       fprintf: <>
3       exit: <>
4       getopt: <>
5       fopen: <>
6       fscanf: <>
7       printf: <>
8       oppty: float(), <oppty.c 7>
9       pft: float(), <pft.c 7>
10      rfe: float(), <rfe.c 8>
```

Figure 2-11. cflow Output, No Options

The **-r** option looks at the caller-callee relationship from the other side. It produces the output shown in Figure 2-12.

```
1    exit: <>
2        main : <>
3    fopen: <>
4        main : 2
5    fprintf: <>
6        main : 2
7    fscanf: <>
8        main : 2
9    getopt: <>
10       main : 2
11   main: int(), <restate.c 11>
12   oppty: float(), <oppty.c 7>
13       main : 2
14   pft: float(), <pft.c 7>
15       main : 2
16   printf: <>
17       main : 2
18   rfe: float(), <rfe.c 8>
19       main : 2
```

Figure 2-12. cflow Output, Using **-r** Option

The **-ix** option causes external and static data symbols to be included. Our sample program has only one such symbol, **opterr**. The output is shown in Figure 2-13.

```
1    main: int(), <restate.c 11>
2         fprintf: <>
3         exit: <>
4         opterr: <>
5         getopt: <>
6         fopen: <>
7         fscanf: <>
8         printf: <>
9         oppty: float(), <oppty.c 7>
10        pft: float(), <pft.c 7>
11        rfe: float(), <rfe.c 8>
```

Figure 2-13. cflow Output, Using **-ix** Option

Combining the **-r** and the **-ix** options produces the output shown in Figure 2-14.

```
1   exit: <>
2        main : <>
3   fopen: <>
4        main : 2
5   fprintf: <>
6        main : 2
7   fscanf: <>
8        main : 2
9   getopt: <>
10       main : 2
11  main: int(), <restate.c 11>
12  oppty: float(), <oppty.c 7>
13       main : 2
14  opterr: <>
15       main : 2
16  pft: float(), <pft.c 7>
17       main : 2
18  printf: <>
19       main : 2
20  rfe: float(), <rfe.c 8>
21       main : 2
```

Figure 2-14. cflow Output, Using **-r** and **-ix** Options

The ctrace Command

The **ctrace** command lets you follow the execution of a C program statement by statement. It takes a **.c** file as input and inserts statements in the source code to print out variables as each program statement is executed. You must direct the output of this process to a temporary **.c** file. The temporary file is then used as input to **cc**. When the resulting **a.out** file is executed, it produces output that can tell you a lot about what is going on in your program.

Options give you the ability to limit the number of times through loops. You can also include functions in your source file that turn the trace off and on so you can limit the output to portions of the program that are of particular interest.

The **ctrace** command accepts only one source code file as input. To use our sample program to illustrate, it is necessary to execute the following four commands:

```
ctrace restate.c > ct.main.c
ctrace oppty.c > ct.op.c
ctrace pft.c > ct.p.c
ctrace rfe.c > ct.r.c
```

The names of the output files are completely arbitrary. Use any names that are convenient for you. The names must end in **.c**, since the files are used as input to the C compilation system.

```
cc -o ct.run ct.main.c ct.op.c ct.p.c ct.r.c
```

Now the command:

```
ct.run -opr
```

produces the output shown in Figure 2-15. The command above will cause the output to be directed to your terminal (**stdout**). It is probably a good idea to direct it to a file or to a printer so you can refer to it.

```
 8 main(argc, argv)
23  if (argc < 2)
    /* argc == 2 */
30  opterr = FALSE;
    /* FALSE == 0 */
    /* opterr == 0 */
31  while ((ch = getopt(argc,argv,"opr")) != EOF)
    /* argc == 2 */
    /* argv == 15729316 */
    /* ch == 111 or 'o' or "t" */
32  {
33       switch(ch)
         /* ch == 111 or 'o' or "t" */
35       case 'o':
36            oflag = TRUE;
              /* TRUE == 1 or "h" */
              /* oflag == 1 or "h" */
37            break;
48  }
31  while ((ch = getopt(argc,argv,"opr")) != EOF)
    /* argc == 2 */
    /* argv == 15729316 */
    /* ch == 112 or 'p' */
32  {
33       switch(ch)
         /* ch == 112 or 'p' */
38       case 'p':
39            pflag = TRUE;
              /* TRUE == 1 or "h" */
              /* pflag == 1 or "h" */
40            break;
48  }
31  while ((ch = getopt(argc,argv,"opr")) != EOF)
    /* argc == 2 */
    /* argv == 15729316 */
    /* ch == 114 or 'r' */
32  {
33       switch(ch)
         /* ch == 114 or 'r' */
41       case 'r':
42            rflag = TRUE;
              /* TRUE == 1 or "h" */
              /* rflag == 1 or "h" */
43            break;
48  }
```

Figure 2-15. ctrace Output (part 1 of 2)

```
31  while ((ch = getopt(argc,argv,"opr")) != EOF)
    /* argc == 2 */
    /* argv == 15729316 */
    /* ch == -1 */
49  if ((fin = fopen("info","r")) == NULL)
    /* fin == 140200 */
54  if (fscanf(fin, "%s%f%f%f%f%f%f",first.pname,&first.ppx,
    &first.dp,&first.i,&first.c,&first.t,&first.spx) != 7)
    /* fin == 140200 */
    /* first.pname == 15729528 */
61  printf("Property: %s0,first.pname);
    /* first.pname == 15729528 or "Linden_Place" */ Property: Linden_Place

63  if(oflag)
    /* oflag == 1 or "h" */
64       printf("  Opportunity Cost: $%#5.2f0,oppty(&first));
 5 oppty(ps)
 8  return(ps->i/12 * ps->t * ps->dp);
    /* ps->i == 1069044203 */
    /* ps->t == 1076494336 */
    /* ps->dp == 1088765312 */  Opportunity Cost: $4476.87
66  if(pflag)
    /* pflag == 1 or "h" */
67       printf("  Anticipated Profit(loss): $%#7.2f0,pft(&first));
 5 pft(ps)
 8  return(ps->spx - ps->ppx + ps->c);
    /* ps->spx == 1091649040 */
    /* ps->ppx == 1091178464 */
    /* ps->c == 1087409536 */  Anticipated Profit(loss): $85950.00

69  if(rflag)
    /* rflag == 1 or "h" */
70       printf("  Return on Funds Employed: %#3.2f%%0,rfe(&first));
 6 rfe(ps)
 9  return(100 * (ps->spx - ps->c) / ps->spx);
    /* ps->spx == 1091649040 */
    /* ps->c == 1087409536 */  Return on Funds Employed: 94.00%

   /* return */
```

Figure 2-15. ctrace Output (part 2 of 2)

Using a program that runs successfully is not the optimal way to demonstrate **ctrace**. It would be more helpful to have an error in the operation that could be detected by **ctrace**. It would seem that this utility might be most useful in cases where the program runs to completion but the output is not as expected.

The cxref Command

The **cxref** command analyzes a group of C source code files and builds a cross-reference table of the automatic, static, and global symbols in each file.

The command:

cxref -c -o cx.op restate.c oppty.c pft.c rfe.c

produces the output shown in Figure 2-16 in a file named **cx.op**. The **-c** option causes the reports for the four **.c** files to be combined in one cross-reference file.

```
restate.c:


oppty.c:


pft.c:


rfe.c:

  SYMBOL       FILE                         FUNCTION     LINE

BUFSIZ       /usr/include/stdio.h           --           *9
EOF          /usr/include/stdio.h           --           49 *50
             restate.c                      --           31
FALSE        restate.c                      --           *6 15 16 17 30
FILE         /usr/include/stdio.h           --           *29 73 74
             restate.c                      main         12
L_ctermid    /usr/include/stdio.h           --           *80
L_cuserid    /usr/include/stdio.h           --           *81
L_tmpnam     /usr/include/stdio.h           --           *83
NULL         /usr/include/stdio.h           --           46 *47
             restate.c                      --           49
P_tmpdir     /usr/include/stdio.h           --           *82
TRUE         restate.c                      --           *5 36 39 42
_IOEOF       /usr/include/stdio.h           --           *41
_IOERR       /usr/include/stdio.h           --           *42
_IOFBF       /usr/include/stdio.h           --           *36
_IOLBF       /usr/include/stdio.h           --           *43
_IOMYBUF     /usr/include/stdio.h           --           *40
```

Figure 2-16. cxref Output, Using **-c** Option (part 1 of 5)

```
 SYMBOL        FILE                          FUNCTION     LINE

_IONBF        /usr/include/stdio.h          --           *39
_IOREAD       /usr/include/stdio.h          --           *37
_IORW         /usr/include/stdio.h          --           *44
_IOWRT        /usr/include/stdio.h          --           *38
_NFILE        /usr/include/stdio.h          --           2 *3 73
_SBFSIZ       /usr/include/stdio.h          --           *16
_base         /usr/include/stdio.h          --           *26
_bufend()
              /usr/include/stdio.h          --           *57
_bufendtab    /usr/include/stdio.h          --           *78
_bufsiz()
              /usr/include/stdio.h          --           *58
_cnt          /usr/include/stdio.h          --           *20
_file         /usr/include/stdio.h          --           *28
_flag         /usr/include/stdio.h          --           *27
_iob          /usr/include/stdio.h          --           *73
              restate.c                     main         25 26 45 51 57
_ptr          /usr/include/stdio.h          --           *21
argc          restate.c                     --           8
              restate.c                     main         *9 23 31
argv          restate.c                     --           8
              restate.c                     main         *10 25 26 31 45 51 57
c             ./recdef.h                    --           *6
              pft.c                         pft          8
              restate.c                     main         55
              rfe.c                         rfe          9
ch            restate.c                     main         *18 31 33
clearerr()
              /usr/include/stdio.h          --           *67
ctermid()
              /usr/include/stdio.h          --           *77
cuserid()
              /usr/include/stdio.h          --           *77
dp            ./recdef.h                    --           *4
              oppty.c                       oppty        8
              restate.c                     main         55
exit()
              restate.c                     main         *13 27 46 52 58
```

Figure 2-16. cxref Output, Using **-c** Option (part 2 of 5)

```
  SYMBOL         FILE                          FUNCTION      LINE

fdopen()
               /usr/include/stdio.h            --            *74
feof()
               /usr/include/stdio.h            --            *68
ferror()
               /usr/include/stdio.h            --            *69
fgets()
               /usr/include/stdio.h            --            *77
fileno()
               /usr/include/stdio.h            --            *70
fin            restate.c                       main          *12 49 54
first          restate.c                       main          *19 54 55 61 64 67 70
fopen()
               /usr/include/stdio.h            --            *74
               restate.c                       main          12 49
fprintf        restate.c                       main          25 26 45 51 57
freopen()
               /usr/include/stdio.h            --            *74
fscanf         restate.c                       main          54
ftell()
               /usr/include/stdio.h            --            *75
getc()
               /usr/include/stdio.h            --            *61
getchar()
               /usr/include/stdio.h            --            *65
getopt()
               restate.c                       main          *14 31
gets()
               /usr/include/stdio.h            --            *77
i              ./recdef.h                      --            *5
               oppty.c                         oppty         8
               restate.c                       main          55
lint           /usr/include/stdio.h            --            60
main()
               restate.c                       --            *8
```

Figure 2-16. cxref Output, Using **-c** Option (part 3 of 5)

```
  SYMBOL        FILE                           FUNCTION      LINE

oflag         restate.c                      main          *15 36 63
oppty()
              oppty.c                        --            *5
              restate.c                      main          *21 64
opterr        restate.c                      main          *20 30
p             /usr/include/stdio.h           --            *57 *58 *61 62
*62 63 64 67 *67 68 *68 69 *69 70 *70
pdp11         /usr/include/stdio.h           --            11
pflag         restate.c                      main          *16 39 66
pft()
              pft.c                          --            *5
              restate.c                      main          *21 67
pname         ./recdef.h                     --            *2
              restate.c                      main          54 61
popen()
              /usr/include/stdio.h           --            *74
ppx           ./recdef.h                     --            *3
              pft.c                          pft           8
              restate.c                      main          54
printf        restate.c                      main          61 64 67 70
ps            oppty.c                        --            5
              oppty.c                        oppty         *6 8
              pft.c                          --            5
              pft.c                          pft           *6  8
              rfe.c                          --            6
              rfe.c                          rfe           *7 9
putc()
              /usr/include/stdio.h           --            *62
putchar()
              /usr/include/stdio.h           --            *66
rec           ./recdef.h                     --            *1
              oppty.c                        oppty         6
              pft.c                          pft           6
              restate.c                      main          19
              rfe.c                          rfe           7
```

Figure 2-16. cxref Output, Using **-c** Option (part 4 of 5)

```
  SYMBOL        FILE                        FUNCTION     LINE

rewind()
              /usr/include/stdio.h          --           *76
rfe()
              restate.c                     main         *21 70
              rfe.c                         --           *6
rflag         restate.c                     main         *17 42 69
setbuf()
              /usr/include/stdio.h          --           *76
spx           ./recdef.h                    --           *8
              pft.c                         pft          8
              restate.c                     main         55
              rfe.c                         rfe          9
stderr        /usr/include/stdio.h          --           *55
              restate.c                     --           25 26 45 51 57
stdin         /usr/include/stdio.h          --           *53
stdout        /usr/include/stdio.h          --           *54
t             ./recdef.h                    --           *7
              oppty.c                       oppty        8
              restate.c                     main         55
tempnam()
              /usr/include/stdio.h          --           *77
tmpfile()
              /usr/include/stdio.h          --           *74
tmpnam()
              /usr/include/stdio.h          --           *77
u370          /usr/include/stdio.h          --           5
u3b           /usr/include/stdio.h          --           8 19
u3b5          /usr/include/stdio.h          --           8 19
vax           /usr/include/stdio.h          --           8 19
x             /usr/include/stdio.h          --           *62 63 64 66 *66
```

Figure 2-16. cxref Output, Using **-c** Option (part 5 of 5)

The lint Command

The **lint** command looks for features in a C program that are apt to cause execution errors, that are wasteful of resources, or that create problems of portability.

The command:

lint restate.c oppty.c pft.c rfe.c

produces the output shown in Figure 2-17.

```
restate.c:

restate.c
==============
(71)  warning: main() returns random value to invocation environment
oppty.c:
pft.c:
rfe.c:


==============
function returns value which is always ignored
    printf
```

Figure 2-17. lint Output

This command has options that will produce additional information. Check the *User's Reference Manual*. The error messages give you the line numbers of some items you may want to review.

The prof Command

The **prof** command produces a report on the amount of execution time spent in various portions of your program and the number of times each function is called. The program must be compiled with the **-p** option. When a program that was compiled with that option is run, a file called **mon.out** is produced. **mon.out** and **a.out** (or whatever name identifies your executable file) are input to the **prof** command.

The sequence of steps needed to produce a profile report for our sample program is as follows:

Step 1: Compile the programs with the **-p** option:

```
cc -p restate.c oppty.c pft.c rfe.c
```

Step 2: Run the program to produce a file **mon.out**.

```
a.out -opr
```

Step 3: Execute the **prof** command:

```
prof a.out
```

The example of the output of this last step is shown in Figure 2-18. The figures may vary from one run to another. You will also notice that programs of small size, like that used in the example, produce statistics that are not overly helpful.

```
%Time     Seconds     Cumsecs     #Calls     msec/call     Name
 50.0        0.03        0.03          3            8.     fcvt
 20.0        0.01        0.04          6            2.     atof
 20.0        0.01        0.05          5            2.     write
 10.0        0.00        0.05          1            5.     fwrite
  0.0        0.00        0.05          1            0.     monitor
  0.0        0.00        0.05          1            0.     creat
  0.0        0.00        0.05          4            0.     printf
  0.0        0.00        0.05          2            0.     profil
  0.0        0.00        0.05          1            0.     fscanf
  0.0        0.00        0.05          1            0.     _doscan
  0.0        0.00        0.05          1            0.     oppty
  0.0        0.00        0.05          1            0.     _filbuf
  0.0        0.00        0.05          3            0.     strchr
  0.0        0.00        0.05          1            0.     strcmp
  0.0        0.00        0.05          1            0.     ldexp
  0.0        0.00        0.05          1            0.     getenv
  0.0        0.00        0.05          1            0.     fopen
  0.0        0.00        0.05          1            0.     _findiop
  0.0        0.00        0.05          1            0.     open
  0.0        0.00        0.05          1            0.     main
  0.0        0.00        0.05          1            0.     read
  0.0        0.00        0.05          1            0.     strcpy
  0.0        0.00        0.05         14            0.     ungetc
  0.0        0.00        0.05          4            0.     _doprnt
  0.0        0.00        0.05          1            0.     pft
  0.0        0.00        0.05          1            0.     rfe
  0.0        0.00        0.05          4            0.     _xflsbuf
  0.0        0.00        0.05          1            0.     _wrtchk
  0.0        0.00        0.05          2            0.     _findbuf
  0.0        0.00        0.05          2            0.     isatty
  0.0        0.00        0.05          2            0.     ioctl
  0.0        0.00        0.05          1            0.     malloc
  0.0        0.00        0.05          1            0.     memchr
  0.0        0.00        0.05          1            0.     memcpy
  0.0        0.00        0.05          2            0.     sbrk
  0.0        0.00        0.05          4            0.     getopt
```

Figure 2-18. prof Output

The size Command

The **size** command produces information on the number of bytes occupied by the three sections (text, data, and bss) of a common object file when the program is brought into main memory to be run. Here are the results of one invocation of the **size** command with our object file as an argument.

```
11832 + 3872 + 2240 = 17944
```

Don't confuse this number with the number of characters in the object file that appears when you do an **ls -l** command. That figure includes the symbol table and other header information that is not used at run time.

The strip Command

The **strip** command removes the symbol and line-number information from a common object file. When you issue this command the number of characters shown by the **ls -l** command approaches the figure shown by the **size** command, but still includes some header information that is not counted as part of the .text, .data, or .bss section. After the **strip** command has been executed, it is no longer possible to use the file with the **sdb** command.

The sdb Command

The **sdb** command stands for "Symbolic Debugger," which means you can use the symbolic names in your program to pinpoint where a problem has occurred. You can use **sdb** to debug C, FORTRAN 77, or PASCAL programs. There are two basic ways to use **sdb**: by running your program under control of **sdb** or by using **sdb** to rummage through a core image file left by a program that failed. The first way lets you see what the program is doing up to the point at which it fails (or lets you skip around the failure point and proceed with the run). The second method lets you check the status at the moment of failure, which may or may not disclose the reason the program failed.

Chapter 15 contains a tutorial on **sdb** that describes the interactive commands you can use to work your way through your program. There are two key things you need to do when using it:

1. Compile your programs with the **-g** option, which causes additional information to be generated for use by **sdb**.
2. Run your program under **sdb** with the command:

   ```
   sdb myprog - srcdir
   ```

   where **myprog** is the name of your executable file (**a.out** is the default) and

srcdir is an optional list of the directories where source code for your modules may be found. The hyphen between the two arguments keeps **sdb** from looking for a core image file.

Program-Organizing Utilities

The following three utilities are helpful in keeping your programming work organized effectively.

The make Command

When you have a program that is made up of more than one module of code, you begin to run into problems of keeping track of which modules are up to date and which need to be recompiled when changes are made in another module. The **make** command is used to ensure that dependencies between modules are recorded, so that a change in a module results in the recompilation of its dependent programs. Even control of a program as simple as the sample shown in Figure 2-10 is made easier by using **make**.

The **make** utility requires a description file that you create with an editor. The description file (also referred to by its default name: **makefile**) contains the information used by **make** to keep a target file current. The target file is typically an executable program. A description file contains three types of information:

dependency information	tells the **make** utility the relationships among the modules that constitute the target program.
executable commands	generate the target program. The **make** command uses the dependency information to determine which executable commands should be passed to the shell for execution.
macro definitions	provide a shorthand notation within the description file to make maintenance easier. Macro definitions can be overridden by information from the command line when the **make** command is entered.

The **make** command works by checking the "last changed" time of the modules named in the description file. When **make** finds a component that has been changed more recently than modules that depend on it, the specified commands (usually compilations) are passed to the shell for execution.

The **make** command takes three kinds of arguments: options, macro definitions, and target filenames. If no description filename is given as an option on the command line, **make** searches the current directory for a file named **makefile** or **Makefile**. Figure 2-19 shows a **makefile** for our sample program.

```
OBJECTS = restate.o oppty.o pft.o rfe.o
all: restate
restate: $(OBJECTS)
     $(CC) $(CFLAGS) $(LDFLAGS) $(OBJECTS) -o restate

$(OBJECTS): ./recdef.h

clean:
     rm -f $(OBJECTS)

clobber: clean
     rm -f restate
```

Figure 2-19. make Description File

The following things are worth noticing in this description file:

- It identifies the target, **restate**, as being dependent on the four object modules. Each of the object modules in turn is defined as being dependent on the header file, **recdef.h**, and by default, on its corresponding source file.
- A macro, OBJECTS, is defined as a convenient shorthand reference to the component modules.

Whenever testing or debugging results in a change to one of the components of **restate**, for example, a command such as the following should be entered:

make CFLAGS=-g restate

This has been a very brief overview of the **make** utility. More on **make** appears in Chapter 3, and a detailed description of **make** can be found in Chapter 13.

The Archive

The most common use of an archive file, although not the only one, is to hold object modules that make up a library. The library can be named on the link editor command line (or with a link editor option on the **cc** command line). This procedure causes the link editor to search the symbol table of the archive file when attempting to resolve references.

The **ar** command is used to create an archive file, to manipulate its contents, and to maintain its symbol table. The structure of the **ar** command is a little different from the normal arrangement of command line options. When you enter the **ar** command, you include a one-character key from the set **drqtpmx** that defines the type of action you intend. The key may be combined with one or more additional characters from the set **vuaibcls** that modify the way the requested operation is performed. The makeup of the command line is:

ar *-key* [*posname*] *afile* [*name*]...

where *posname* is the name of a member of the archive and may be used with some optional key characters to make sure that the files in your archive are in a particular order. The *afile* argument is the name of your archive file. By convention, the suffix **.a** is used to indicate the named file is an archive file. (The file **libc.a**, for example, is the archive file that contains many of the object files of the standard C subroutines.) One or more *names* may be furnished. These identify files that are subjected to the action specified in the *key*.

We can make an archive file to contain the modules used in our sample program, **restate**. The command to do this is:

```
ar -rv rste.a restate.o oppty.o pft.o rfe.o
```

If these are the only **.o** files in the current directory, you can use shell metacharacters as follows:

```
ar -rv rste.a *.o
```

Either command will produce this feedback:

```
a - restate.o
a - oppty.o
a - pft.o
a - rfe.o
ar: creating rste.a
```

The **nm** command is used to get a variety of information from the symbol table of common object files. The object files can be, but don't have to be, in an archive file. Figure 2-20 shows the output of this command when executed with the **-f** (for "full") option on the archive we just created. The object files were compiled with the **-g** option.

```
Symbols from rste.a[restate.o]

Name           Value    Class             Type          Size     Line    Section

.0fake        |       |          |              strtag|struct  |  16  |
restate.c     |       |file      |                    |        |      |
_cnt          |      0|strmem    |                 int|        |      |
_ptr          |      4|strmem    |              *Uchar|        |      |
_base         |      8|strmem    |              *Uchar|        |      |
_flag         |     12|strmem    |                char|        |      |
_file         |     13|strmem    |                char|        |      |
.eos          |       |endstr    |                    |     16 |      |
rec           |       |strtag    |              struct|     52 |      |
pname         |      0|strmem    |            char[25]|     25 |      |
ppx           |     28|strmem    |               float|        |      |
dp            |     32|strmem    |               float|        |      |
i             |     36|strmem    |               float|        |      |
c             |     40|strmem    |               float|        |      |
t             |     44|strmem    |               float|        |      |
spx           |     48|strmem    |               float|        |      |
.eos          |       |endstr    |                    |     52 |      |
main          |      0|extern    |               int( )|    520 |      | .text
.bf           |     10|fcn       |                    |        |  11  | .text
argc          |      0|argm't    |                 int|        |      |
argv          |      4|argm't    |              **char|        |      |
fin           |      0|auto      |    *struct-.0fake|     16 |      |
oflag         |      4|auto      |                 int|        |      |
pflag         |      8|auto      |                 int|        |      |
rflag         |     12|auto      |                 int|        |      |
ch            |     16|auto      |                 int|        |      |
```

Figure 2-20. nm Output, with **-f** Option (part 1 of 5)

```
Symbols from rste.a[restate.o]

Name           Value    Class        Type              Size   Line   Section

first        |    20 | auto     |       struct-rec |   52 |      |
.ef          |   518 | fcn      |                  |      |   61 | .text
FILE         |       | typdef   | struct-.0fake    |   18 |      |
.text        |     0 | static   |                  |   31 |   39 | .text
.data        |   520 | static   |                  |      |    4 | .data
.bss         |   824 | static   |                  |      |      | .bss
_iob         |     0 | extern   |                  |      |      |
fprintf      |     0 | extern   |                  |      |      |
exit         |     0 | extern   |                  |      |      |
opterr       |     0 | extern   |                  |      |      |
getopt       |     0 | extern   |                  |      |      |
fopen        |     0 | extern   |                  |      |      |
fscanf       |     0 | extern   |                  |      |      |
printf       |     0 | extern   |                  |      |      |
oppty        |     0 | extern   |                  |      |      |
pft          |     0 | extern   |                  |      |      |
rfe          |     0 | extern   |                  |      |      |
```

Figure 2-20. nm Output, with **-f** Option (part 2 of 5)

```
Symbols from rste.a[oppty.o]

Name        Value     Class          Type           Size   Line   Section

oppty.c    |        | file     |                   |      |      |
rec        |        | strtag   |            struct |  52  |      |
pname      |      0 | strmem   |          char[25] |  25  |      |
ppx        |     28 | strmem   |             float |      |      |
dp         |     32 | strmem   |             float |      |      |
i          |     36 | strmem   |             float |      |      |
c          |     40 | strmem   |             float |      |      |
t          |     44 | strmem   |             float |      |      |
spx        |     48 | strmem   |             float |      |      |
.eos       |        | endstr   |                   |  52  |      |
oppty      |      0 | extern   |           float() |  64  |      |  .text
.bf        |     10 | fcn      |                   |      |    7 |  .text
ps         |      0 | argm't   |      *struct-rec  |  52  |      |
.ef        |     62 | fcn      |                   |      |    3 |  .text
.text      |      0 | static   |                   |    4 |    1 |  .text
.data      |     64 | static   |                   |      |      |  .data
.bss       |     72 | static   |                   |      |      |  .bss
```

Figure 2-20. nm Output, with **-f** Option (part 3 of 5)

```
Symbols from rste.a[pft.o]

Name          Value     Class                 Type      Size   Line   Section

pft.c       |        | file     |                     |      |      |
rec         |        | strtag   |              struct |  52  |      |
pname       |      0 | strmem   |           char[25]  |  25  |      |
ppx         |     28 | strmem   |               float |      |      |
dp          |     32 | strmem   |               float |      |      |
i           |     36 | strmem   |               float |      |      |
c           |     40 | strmem   |               float |      |      |
t           |     44 | strmem   |               float |      |      |
spx         |     48 | strmem   |               float |      |      |
..eos       |        | endstr   |                     |  52  |      |
pft         |      0 | extern   |            float()  |  60  |      |  .text
..bf        |     10 | fcn      |                     |      |    7 |  .text
ps          |      0 | argm't   |         *struct-rec |  52  |      |
..ef        |     58 | fcn      |                     |      |    3 |  .text
..text      |      0 | static   |                     |   4  |      |  .text
..data      |     60 | static   |                     |      |      |  .data
..bss       |     60 | static   |                     |      |      |  .bss
```

Figure 2-20. nm Output, with **-f** Option (part 4 of 5)

```
Symbols from rste.a[rfe.o]

Name          Value     Class              Type          Size    Line    Section

rfe.c               |  file    |                        |       |       |
rec                 |  strtag  |               struct   |  52   |       |
pname            0  |  strmem  |             char[25]   |  25   |       |
ppx             28  |  strmem  |                float   |       |       |
dp              32  |  strmem  |                float   |       |       |
i               36  |  strmem  |                float   |       |       |
c               40  |  strmem  |                float   |       |       |
t               44  |  strmem  |                float   |       |       |
spx             48  |  strmem  |                float   |       |       |
.eos                |  endstr  |                        |  52   |       |
rfe              0  |  extern  |              float()   |  68   |       |  .text
.bf             10  |  fcn     |                        |       |     8 |  .text
ps               0  |  argm't  |          *struct-rec   |  52   |       |
.ef             64  |  fcn     |                        |       |     3 |  .text
.text            0  |  static  |                        |    4  |     1 |  .text
.data           68  |  static  |                        |       |       |  .data
.bss            76  |  static  |                        |       |       |  .bss
```

Figure 2-20. nm Output, with **-f** Option (part 5 of 5)

For **nm** to work on an archive file, all of the contents of the archive have to be object modules. If you have stored other things in the archive, you will get the message :

```
nm: rste.a  bad magic
```

when you try to execute the command.

Use of SCCS by Single-User Programmers

The Source Code Control System (SCCS) is a set of programs designed to keep track of different versions of programs. When a program has been placed under control of SCCS, only a single copy of any one version of the code can be retrieved for editing at a given time. When program code is changed and the program returned to SCCS, only the changes are recorded. Each version of the code is identified by its SID, or **S**CCS **ID**entifying number. By specifying the SID when the code is extracted from the SCCS file, it is possible to return to an earlier version. If an early version is extracted with the intent of editing it and returning it to SCCS, a new branch of the development tree is started. The set of programs that make up SCCS appear as operating system commands. The commands are:

admin
get
delta
prs
rmdel
cdc
what
sccsdiff
comb
val

It is most common to think of SCCS as a tool for control of large programming projects. It is, however, entirely possible for any individual user of the operating system to set up a private SCCS system. Chapter 14 is an SCCS user's guide.

2